VAX Pascal Reference Manual

Order Number: AA–L369D–TE

December 1989

This document contains information about selected programming tasks using the VAX
Pascal programming language.

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

CDA	MASSBUS	VAX RMS
DDIF	PrintServer 40	VAXstation
DEC	Q-bus	VMS
DECnet	ReGIS	VT
DECUS	ULTRIX	XUI
DECwindows	UNIBUS	
DIGITAL	VAX	
LN03	VAXcluster	digital™

The following is a third-party trademark:

PostScript is a registered trademark of Adobe Systems, Inc.

ZK4569

Contents

Chapter 1   Language Elements

Chapter 2   Data Types and Values

iii

Chapter 3   The Declaration Section

Chapter 6   Procedures and Functions

Chapter 7   Program Structure and Scope

Chapter 8   Predeclared Routines

Chapter 9   Input and Output Processing

Chapter 10   Attributes

Chapter 11   Directives

Appendix A   ASCII Character Set

Appendix B   Language Syntax Summary

Appendix C   Compatibility of VAX Pascal Versions

Appendix D Summary of VAX Pascal Extensions

Appendix E Description of Implementation Features

Appendix F Error Detection

Index

Figures

Tables

Preface

This document is the complete description of the VAX Pascal programming language. It contains information on the VAX Pascal language syntax and semantics, on VAX Pascal adherance to various Pascal standards, and on the VAX Pascal extensions to those standards.

Intended Audience

This manual is intended for experienced applications programmers with a basic understanding of the Pascal language. Some familiarity with your operating system is helpful. This is not a tutorial manual.

Document Structure

This manual consists of the following chapters and appendixes:

- Chapter 1 describes lexical elements.
- Chapter 2 describes data types.
- Chapter 3 describes declaration sections.
- Chapter 4 describes expressions.
- Chapter 5 describes statements.
- Chapter 6 describes user-written procedures and functions.
- Chapter 7 describes programs and modules.
- Chapter 8 describes predeclared procedures and functions.
- Chapter 9 describes the predeclared procedures and functions that perform input and output.
- Chapter 10 describes attributes.

- Chapter 11 describes directives.
- Appendix A describes the ASCII character set.
- Appendix B provides syntax drawings of VAX Pascal constructs.
- Appendix C describes changes made to the VAX Pascal language after VAX Pascal Version 1.0.
- Appendix D describes the VAX Pascal extensions to the Pascal standards.
- Appendix E describes the VAX Pascal implementation features that the Pascal standards allow each implementation to define.
- Appendix F describes how the VAX Pascal compiler detects errors defined by the Pascal standard.

Associated Documents

The following documents may also be useful when programming in VAX Pascal:

- *VAX Pascal Reference Supplement for VMS Systems*—Provides information on the programming environment beyond the scope of the VAX Pascal language syntax and semantics. It contains reference information on VMS operating-system features, VAX architecture information, and environment-specific affects of VAX Pascal language features.
- *VAX Pascal User Manual*—Provides information about programming tasks, about using VAX Pascal features in conjunction with one another, and about increasing the efficiency of program execution.
- *VAX Pascal Installation Guide*—Provides information on how to install VAX Pascal on your operating system.
- VMS operating system manuals—Provide full information about the system. The *VMS Master Index* briefly describes all VMS system documentation, defines the intended audience for each manual, and provides a synopsis of each manual's contents.

Conventions

Table 1 presents the conventions used in this manual.

Table 1:   Conventions Used in This Manual

Convention	Meaning
{ }	Large braces enclose lists from which you must choose one item. For example: $\left\{ \begin{array}{l} \text{expression} \\ \text{statement} \end{array} \right\}$
. . .	A horizontal ellipsis means that the item preceding the ellipsis can be repeated. For example: digit . . .
{ }, . . .	Braces followed by a comma and a horizontal ellipsis mean that you can repeat the enclosed item one or more times, separating two or more items with commas. For example: {label}, . . .
{ }; . . .	Braces followed by a semicolon and a horizontal ellipsis mean that you can repeat the enclosed item one or more times, separating two or more items with semicolons. For example: REPEAT {statement}; . . . UNTIL expression
. . .	A vertical ellipsis in a figure or example means that not all of the statements are shown.
[ ]	Square brackets mean that the statement syntax requires the square bracket characters. This notation is used with arrays, sets, and attribute lists. For example: ARRAY[index1]

Table 1 (Cont.):   Conventions Used in This Manual

Convention	Meaning
[[ ]]	Double brackets enclose items that are optional. For example: EOLN [[(file-variable)]]
PROGRAM WRITELN	Uppercase letters and special symbols in syntax descriptions indicate reserved words and predeclared identifiers. For example: BEGIN END
temp : INTEGER; PRED(n)	Lowercase letters represent user-defined identifiers or elements that you must replace according to the description in the text.
Extensions	In the hardcopy version, VAX Pascal extensions to the Extended Pascal standard are color coded in blue. In the online version, these extensions are shaded.
module	A term that appears in bold is defined in the glossary in the *VAX Pascal User Manual*.

In this manual, complex examples and syntax diagrams have been divided into several lines to make them easy to read. VAX Pascal does not require that you format your programs in any particular way.

New and Changed Features

VAX Pascal Version 4.0 supports most (but not all) of the proposed Extended Pascal standard. Also, VAX Pascal Version 4.0 accepts all programs that compiled with previous versions of the compiler. VAX Pascal Version 4.0 accepts environment files created by previous versions of the compiler (2.0 through 3.n). However, previous versions of the compiler do not accept environment files created by VAX Pascal Version 4.0.

The following are specific changes made to the VAX Pascal language since VAX Pascal Version 3.5:

- **Schema Types**—This feature, defined by the Extended Pascal standard, is a user-defined construct that provides a template for a family of distinct data types. A schema type definition contains one or more formal discriminants that take the place of specific boundary values or variant-record selectors. By specifying boundary or selector values to a schema type, you form a valid data type; the provided boundary or selector values are called actual discriminants. See Section 2.5 for more information.

- **Declarations Checks**—This feature of the compiler allows you to use a compilation switch or an attribute within your program to tell VAX Pascal the run-time checking of legal schema definitions and uplevel GOTO usage. See Section 10.2.7 or the *VAX Pascal Reference Supplement for VMS Systems* for more information.

- **Initial-State Specifiers**—This feature, defined by the Extended Pascal standard, allows you to use the VALUE reserved word to initialize variables and record fields, and to initialize type identifiers so that the value is propagated to all variables created from that type. See Section 3.5 for more information.

- **Structured Constructors**—This feature, defined by the Extended Pascal standard, allows you to specify values of a structured type, and to use run-time expressions in constructors. See Section 2.4 for more information.

- **Access to Structured Function-Return Values**—This feature, defined by the Extended Pascal standard, allows you to index arrays, to select fields, and to dereference pointers that are returned by functions at the time of the function call; you do not need to use an intermediate variable to store the return value. See Section 6.2 for more information.

- **AND_THEN and OR_ELSE Logical Operators**—These new logical operators, defined by the Extended Pascal standard, allow you to guarantee left-to-right evaluation and short-circuiting in the expression. See Section 4.2.3 for more information.

- **DATE, TIME and GETTIMESTAMP Routines**—The DATE and TIME functions and the GETTIMESTAMP procedure, which are defined by the Extended Pascal standard, allow you to use a standard method of obtaining and using date and time values. See Chapter 8 for more information.

- **EQ, GE, GT, LE, LT, and NE Functions**—These functions, defined by the Extended Pascal standard, allow you to compare strings without the compiler padding shorter strings with blanks. See Chapter 8 for more information.

- **FOR-IN Statement**—This feature, defined by the Extended Pascal Standard, allows you to use an incrementor to select from each item in a set. See Section 5.5 for more information.

- **Extended-Digit Integer Notation**—This enhancement, defined by the Extended Pascal standard, allows you to specify integer constants using a base number, followed by a number sign (#), followed by the extended digits. See Section 2.1.1 for more information.

- **MFPR and MTPR Routines**—These features allow you to manipulate a VAX internal processor register. See Sections 8.46 and 8.48.

- **Module Initialization and Finalization (TO BEGIN DO and TO END DO Declaration Sections)**—This feature, defined by the Extended Pascal standard, allows you to specify a statement to be executed either before (TO BEGIN DO) or after (TO END DO) the activation of a module. See Sections 3.3 and 3.4 for more information.

- **STRING Predefined Schema**—This feature, defined by the Extended Pascal standard, is a predefined schema type that you can use to declare variable-length character strings. See Section 2.6.3 for more information.

- **WITH Statement Enhancement**—This feature, defined by the Extended Pascal standard, allows the WITH statement to open the scope of discriminants of a schema type as well as the scope of fields of a record variable. See Section 5.11 for more information.

- **/DESIGN Qualifier**—This new PASCAL command qualifier directs the compiler to accept design phase placeholders and comments as valid program elements. See the *VAX Pascal Reference Supplement for VMS Systems* for more information.

- **/STANDARD Qualifier Enhancement**—This PASCAL command qualifier has the new option EXTENDED, which controls flagging of VAX Pascal extensions to the Extended Pascal standard. See the *VAX Pascal Reference Supplement for VMS Systems* for more information.

- **/NOOPTIMIZE Qualifier Enhancement**—This PASCAL command qualifier now guarantees left-to-right evaluation order with full evaluation of both operands of AND and OR Boolean operators. See Section 10.2.26 or the *VAX Pascal Reference Supplement for VMS Systems* for more information.

Most of the information in the *VAX Pascal Reference Supplement for VMS Systems* appeared in the *VAX Pascal User Manual* for VAX Pascal Version 3.5. A few sections appeared in the *VAX Pascal Reference Manual* for VAX Pascal Version 3.5.

Most of the information in the *VAX Pascal User Manual* is new. The program optimization chapter and some of the examples of calling system services appeared in the *VAX Pascal User Manual* for VAX Pascal Version 3.5.

The following are changes made to the reference manual since VAX Pascal Version 3.5:

- **Dictionary Approach**—The predeclared routines, input and output routines, attributes, and directives are presented in alphabetical order instead of grouped by category.

- **Removal of Programming Examples Appendix**—All detailed programming examples are now located in the *VAX Pascal User Manual*.

- **Removal of Sharing Declarations and Definitions Section**—This information has been moved to the *VAX Pascal User Manual*.

- **Removal of System or Architecture Information**—Any information that has to do with an operating system or machine architecture feature has been removed and placed in the *VAX Pascal Reference Supplement for VMS Systems*. Such information includes compilation information, INTEGER and REAL number ranges, system-specific input/output information, system-specific parameter passing information, and other programming information that is relevant to your particular machine or operating system.

- **Removal of Task Oriented Information**—All tutorial and task-oriented information has been removed and placed in the *VAX Pascal User Manual*.

- **Removal of CDD Information**—All Common Data Dictionary information has been moved to the *VAX Pascal Reference Supplement for VMS Systems* (%DICTIONARY syntax remains in Chapter 11).

For More Information:

- On Pascal standards (Section 1.1)

- On VAX Pascal extensions to the Pascal standards (Appendix E)

Chapter 1

Language Elements

VAX Pascal is a general-purpose programming language. This chapter describes the following information and components of the VAX Pascal language:

* Pascal standards (Section 1.1)
* Lexical elements (Section 1.2)
* Comments (Section 1.3)

1.1 Pascal Language Standards

The VAX Pascal compiler accepts programs that comply with two standards and a subset of programs that comply with a third. Also, VAX Pascal provides features that are not part of any standard (called **extensions**). If you require portable code, do not use the VAX Pascal extensions.

1.1.1 Unextended Pascal Standards

The **unextended Pascal** standards are as follows:

* American National Standard ANSI/IEEE770X3.97-1983 (ANSI)
* International Standard ISO 7185-1983(E) (ISO)

VAX Pascal accepts programs that comply to either standard. In the VAX Pascal documentation set, the term "unextended Pascal" applies to both the ANSI and ISO standards.

VAX Pascal contains FIPS-109 (Federal Information Processing Standard) validation support.

The ISO standard is divided into two levels of standardization: Level 0 and Level 1. An example of a technical difference between the ANSI standard and the ISO standard is that ANSI does not include conformant arrays, while ISO standard Level 0 does not, but Level 1 does.

VAX Pascal has passed the validation suite for Pascal compilers. It received a CLASS A certificate for both levels of the ISO standard as well as the ANSI standard. CLASS A certificates are given to compilers with a fully conforming implementation.

For More Information:

- On VAX Pascal extensions to unextended Pascal (Appendix D)
- On VAX Pascal implementation-dependent features (Appendix E)
- On VAX Pascal error processing as defined by the standard (Appendix F)
- On flagging nonstandard constructs during compilation (*VAX Pascal Reference Supplement for VMS Systems*)

1.1.2 Extended Pascal Standard

The **Extended Pascal** standard is a proposed standard at the time of this writing. A joint American (X3J9/IEEE P770) and International (ISO IEC/JTC1/SC22/WG2) committee is developing this standard. In the VAX Pascal documentation set, the term "the Pascal standard" refers to this proposed standard. Extended Pascal is a superset of the unextended Pascal standards. For your convenience, the VAX Pascal extensions to the Extended Pascal standard are printed in blue in this manual.

Since VAX Pascal supports many (but not all) Extended Pascal standard features, it cannot compile all programs that comply with Extended Pascal.

For More Information:

- On VAX Pascal support for Extended Pascal features (Appendix D)
- On flagging nonstandard constructs during compilation (*VAX Pascal Reference Supplement for VMS Systems*)

1.2 Lexical Elements

This section discusses **lexical elements** of the VAX Pascal language.

1.2.1 Character Set

VAX Pascal uses the extended American Standard Code for Information Interchange (ASCII) character set. This extended ASCII character set contains 256 characters, which include the following:

- Uppercase letters A through Z and lowercase letters a through z
- Integers 0 through 9
- Special characters, such as the ampersand (&), question mark (?), and equal sign (=)
- Nonprinting characters, such as the space, tab, line feed, carriage return, and form feed (use of these characters may improve the legibility of your programs)
- Extended, unspecified characters with numeric codes from 128 to 255

Each ASCII character corresponds to a numeric value.

The VAX Pascal compiler does not distinguish between uppercase and lowercase characters except when they appear inside apostrophes. For example, the word PROGRAM has the same meaning when written as any of the following:

```
PROGRAM
PRogrAm
program
```

The following characters, however, represent different values:

```
'b'
'B'
```

For More Information:

For information on the ASCII character set, see Appendix A.

1.2.2 Special Symbols

Special symbols represent operators, delimiters, and other syntactic elements. Some symbols are composed of more than one character; you cannot place a space between the characters of these special symbols. Table 1–1 lists VAX Pascal special symbols.

Table 1–1: Special Symbols

Symbol	Name	Symbol	Name
'	Apostrophe	<=	Less than or equal to
:=	Assignment operator	–	Minus sign
[ ] or (. .)	Brackets	*	Multiplication
:	Colon	<>	Not equal
,	Comma	( )	Parentheses
(* *) or { }	Comments	%	Percent
/	Division	.	Period
=	Equal sign	+	Plus sign
**	Exponentiation	^ or @	Pointer
>	Greater than	;	Semicolon
>=	Greater than or equal to	..	Subrange operator
<	Less than	::	Type cast operator

1.2.3 Reserved Words

Reserved words are used to designate data types, directives, identifiers, specifiers, statements, and operators. You cannot redefine these identifiers. Table 1–2 presents the VAX Pascal reserved words.

Table 1–2: Reserved Words

AND	END	NIL	%STDESCR
ARRAY	FILE	NOT	%SUBTITLE
BEGIN	FOR	OF	THEN
CASE	FUNCTION	OR	%TITLE
CONST	GOTO	PACKED	TO
%DESCR	IF	PROCEDURE	TYPE
%DICTIONARY	%IMMED	PROGRAM	UNTIL
DIV	IN	RECORD	VAR
DO	%INCLUDE	%REF	WHILE

Table 1–2 (Cont.): Reserved Words

DOWNTO	LABEL	REPEAT	WITH
ELSE	MOD	SET	

The manuals in the VAX Pascal documentation set show these reserved words in uppercase letters. If you choose, you can express them in mixed case or lowercase in your programs.

Table 1–3 presents the **redefinable reserved words** that are used to name operators and identifiers. You can redeclare these words, but, if you do, the language extension becomes unavailable within the block in which you redeclare the word.

Table 1–3: Redefinable Reserved Words

AND_THEN	OTHERWISE	VARYING
MODULE	REM	
OR_ELSE	VALUE	

This manual shows redefinable reserved words in uppercase letters. If you choose, you can express them in mixedcase or lowercase in your programs.

1.2.4   Identifiers

An identifier is a combination of letters, digits, dollar signs ($), and underscores (_) that conforms to the following restrictions:

*   An identifier cannot start with a digit.

*   An identifier cannot contain spaces or special symbols.

*   The first 31 characters of an identifier must denote a unique name within the block in which the identifier is declared. An identifier longer than 31 characters generates a warning message. The compiler ignores characters beyond the thirty-first character.

*   An identifier cannot start or end with an underscore, nor can two adjacent underscores be used within an identifier. VAX Pascal allows both cases of underscore use and generates an informational message if /STANDARD=EXTENDED is specified.

The following examples show valid and invalid identifiers:

Valid:

```
For2n8
MAX_WORDS
upto
LOGICAL_NAME_TABLE        {Unique in first}
Logical_Name_Scanner      { 31 characters}
SYS$CREMBX
```

Invalid:

```
4Awhile                                      {Starts with a digit}
up&to                                        {Contains an ampersand}
YEAR_END_87_MASTER_FILE_TOTAL_DISCOUNT        {Not unique in first}
Year_End_87_Master_File_Total_Dollars        { 31 characters}
```

Table 1–4 presents the VAX Pascal **predeclared identifiers** that name data types, symbolic constants, file variables, procedures, and functions. You can redefine a predeclared identifier, but if you do, the original declaration becomes unavailable within the block in which you redeclared the word.

Table 1–4:   Predeclared Identifiers

— Read across —		
ABS	ADD_INTERLOCKED	ADDRESS
ARCTAN	ARGUMENT	ARGUMENT_LIST_LENGTH
BIN	BITNEXT	BIT_OFFSET
BITSIZE	BOOLEAN	BYTE_OFFSET
CARD	CHAR	CHR
CLEAR_INTERLOCKED	CLOCK	CLOSE
COS	CREATE_DIRECTORY	DATE
DBLE	DEC	DELETE
DELETE_FILE	DISPOSE	DOUBLE
EOF	EOLN	EPSDOUBLE
EPSQUADRUPLE	EPSREAL	EQ
ESTABLISH	EXP	EXPO
EXTEND	FALSE	FIND
FIND_FIRST_BIT_CLEAR	FIND_FIRST_BIT_SET	FIND_MEMBER

Table 1–4 (Cont.): Predeclared Identifiers

— Read across —		
FIND_NONMEMBER	FINDK	GE
GET	GETTIMESTAMP	GT
HALT	HEX	IADDRESS
INDEX	INPUT	INT
INTEGER	LE	LENGTH
LINELIMIT	LN	LOCATE
LOWER	LT	MAX
MAXCHAR	MAXDOUBLE	MAXINT
MAXQUADRUPLE	MAXREAL	MAXUNSIGNED
MIN	MINDOUBLE	MINQUADRUPLE
MINREAL	NE	NEW
NEXT	NIL	OCT
ODD	OPEN	ORD
OUTPUT	PACK	PAD
PAGE	PRED	PRESENT
PUT	QUAD	QUADRUPLE
READ	READLN	READV
REAL	RENAME_FILE	RESET
RESETK	REVERT	REWRITE
ROUND	SET_INTERLOCKED	SIN
SINGLE	SIZE	SNGL
SQR	SQRT	STATUS
STATUSV	STRING	SUBSTR
SUCC	TEXT	TIME
TIMESTAMP	TRUE	TRUNC
TRUNCATE	UAND	UDEC
UFB	UINT	UNDEFINED
UNLOCK	UNOT	UNPACK
UNSIGNED	UOR	UPDATE

Table 1–4 (Cont.): Predeclared Identifiers

	— **Read across** —	
UPPER	UROUND	UTRUNC
UXOR	WRITE	WRITELN
WRITEV	XOR	ZERO

This manual shows predeclared identifiers in uppercase letters. If you choose, you can express them in mixed case or lowercase in your programs.

1.3 Comments

Comments document the actions or elements of a program. The text of a comment can contain any ASCII character except a nonprinting control character, such as an ESCAPE character. You can place comments anywhere in a program that white space can appear.

You signify a comment with braces or with a parenthesis and asterisk pair, as follows:

```
{ This is a comment. }
(* This is a comment, too. *)
```

VAX Pascal allows you to mix the two symbol pairs in one comment, as follows:

```
{ The delimiters of this comment do not match. *)
(* VAX Pascal allows you to mix delimiters in this way. }
```

VAX Pascal does not allow you to nest comments. The following example causes a compile-time error because the comment ends at the first closing delimiter (}).

```
(* Cannot { nest  comments inside } of comments like this *)
```

1.4 Page Breaks and Form Feeds

A page break or form feed character can appear anywhere in your program except on a line with text surrounding the form feed. For example, the following lines are legal:

```
[FF]  %TITLE 'Variable Declarations'
end;  [FF]
[FF]
[FF]  VAR  [FF]
```

However, the following line generates an error:

```
BEGIN  [FF]  END.
```

The page break does not affect the meaning of the program, but causes a page to eject at the corresponding line in a listing file.

Chapter 2

Data Types and Values

Every piece of data that is created or manipulated by a VAX Pascal program has a **data type**. The data type determines the range of values, set of valid operations, and maximum storage allocation for each piece of data.

This chapter provides information on the following:

- Ordinal types (Section 2.1)
- Real types (Section 2.2)
- Pointer type (Section 2.3)
- Structured types (Section 2.4)
- Schema types (Section 2.5)
- String schemas and types (Section 2.6)
- Predefined structured and schema types (Section 2.7)
- Static and nonstatic types (Section 2.8)
- Rules of type compatibility (Section 2.9)

For More Information:

- On user-defined types and the TYPE section (Section 3.5)
- On variable declarations and the VAR section (Section 3.7)

2.1 Ordinal Types

This section describes the ordinal types that are predefined by VAX Pascal and user-defined ordinal types (types that require you to provide identifiers or boundary values to completely define the data type).

The ranges of values for these types are ordinal in nature; the values are ordered so that each has a unique ordinal value indicating its position in a list of all the values of that type. There is a one-to-one correspondence between the values in an ordinal type and the set of positive integers.

2.1.1 INTEGER Type

The range of the INTEGER values consists of positive and negative integer values, and of the value 0; the range boundaries may depend on the architecture of the machine you are using. The largest possible value of the INTEGER type is represented by the predefined constant MAXINT; the smallest possible value of the INTEGER type is represented by the value of the expression –MAXINT. The INTEGER type has the following form:

$$
[\![\left\{ \begin{array}{c} - \\ + \end{array} \right\}]\!] \left\{ \begin{array}{l} \text{decimal-number} \\ \text{base-number\#}[\![']\!] \text{ extended-digit}[\![']\!] \\ \% \left\{ \begin{array}{c} b \\ o \\ x \end{array} \right\} [\![']\!] \text{extended-digit}[\![']\!] \end{array} \right\}
$$

decimal-number
Specifies an integer in conventional Pascal integer notation. You cannot specify commas or decimal points. Examples of decimal notation are as follows:

```
17          0          89324
```

base-number
Specifies the base of the number. VAX Pascal accepts numbers in bases 2 through 36.

extended-digit
Specifies the notation that is appropriate for the specified base.

b
o
x
Specifies an integer in either binary (base 2), octal (base 8), or hexadecimal (base 16) notation. VAX Pascal accepts either uppercase or lowercase letters.

Using Extended-Digit Notation:

You can use **extended-digit notation** in the same way you use the conventional integer notation, with the following exceptions:

- Extended-digit values cannot be used as labels.

- Extended-digit notation for INTEGER objects cannot be used to express numbers outside the range of 0 to MAXINT. (To express signed numbers, place the unary plus operator (+) or the unary minus operator (−) in front of the notation; setting or clearing the high order bit does not set or clear the sign bit.)

VAX Pascal allows the use of spaces and tabs to make the extended-digit notation easier to read. To use spaces and tabs, enclose the extended digit in single quotation marks (' '). The following are integer values in the extended-digit notation:

```
2#10000011
2#'1000 0011'
32#1J
-16#'7FFF FFFF'
```

VAX Pascal provides another extended integer convention only for the sake of compatibility with previous versions of the language. The following are extended integer values in the VAX Pascal specific notation:

```
%b'1000 0011'
%O'7712'
-%x'DEC'
```

For More Information:

- On unary operators (Section 4.2)

- On value of MAXINT (*VAX Pascal Reference Supplement for VMS Systems*)

2.1.2 UNSIGNED Type

The UNSIGNED data type consists of nonnegative integer values. The largest possible value of the UNSIGNED type is represented by the predefined constant MAXUNSIGNED; the smallest possible value of the UNSIGNED type is 0. The UNSIGNED data type is a VAX Pascal extension that is provided to facilitate systems programming using certain operating systems. Since this data type is not standard, you should not use it for every application involving nonnegative integers.

When a VAX Pascal program contains an integer constant greater than MAXINT, the constant is treated as being of type UNSIGNED. Unsigned integers can be written using the conventional notation, the extended-digit notation, or the VAX Pascal specific notation.

Integer expressions whose constant value is not greater than MAXINT and not less then –MAXINT are always treated as being of type INTEGER. To force an integer constant to become UNSIGNED rather than INTEGER, use the UINT predeclared function.

For More Information:

- On INTEGER notations (Section 2.1.1)

- On the UINT function (Section 8.78)

- On the UNSIGNED value range (*VAX Pascal Reference Supplement for VMS Systems*)

2.1.3 CHAR Type

The CHAR data type consists of single character values from the ASCII character set. The largest possible value of the CHAR type is the predefined constant MAXCHAR.

To specify a character constant, enclose a printable ASCII character in single quotation marks. To specify the single-quote character, enclose two single quotation marks in single quotation marks. Each of the following is a valid character constant:

```
'A'
'z'
'0'    { This is the character 0, not the integer value 0 }
''''   { The apostrophe character }
'?'
```

The ORD function accepts parameters of type CHAR. The function return value is the ordinal value of the character in the ASCII character set.

You can specify a nonprinting character, such as a control character, by writing an empty string followed immediately by the ordinal value of the character in the ASCII character set, or by using the CHR function followed by the ordinal value of the character in the ASCII character set. The following examples show the two ways to specify the bell control character:

```
'' (7)
CHR( 7 )
```

For More Information:

- On the ORD function (Section 8.54)
- On the CHR function (Section 8.13)
- On the ASCII character set (Appendix A)
- On character strings (Section 2.6)

2.1.4 BOOLEAN Type

Boolean values are the result of testing relationships for truth or validity. The BOOLEAN data type consists of the two predeclared identifiers FALSE and TRUE. The expression ORD(FALSE) results in the value 0; ORD(TRUE) returns the integer 1.

The relational operators operate on the ordinal, real, string, or set expressions, and produce a Boolean result.

For More Information:

- On the ORD function (Section 8.54)
- On relational operators (Section 4.2.2)

2.1.5 Enumerated Types

An enumerated type is a user-defined ordered set of constant values specified by identifiers. It has the following form:

({enumerated-identifier},....)

enumerated-identifier
The identifier of the enumerated type being defined. VAX Pascal allows a maximum of 65,535 identifiers in an enumerated type.

The values of an enumerated type begin with the value 0 and follow a left-to-right order. Subsequent identifiers have a value one greater than the identifier preceding it. Consider the following:

```
TYPE
    Seasons = ( Spring, Summer, Fall, Winter );
VAR
    Some_Seasons : Seasons VALUE Winter;   {Initialized}
```

In this enumerated type, Spring (value 0) and Summer (value 1) are less than Fall (value 2) because they precede Fall in the list of constant values. Winter (value 3) is greater than Fall because it follows Fall.

The ORD function accepts expressions of an enumerated type.

An identifier in an enumerated type cannot be defined for any other purpose in the same block. Consider the following:

```
TYPE
    Seasons2 = ( Fall, Winter, Spring );
```

This enumerated type cannot be defined in the same block as the previous type, because the identifiers Spring, Fall, and Winter would not be unique.

For More Information:

For information on the ORD function, see Section 8.54.

2.1.6 Subrange Types

A subrange type is user-defined and specifies a limited portion of another ordinal type (called the base type). It has the following form:

lower-bound..upper-bound

lower-bound
A constant expression or a formal discriminant identifier that establishes the lower limit of the subrange.

upper-bound
A constant expression or formal discriminant identifier that establishes the upper limit of the subrange. The value of the upper bound must be greater than or equal to the value of the lower bound.

The base type can be any enumerated or predefined ordinal type. The values in the subrange type appear in the same order as they are in the base type. For instance, the result of the ORD function applied to a value of a subrange type is the ordinal value that is associated with the relative position of the value in the base type, not in the subrange type.

You can use a subrange type anywhere in a program that its base type is legal. A value of a subrange type is converted to a value of its base type before it is used in an operation. All rules that govern the operations performed on an ordinal type pertain to subranges of that type.

Consider the following:

```
TYPE
    Day = ( Mon, Tues, Wed, Thur, Fri, Sat, Sun );
    Weekday = Mon..Fri;             {subrange of base type Day}
    Weekend = Sat..Sun;             {subrange of base type Day}
    Digit   = '0'..'9';             {subrange of base type CHAR}
    Month   = 1..31;                {subrange of base type INTEGER}
```

For More Information:

- On the ORD function (Section 8.54)

- On the TYPE section (Section 3.5)

- On discriminant identifiers in subranges (Section 2.5)

- On using the CHECK attribute for subrange checking (Section 10.2.7)

2.2 Real Types

The following are real data types that are predefined by VAX Pascal:

- REAL

- SINGLE

- DOUBLE

- QUADRUPLE

The REAL, SINGLE, DOUBLE, and QUADRUPLE data types specify real number values with different degrees of precision. The types REAL and SINGLE are synonymous; both designate single-precision real values. The type DOUBLE designates double-precision real values. The type QUADRUPLE designates quadruple-precision real values. (In this manual, the term "REAL type" refers to both the REAL and SINGLE types.)

Table 2–1 presents the identifiers that are predefined by VAX Pascal for use with the real data types.

Table 2–1: Predefined Identifiers For Use With Real Data Types

Identifiers	Values
MAXREAL MAXDOUBLE MAXQUADRUPLE	Maximum values of the REAL, DOUBLE, and QUADRUPLE data types.
MINREAL MINDOUBLE MINQUADRUPLE	Minimum values of the REAL, DOUBLE, and QUADRUPLE data types.
EPSREAL EPSDOUBLE EPSQUADRUPLE	Smallest value of the REAL, DOUBLE, and QUADRUPLE data types, such that $(1.0 + EPSREAL) >$ 1.0, $(1.0D0 + EPSDOUBLE) > 1.0D$, and $(1.0Q0 + EPSQUADRUPLE) > 1.0Q0$.

To express REAL numbers, you can use either decimal or exponential notation. To express DOUBLE or QUADRUPLE numbers, you must use exponential notation.

To express REAL numbers in decimal notation, use the set of decimal digits and a decimal point. At least one digit must appear on either side of the decimal point. The following are valid real numbers in decimal notation:

```
   2.4
893.2497
   8.0
   0.0
```

To express real numbers in exponential notation, you include a real number or an integer, an uppercase or lowercase letter indicating the type of precision, and an integer exponent with its minus sign or optional plus sign. For example:

```
   2.3E2
  10.0E-1
9.14159e0
```

Table 2–2 presents the letters that indicate precision in exponential notation.

Table 2–2:   Precision in Exponential Notation

Letters	Meaning
E or e	Single-precision real number. The integer exponent following this letter specifies the power of 10.
D or d	Double-precision real number. All double-precision numbers in your program must appear in this exponential format; otherwise, the compiler reverts to single-precision representation.
Q or q	Quadruple-precision real number. All quadruple-precision numbers in your program must appear in this exponential format; otherwise, the compiler reverts to single-precision format.

To express negative real numbers in exponential notation, use the negation operator (–). Remember that a negative real number such as –4.5E+3 is not a constant, but is actually an expression consisting of the negation operator (–) and the real number 4.5E+3. Use caution when expressing negative real numbers in complex expressions.

For More Information:

*   On operators (Section 4.2)
*   On DOUBLE precisions:
    *   —   G_FLOATING attribute (Section 10.2.14)
    *   —   Compilation switches (*VAX Pascal Reference Supplement for VMS Systems*)
*   On the real value ranges (*VAX Pascal Reference Supplement for VMS Systems*)

2.3 Pointer Type

A pointer type allows you to refer to a dynamic variable. Dynamic variables do not have lifetimes that are strictly related to the scope of a routine, module, or program; you can create and eliminate them at various times during program execution. Also, pointer types clearly define the type of an object, but you can create or eliminate objects during program execution. The syntax of a pointer type is as follows:

^ [[attribute-list]] base-type-identifier

attribute-list

One or more identifiers that provide additional information about the base type.

base-type-identifier

The type identifier of the dynamic variable to which the pointer refers. The base type can be any type name or schema name. (If the base type is an undiscriminated schema type, you need to supply actual discriminants when you call the NEW function.)

Unlike other variables, dynamic variables do not have identifiers. Instead, you access them indirectly with pointers.

When you use pointers, you call the procedure NEW to allocate storage for dynamic variables. You call the procedure DISPOSE to deallocate this storage.

A variable of a pointer type refers to a dynamic variable of the base type and is said to be associated with that type. In the following example, the pointer variable Ptr is associated with a record of type My_Rec:

```
TYPE
   My_Rec = RECORD
      Name : STRING( 30 );
      Age  : INTEGER;
   END VALUE [Name: 'Barney Frank'; Age: 29]; {Initialized}
VAR
   Ptr : ^My_Rec;

{In executable section:}
NEW( Ptr );
```

To reference the dynamic variable to which a pointer refers, you write the pointer variable name followed by a circumflex (^). The following example assigns values to the record variable Ptr^:

```
Ptr^ := My_Rec[Name: 'David Leavitt'; age: 65];
```

Pointers assume values through initialization, assignment, the READ procedure, and the NEW procedure. The value of a pointer is either the storage address of a dynamic variable or the predeclared identifier NIL. NIL indicates that the pointer does not currently refer to a dynamic variable.

A file referenced by a pointer is not closed until the execution of the program terminates or until the dynamic variable is deallocated with the DISPOSE procedure. If you do not want the file to remain open throughout program execution, you must use the CLOSE procedure to close it.

The following example declares the pointer variable Ptr as a pointer to an integer and initializes Ptr to NIL:

```
VAR
    Ptr : ^INTEGER VALUE NIL;
```

For More Information:

- On the NEW procedure (Section 8.50)
- On the DISPOSE procedure (Section 8.23)
- On records (Section 2.4.2)
- On pointers to schema types (Section 2.5)
- On linked lists (*VAX Pascal User Manual*)

2.4 Structured Types

The structured data types are user defined and consist of **components**. Each component of a structured data type has its own data type; components can be any type.

To express values of structured objects (arrays, records, and sets), you can use a list of values called **constructors**. Constructors are valid in the TYPE, CONST, VAR, and executable sections of your program. Examples of valid constructors are provided in examples throughout the following sections. The following sections also contain examples that show how to assign values to individual components of structured objects.

To save storage space, you can specify PACKED before any structured type identifier except VARYING OF CHAR (for instance, PACKED ARRAY, PACKED RECORD, and PACKED SET). Defining PACKED structured types causes the compiler to economize storage by storing the structure in as few bits as possible. Keep in mind, however, that a packed data item is not compatible with a data item that is not packed. Also, accessing components of some packed structures may be slower than accessing components of unpacked structures.

VAX Pascal also provides predefined structured types for your use (for instance, to more easily manipulate date and time information).

For More Information:

- On string data types (Section 2.6)
- On VARYING OF CHAR (Section 2.6.2)
- On predefined structured types (Section 2.7)
- On array constructors (Section 2.4.1.1)
- On record constructors (Section 2.4.2.2)
- On set constructors (Section 2.4.3.1)

2.4.1 ARRAY Types

An **array** is a group of components (called **elements**) that all have the same data type and share a common identifier. An individual element of an array is referred to by an ordinal index (or **subscript**) that designates the element's position (or order) in the array. An array type has the following form:

[[PACKED]] ARRAY [{[[attribute-list]] index-type},...] OF   [[attribute-list]] component-type

attribute-list
One or more identifiers that provide additional information about the component type.

index-type
The type of the index, which can be any ordinal type or discriminated ordinal schema type.

component-type
The type of the array components, which can be any type. The components of an array can be another array.

The indexes of an array must be of an ordinal type. However, specifying INTEGER as the index type may cause the memory request to exceed available memory space. To use integer values as indexes, you must specify an integer subrange in the data type definition (unless you are using a conformant-array parameter).

You can use an array component (unless your array components are of a FILE type) anywhere in a program that a variable of the component type is allowed. Also, the only operation defined for entire array objects is the assignment operation (:=) (unless your array components are of a FILE type).

To refer to an array component, you specify the name of the array variable
(or the name of an object whose result, when used as an expression, is of an
array type), followed by an index value enclosed in brackets ([]). Consider
the following:

```
TYPE
    Count = ARRAY[1..10] OF INTEGER;   {Array type of 10 integers}
VAR
    Numbers : Count;                   {Array variable}
{In the executable section:}
Numbers[5] := 18;   {Assigns the value 18 to the fifth element}
```

VAX Pascal also allows array components to be arrays. These types of
arrays are called **multidimensional arrays**. The following example shows
two ways of declaring the same multidimensional array:

```
VAR
    Tic_Tac_Toe : ARRAY[1..3] OF ARRAY['a'..'c'] OF CHAR;
        {Or equivalently:}
    Tic_Tac_Toe : ARRAY[1..3, 'a'..'c'] OF CHAR; {3x3 matrix}

{In the executable section:}
Tic_Tac_Toe[ 1, 'a' ] := 'X';   {Or equivalently:}
Tic_Tac_Toe[1]['a']   := 'X';
```

For More Information:

* On character-string types (Section 2.6)

* On conformant-array parameters (Section 6.3.6.1)

* On attributes (Chapter 10)

* On the TEXT predefined data type (Section 2.7.1)

2.4.1.1 Array Constructors

Array constructors are lists of values that you can use to specify an array
value; they have the following form:

[[data-type]] [[[{ { component / component-subrange } } },... : component-value};...]]

    [[OTHERWISE component-value [[;]]]]]

data-type
Specifies the constructor's data type. If you use the constructor in the
executable section or in the CONST section, a data-type identifier is
required. Do not use a type identifier in initial-state specifiers elsewhere
in the declaration section or in nested constructors.

component
component-subrange
Specifies an element number to which the component-value applies. You
can specify a subrange of components. Array elements do not have to be
specified in order. The component must be a compile-time value or constant.

component-value
Specifies the value to be assigned to the array elements in the
component-list; the value must be of the same data type as the
array-component type. This value is a compile-time value; if you use the
constructor in the executable section, you can also use a run-time value.

OTHERWISE
Specifies a value to be assigned to all array elements that have not already
been assigned values.

When using array constructors, you must initialize all elements of the array;
you cannot partially initialize the array.

For instance, you can use either of these constructors to assign values to the
array variable:

```
VAR
    Numbers : Count VALUE [1..3,5 : 1;   4,6 : 2;   7..9 : 3;   10 : 6];
{In the executable section, constructor type is required:}
Numbers := Count[1..3,5 : 1;   4,6 : 2;   7..9 : 3;   10 : x+3];
```

These constructors give the first, second, third, and fifth component the
value 1; the fourth and sixth component the value 2; and the seventh,
eighth, and ninth components the value 3. The first constructor gives
the tenth component the value 6; the second constructor, since it is in
the executable section, can assign the run-time value x+3 to the tenth
component.

To specify values for all remaining elements, you can use the OTHERWISE
clause, as follows:

```
Numbers := Count[4,6 : 2;   7..9 : 3;   10 : x+3; OTHERWISE 1];
```

When you specify constructors for multidimensional arrays in the executable section, only specify the type of the outermost array. Consider the following example:

```
TYPE
   One_Dimension = ARRAY[1..3] OF CHAR;
   Matrix = ARRAY['a'..'b'] OF One_Dimension;
VAR
   Tic_Tac_Toe : Matrix;
{In the executable section:}
Tic_Tac_Toe := Matrix[ 1,3 : [OTHERWISE ' '];
                       2 : [ 1,3 : ' '; 2 : 'X']];
```

For More Information:

For information on nonstandard array constructors, see Section 2.4.5.1.

2.4.2 RECORD Types

A **record** is a group of components (called **fields**) that can be of various data types. Each record component may contain one or more data items, including embedded records. The record type has the following from:

[[PACKED]] RECORD [[field-list]] END

If field-list is not specified, an empty record is created. The syntax of field-list is as follows:

$$\left\{ \begin{array}{l} \text{\{\{field-identifier\},... : [[attribute-list]] type\};... [[; variant-clause]] [[;]]} \\ \text{variant-clause [[;]]} \end{array} \right\}$$

field-identifier
The name of a field.

attribute-list
One or more identifiers that provide additional information about the field.

type
The type of the corresponding field. A field can be of any type.

variant-clause
The variant part of the record.

The names of the fields must be unique within one record type, but can be repeated in different record types. You can specify the fields by specifying the record variable name (or the name of an object whose result, when used as an expression, is of an record type), followed by a period (.), and followed by the field name. If the record is unpacked and if your record components are not of a FILE type, you can use a field anywhere in a program that a variable of the field type is allowed. (This manual flags circumstances in which components of packed records cannot appear where a variable of the field type is allowed.) The only operation defined for entire records is the assignment operation (:=).

The following example shows how to assign a value to a record component:

```
TYPE
    Player_Rec = RECORD
        Wins        : INTEGER;
        Losses      : INTEGER;
        Percentage : REAL;
        END;
VAR
    Player1, Player2 : Player_Rec;
{In the executable section:}
Player1.Wins := 18;   {Assigns the value 18 to the Wins field.}
```

You can partially initialize a record using the VALUE predeclared identifier on individual fields, as follows:

```
VAR
    Player = RECORD
        Wins        : INTEGER VALUE 18; {Initial value for one field}
        Losses      : INTEGER;
        Percentage : REAL;
        END;
```

A record type can include fields that are themselves records. In this case, the name of the field includes the name of every record within which it is nested. Consider the following:

```
TYPE
    Team_Rec = RECORD
        Total_Wins       : INTEGER;
        Total_Losses     : INTEGER;
        Total_Percentage : REAL;
        Player1          : Player_Rec;   {Defined in previous example}
        Player2          : Player_Rec;
        Player3          : Player_Rec;
        END;
VAR
    Team : Team_Rec;
```

You can calculate the team's wins with the following code:

```
Team.Total_Wins := Team.Player1.Wins +
                   Team.Player2.Wins +
                   Team.Player3.Wins;
```

For More Information:

- On variant clauses (Section 2.4.2.1)
- On record constructors (Section 2.4.2.2)
- On specifying record fields using the WITH statement (Section 5.11)
- On attributes (Chapter 10)

2.4.2.1 Records with Variants

A record can include one or more fields or groups of fields called variants, which can contain different types or amounts of data at different times during program execution. When you use a record with variants, two variables of the same record type can represent different data. You can define a variant clause only for the last field in the record. The syntax for record variants is as follows:

CASE $\left\{ \begin{array}{l} \text{[[tag-identifier :]] [[attribute-list]] tag-type-identifier} \\ \text{discriminant-identifier} \end{array} \right\}$ OF

    {case-label-list : (field-list)};...
    [[[[;]] OTHERWISE (field-list)]]

tag-identifier
The name of the tag field.

attribute-list
One or more identifiers that provide additional information about the variant.

tag-type-identifier
The type identifier for the tag field.

discriminant-identifier

The name of the formal discriminant of a schema type. The value of the corresponding actual discriminant selects the active variant. Once you select the variant by discrimination, you cannot change it again. Consider the following:

```
TYPE
    Record_Template( a : INTEGER ) = RECORD
        Field_1 : REAL;
        CASE a OF
            0 : ( x : INTEGER );
            1 : ( y : REAL );
        END;
```

case-label-list

One or more case constant values of the tag field type separated by commas. A case constant is either a single constant value (for example, 1) or a range of values (for example, 5..10). You must enumerate one label for each possible value in the tag-type-identifier.

field-list

The names, types, and attributes of one or more fields. At the end of a field list, you can specify another variant clause. The field list can be empty.

The tag field consists of the elements between the reserved words CASE and OF. The tag field is common to all variants in the record type. The tag field data type corresponds to the case label values and determines the current variant.

As the syntax description shows, the tag field can be a discriminant-identifier or can be specified in one of the following ways:

• tag-identifier : [[attribute-list]] tag-type-identifier

  The tag-identifier and tag-type-identifier define the name and type of the tag field. The tag-type-identifier must denote an ordinal type. You refer to the tag field in the same way that you refer to any other field in the record (with the record.field-identifier syntax).

  The following example shows the use of the tag-identifier form:

```
TYPE
    Orders = RECORD
    Part : 1..9999;
    CASE On_Order : BOOLEAN OF
        TRUE  : ( Order_Quantity : INTEGER;
                  Price          : REAL );
        FALSE : ( Rec_Quantity   : INTEGER;
                  Cost           : REAL );
    END;
```

In this example, the last two fields in the record vary depending on whether the part is on order. Records for which the value of the tag-identifier On_Order is TRUE will contain information about the current order; those for which it is FALSE, about the previous shipment.

- [[attribute-list]] tag-type-identifier

    In the second form, there is no tag-identifier you can evaluate to determine the current variant. If you use this form, you must keep track of the current variant yourself. The tag-type-identifier must denote an ordinal type.

    The following example shows the specification of a tag field without a tag-identifier:

```
TYPE
    Characters = RECORD
    CASE CHAR OF
        'A'..'Z'   : ( Capital : INTEGER );
        '0'..'9'   : ( Number  : INTEGER );
        OTHERWISE    ( Misc    : BOOLEAN );
    END;
```

    In this example, the last field in this record will be one of the following:

    - The integer field Capital if the range 'A'..'Z' is the variant most recently referred to

    - The integer field Number if the range '0'..'9' is the variant most recently referred to

    - The Boolean field Misc if the character value falls outside the previous two variants

You can refer only to the fields in the current variant. You should not change the variant while a reference exists to any field in the current variant.

You can include an OTHERWISE clause as the last case label list. OTHERWISE is equivalent to a case label list that contains tag values (if any) not previously used in the record. The variant labeled with OTHERWISE is the current variant when the tag-identifier has a value that does not occur in any of the case label lists.

The variant can contain a nested variant, as follows:

```
VAR
    Hospital : RECORD
    Patient   : Name;
    Birthdate : Date;
    Age       : INTEGER;
    CASE Pat_Sex : Sex OF
         Male   : ();
         Female : ( CASE Births : BOOLEAN OF
                         FALSE  : ();
                         TRUE   : ( Num_Kids : INTEGER ));
    END;
```

This record includes a variant field for each woman based on whether she has children. A second variant, which contains the number of children, is defined for women who have children.

For More Information:

* On the syntax of a field list (Section 2.4.2)

* On conditions that establish a variable reference (Section 3.7)

* On attributes (Chapter 10)

2.4.2.2   Record Constructors

Record constructors are lists of values that you can use to initialize a record; they have the following form:

[[data-type]] [[[{{component},... : component-value};...]]

```
      ( CASE [[tag-identifier :]] tag-value OF        )
   [[ {     [[{component},... : component-value};...]  }]] ]
      ( OTHERWISE ZERO [[;]]                           )
```

data-type
Specifies the constructor's data type. If you use the constructor in the executable section or in the CONST section, a data-type identifier is required. Do not use a type identifier in initial-state specifiers elsewhere in the declaration section or in nested constructors.

component
Specifies a field in the fixed-part of the record. Fields in the constructor do not have to appear in the same order as they do in the type definition. (If you choose, you can specify fields from the variant-part as long as the fields do not overlap.)

component-value
Specifies a value of the same data type as the component. The value is a
compile-time value; if you use the constructor in the executable section, you
can also use run-time values.

CASE
Provides a constructor for the variant portion of a record. If the record
contains a variant, its constructor must be the last component in the
constructor list.

tag-identifier
Specifies the tag-identifier of the variant portion of the record. This is only
required if the variant part contained a tag-identifier.

tag-value
Determines which component list is applicable according to the variant
portion of the record.

OTHERWISE ZERO
Sets all remaining components to their binary zero value. If you use
OTHERWISE ZERO, it must be the the last component in the constructor.

You can use either of the following constructors to assign values to the
record variable:

```
VAR
    Player1 : Player_Rec VALUE [Wins: 18; Losses: 3;
                                    Percentage: 21/18];
{In executable section, constructor type is required
 and run-time expressions are legal:}
Player1 := Player_Rec[Wins: 18; Losses: y; Percentage: y+18/18];
```

When you specify constructors for records that nest records, specify the type
of the outermost record, but do not specify the type of the constructors for
any nested records. Consider the following example:

```
TYPE
    Team_Rec = RECORD
        Total_Wins       : INTEGER;
        Total_Losses     : INTEGER;
        Total_Percentage : REAL;
        Player1          : Player_Rec;
        Player2          : Player_Rec;
        Player3          : Player_Rec;
        END;
```

```
VAR
   Team : Team_Rec;
{In the executable section: }
Team :=
 Team_Rec[Total_Wins: 18; Total_Losses: 3; Total_Percentage: 21/18;
        Player1: [Wins: 6; Losses: 0; Percentage: 1.0 ];
        Player2: [Wins: 5; Losses: 2; Percentage: 7/5 ];
        Player3: [Wins: 7; Losses: 1; Percentage: 8/7 ]];
```

You can call the ZERO function within record constructors to initialize all
nonspecified components to their binary zero values, which are determined
by the data type of each component. Consider the following examples:

```
VAR
   Team : Team_Rec VALUE ZERO;
   Team : Team_Rec VALUE
        [Total_Wins: 5; Total_Losses: 2; Total_Percentage: 7/5;
         Player2: [Wins: 5; Losses: 2; Percentage: 7/5 ];
         OTHERWISE ZERO]; {Initializes Player1 and Player3}
```

To create a constructor for a record that contains a variant, you use the
reserved word CASE, followed by one of the following:

- A tag-identifier and a colon (:), followed by a constant expression (if you
  use both a tag-identifier and a tag-type-identifier in the declaration)

- A constant expression (if you use only a discriminant-identifier or a
  tag-type-identifier in the declaration)

To complete the constructor, use the reserved word OF, followed by
component-list values contained in a nested constructor. Consider the
following valid constructors:

```
TYPE
   Orders = RECORD
   Part : 1..9999;
   CASE On_Order : BOOLEAN OF
      TRUE  : ( Order_Quantity : INTEGER;
                Price          : REAL );
      FALSE : ( Rec_Quantity   : INTEGER;
                Cost           : REAL );
   END;
VAR
   An_Order : Orders VALUE
             [Part: 2358;
              CASE On_Order : FALSE OF
              [Rec_Quantity: 10; Cost: 293.99]];

{In the executable section, constructor type is required:}
An_Order := Orders
             [Part: 2358;
              CASE On_Order : FALSE OF [Rec_Quantity: 10; Cost: 293.99]];
```

Note that if you use a constructor in the type definition, you can specify an initial state for only one variant in the type. To specify an initial state for more than one variant, you must put initial state specifiers on the fields themselves. For example:

```
TYPE
   Orders = RECORD
   Part : 1..9999 VALUE 25;
   CASE On_Order : BOOLEAN OF
      TRUE  : ( Order_Quantity : INTEGER VALUE 18;
                Price          : REAL VALUE 4.65 );
      FALSE : ( Rec_Quantity   : INTEGER VALUE 10;
                Cost           : REAL VALUE 46.50 );
   END;
```

For More Information:

*   On the ZERO function (Section 8.89)

*   On nonstandard record constructors (Section 2.4.5.2)

2.4.3   SET Type

A **set** is a collection of data items of the same ordinal type (called the **base type**). The SET type definition specifies the values that can be elements of a variable of that type. The SET type has the following form:

[[PACKED]] SET OF [[attribute-list]] base-type

attribute-list
One or more identifiers that provide additional information about the base type.

base-type
The ordinal type identifier or type definition, or discriminated schema type, from which the set elements are selected. Real numbers cannot be elements of a set type.

You define a set by listing all the values that can be its elements. A set whose base type is INTEGER or UNSIGNED has two restrictions: the set can contain no more than 256 elements, and the ordinal value of these elements must be within the range of 0 and 255. For sets of other ordinal base types, elements can include the full range of the type.

For More Information:

- On the INTEGER type (Section 2.1.1)
- On the UNSIGNED type (Section 2.1.2)
- On the subrange types (Section 2.1.6)
- On attributes (Chapter 10)
- On schema discriminants in sets (Section 2.5)

2.4.3.1 Set Constructors

Set constructors are lists of values that you can use to initialize a set; they have the following form:

[[data-type]] [[[{component-value},...]]]

data-type
The data type of the constructor. This identifier is optional when used in the CONST and executable sections; do not use this identifier in the TYPE and VAR sections or in nested constructors.

component-value
Specifies values within the range of the defined data type. Component values can be subranges (..) to indicate consecutive values that appear in the set definition. These values are compile-time values; if you use the constructor in the executable section, you can also use run-time values.

A set having no elements is called an empty set and is written as empty brackets ([]).

A possible constructor for a variable of type SET OF 35..115 is the following:

```
VAR
    Numbers : SET OF 35..115 VALUE [39, 67, 110..115];
{In the executable section, run-time expressions are legal:}
Numbers := [39, 67, x+95, 110..115];
```

The set constructors contain up to nine values: 39, 67, x+95 (in the executable section only), and all the integers between 110 and 115, inclusive. If the expression x+95 evaluates to an integer outside of the range 35..115, then VAX Pascal includes no set element for that expression.

To initialize a set to the empty set, do the following:

```
VAR
    Day : SET OF 1..31 VALUE [];
```

2.4.4 FILE Type

A **file** is a sequence of components of the same type. The number of
components is not fixed; a file can be of any length. The FILE type definition
identifies the component type and has the following form:

[[PACKED]] FILE OF [[attribute-list]] component-type

attribute-list
One or more identifiers that provide additional information about the file
components.

component-type
The type of the file components. This type can be any ordinal, real, pointer,
or structured type except for the following:

- A nonstatic type

- A structured type with a nonstatic component

- A file type

- A structured type with a file component

The arithmetic, relational, Boolean, and assignment operators cannot be
used with file variables or structures containing file components. You cannot
form constructors of file types.

Consider the following:

```
VAR
    True_False_File : FILE OF BOOLEAN;
                         {File of TRUE and FALSE values}
    Experiment_Records : FILE OF RECORD {File of records}
        Trial : INTEGER; {To access, Experiment_Records^.Trial}
        Date  : RECORD
            Month : ( Jan, Feb, Mar, Apr, May, Jun,
                      Jul, Aug, Sep, Oct, Nov, Dec );
            Day   : 1..31;
            Year  : INTEGER;
            END;
        Temp, Pressure  : INTEGER;
        Yield, Purity   : REAL;
        END;
```

For More Information:

- On file organizations (Section 9.1)
- On component formats (Section 9.2)
- On conditions that establish a variable reference (Section 3.7)
- On attributes (Chapter 10)

2.4.5 Nonstandard Constructors

As an option, you can use another format for constructors that is provided as a VAX Pascal extension. VAX Pascal retains this format only for compatibility with programs written for use with previous versions of this product. Also, you cannot use nonstandard constructors for variables of nonstatic types.

For all nonstandard constructors, you place constant values, of the same type as the corresponding component, in a comma list within parentheses. The compiler matches the values with the components using positional syntax; you must provide a value for each component in the variable. Nested structured components are designated by another comma list inside of another set of parentheses. Nonstandard constructors are legal in the VAR and VALUE initialization sections, and in the executable section. Specifying a type identifier as part of a constructor is optional for constructors used in the VAR and VALUE initialization sections, are required for constructors in the executable section, and cannot be used for nested constructors.

For More Information:

- On Pascal standards (Section 1.1)
- On standard constructors (Section 2.4)

2.4.5.1 Nonstandard Array Constructors

The format for nonstandard array constructors is as follows:

[[data-type]] ([[{component-value},...]] [[REPEAT component-value]])

data-type
Specifies the constructor's data type. If you use the constructor in the executable section, a data-type identifier is required. Do not use a type identifier in the VAR or VALUE sections, or for a nested constructor.

component-value

Specifies the compile-time value to be assigned to the corresponding array element. The compiler assigns the first value to the first element, the second value to the second element, and so forth. If you want to assign more than one value to more than one consecutive element, you can use the following syntax for a component-value:

n OF value

For instance, the following component value assigns the value of 15 to the first three components of an array:

```
VAR
    Array1 : ARRAY[1..4] OF INTEGER;
VALUE
    Array1 := ( 3 OF 15, 78 );
```

You cannot use the OF reserved word in a REPEAT clause.

REPEAT

Specifies a value to be assigned to all array elements that have not already been assigned values.

An example of an array constructor is as follows:

```
TYPE
    Count = ARRAY[1..10] OF INTEGER;
VAR
    Numbers : Count;
VALUE
    Count := ( 3 OF 1, 2, 1, 2, 3 OF 3, 3 );
{In the executable section, constructor type is required:}
Numbers := Count( 3 OF 1, 2, 1, 2, REPEAT 3 );
```

An example of a constructor for a multidimensional array is as follows:

```
TYPE
    One_Dimension = ARRAY[1..3] OF CHAR;
    Matrix = ARRAY['a'..'b'] OF One_Dimension;
VAR
    Tic_Tac_Toe : Matrix;
    { In the executable section: }
Tic_Tac_Toe := Matrix( (3 OF ' '), (' ', 'X', ' '), (3 OF ' ') );
```

For More Information:

For information on standard array constructors, see Section 2.4.1.1.

2.4.5.2 Nonstandard Record Constructors

The format for a nonstandard record constructor is as follows:

[[data-type]] ([[{component-value},...]] [[tag-value, {component-value};...]])

data-type
Specifies the constructor's data type. If you use the constructor in the executable section, a data-type identifier is required. Do not use a type identifier in the VAR or VALUE sections, or for a nested constructor.

component-value
Specifies a compile-time value of the same data type as the component. The compiler assigns the first value to the first record component, the second value to the second record component, and so forth.

tag-value
Specifies a value for the tag-identifier of a variant record component. The value that you specify as this component of the constructor determines the types and positions of the remaining component values (according to the variant portion of the type definition).

An example of a record constructor is as follows:

```
TYPE
    Player_Rec = RECORD
        Wins        : INTEGER;
        Losses      : INTEGER;
        Percentage  : REAL;
VAR
    Player1 : Player_Rec := ( 18, 6, 24/18 );
{In the executable section, constructor type is required:}
Player1 := Player_Rec( 18,  6, 24/18 );
```

The following is an example of a nested record constructor:

```
TYPE
    Team_Rec = RECORD
        Total_Wins       : INTEGER;
        Total_Losses     : INTEGER;
        Total_Percentage : REAL;
        Player1          : Player_Rec;
        Player2          : Player_Rec;
        Player3          : Player_Rec;
        END;
VAR
    Team : Team_Rec;
{In the executable section: }
Team :=  Team_Rec ( 18, 3, 18/21,
                  ( 6, 0, 1.0 ),
                  ( 5, 2, 5/7 ),
                  ( 7, 1, 7/8 ) );
```

The following is an example of a variant record constructor:

```
TYPE
    Orders = RECORD
    Part : 1..9999;
    CASE On_Order : BOOLEAN OF
                  TRUE  : ( Order_Quantity : INTEGER;
                            Price          : REAL );
                  FALSE : ( Rec_Quantity   : INTEGER;
                            Cost           : REAL );
    END;
VAR
    An_order : Orders := ( 2358, FALSE, 10, 293.99 );
```

For More Information:

*   On standard record constructors (Section 2.4.2.2)
*   On record variants (Section 2.4.2.1)

2.5 Schema Types

A **schema type** is a user-defined construct that provides a template for
a family of distinct data types. A schema type definition contains one or
more **formal discriminants** that take the place of specific boundary values
or variant-record selectors. By specifying boundary or selector values to a
schema type, you form a valid data type; the provided boundary or selector
values are called **actual discriminants**. Schema types have the following
form:

schema-identifier ({{discriminant-identifier},.... : [[attribute-list]] ordinal-type-name};...)
    = [[attribute-list]] type-denoter;

schema-identifier
The name of the schema.

discriminant-identifier
The name of a formal discriminant.

ordinal-type-name
The type of the formal discriminant, which must be an ordinal type.

attribute-list
One or more identifiers that provide additional information about the
type-denoter.

type-denoter
The type definition of the components of the schema. This must define a
new record, array, set, or subrange type.

Each schema type definition requires at least one discriminant identifier.
A discriminant identifier does not have to be used in the type-denoter
definition, but VAX Pascal still uses the discriminant identifier to determine
type compatibility. Discriminant identifiers can appear anywhere a value is
required in this definition.

Consider the following example:

```
TYPE
    Array_Template( Upper_Bound : INTEGER )
                  = ARRAY[1..Upper_Bound] OF INTEGER;
```

The identifier Upper_Bound is the formal discriminant of the
Array_Template schema. The Array_Template schema is not a complete
description of data. It is not a valid data type until you provide an actual
discriminant that designates the upper boundary of the array template.
Schema types that have not been provided actual discriminants are called
undiscriminated schema; in the previous example, Array_Template is an
undiscriminated schema. You can use an undiscriminated schema in the
following instances:

• As the domain type of a pointer

• As the type of a formal parameter

In a undiscriminated schema declaration, you can use a combination of
formal discriminants, compile-time values, and nested descriminants to
form subrange bounds. These types of expressions are called **nonvarying
expressions**. Consider the following:

```
TYPE
    Vector( d : INTEGER ) = ARRAY[0..d-1] OF BOOLEAN;
    Number_Line( Starting, Distance : INTEGER ) =
        Starting..Starting+Distance;
    My_Subrange( l,u : INTEGER ) = l..u;
    Shift_Array_Index( l2, u2, Length : INTEGER ) =
        ARRAY[My_Subrange( l2+10, u2+10 )] OF STRING( Length );
```

The following example provides the Array_Template schema with actual
discriminants to form complete data types (remaining examples in this
section use the Array_Template declaration).

```
TYPE
   Array_Template( Upper_Bound : INTEGER )
                  = ARRAY[1..Upper_Bound] OF INTEGER;
VAR
   Array1 : Array_Template( 10 );   {ARRAY[1..10] OF INTEGER;}
   Array2 : Array_Template( x );    {Upper boundary determined at
                                     run-time by variable or
                                     function call}
```

In the previous example, the actual discriminants 10 and x complete the boundaries for Array_Template, forming two complete data types within the same schema type family. A schema type that has been provided actual discriminants is called a **discriminated schema**; discriminated schema can appear in either the TYPE or VAR sections. The type specifiers Array_Template(10) and Array_Template(x) are examples of discriminated schema.

Actual discriminants can be compile- or run-time expressions. This expression must be assignment compatible with the ordinal type specified for the formal discriminant. Also, the actual discriminant value must be inside the range specified for the formal discriminant; in the case of subranges, the upper value must be greater than or equal to the lower value. In the previous example, 10 and x must be within the range –MAXINT..MAXINT.

If you want to use a discriminated schema type as the domain type of a pointer or as the type of a formal parameter, give the discriminated schema type a name by declaring it in the TYPE section. Consider the following:

```
TYPE
   Array_Type1 = Array_Template( 10 );

PROCEDURE Example( Param : Array_Type1 );   {Procedure body...}
```

For any undiscriminated schema, there is a range of possible data types that you can form by discrimination. A **schema family** is the undiscriminated schema type and the range of data types that can be formed from it. Also, two separate discriminations that provide the same actual discriminant value specify the same data type. Consider the following:

```
VAR
   Array1 : Array_Template( 10 );
   Array2 : Array_Template( 10 );
   Array3 : Array_Template( 15 );
```

Array_Template, Array1, Array2, and Array3 are all of the same schema family. Array1 and Array2 are of the same data type.

Once you create a discriminated schema, you can access the value of an actual discriminant. Consider the following example:

```
VAR
    Array1 : Array_Template( 10 );
{In the executable section:}
WRITELN( Array1.Upper_Bound );    {Writes 10 to the default device}
```

Discriminant values can appear in all expressions except constant expressions. The following example shows a valid use of the discriminant-value expression:

```
FOR i := 1 TO Array1.Upper_Bound DO
    Array1[i] := i;
```

You can use discriminated schema in the type-denoter of a schema definition. You can also discriminate a schema in the type-denoter of a schema definition, but the actual discriminants must be expressions whose values are nonvarying; the actual discriminants cannot be variables or function calls.

Consider the following valid schema definitions:

```
TYPE
    {   Legal schema types:    }
    Range1( a, b : INTEGER ) = SET OF a..b+1;   {Run-time bounds checking}

    My_Record( Number_Size, Status_Size : INTEGER ) = RECORD
        Part_Number : PACKED ARRAY[1..Number_Size] OF INTEGER;
        Status      : STRING( Status_Size );    {Nested schema}
        END;

    Range2( Low, Span : INTEGER ) = Low..Low + Span;
    My_Integer( Dummy : INTEGER ) = -MAXINT-1..MAXINT;
    Matrix( Bound : INTEGER ) = ARRAY[1..Bound, 1..Bound] OF REAL;

    {   Illegal schema types (they do not form "new" types):   }
    My_String( Len : INTEGER ) = VARYING[Len] OF CHAR;
    My_Integer( Dummy : INTEGER ) = INTEGER;
```

For More Information:

- On ordinal types (Section 2.1)

- On compile-time and run-time expressions (Section 4.1)

- On attributes (Chapter 10)

- On predeclared routines (Chapter 8)

- On using schema types (*VAX Pascal User Manual*)

2.6 String Types

You can use schema and data types to store and to manipulate character strings. These types have the following order of complexity:

1. CHAR type
2. PACKED ARRAY OF CHAR user-defined types
3. VARYING OF CHAR user-defined types
4. STRING predefined schema

Objects of the CHAR data type are character strings with a length of 1 and are lowest in the order of character string complexity. You can assign CHAR data to variables of the other string types.

The PACKED ARRAY OF CHAR types allow you to specify fixed-length character strings. The VARYING OF CHAR types are a VAX Pascal extension that allows you to specify varying-length character strings with a constant maximum length. The STRING types provide a standard way for you to specify storage for varying-length character strings with a maximum length that can be specified at run time.

To provide values for variables of these types, you should use a character-string constant (or an expression that evaluates to a character string) instead of an array constructor. Using array constructors with STRING and VARYING OF CHAR types generates an error; to use array constructors with PACKED ARRAY OF CHAR types, you must specify component values for every element in the array (otherwise, you generate an error).

Consider the following example:

```
VAR
    String1 : VARYING[10] OF CHAR VALUE 'abc';
```

Generally, you can use any member of the ASCII character set in character-string constants and expressions. However, some members of the ASCII character set, including the bell, the backspace, and the carriage return, are nonprinting characters. The **extended string** format for printing character strings with nonprinting characters is as follows:

```
{'printing-string'({ordinal-value},...)}...
```

printing-string

A character string constant.

ordinal-value

An integer denoting the ordinal value of an ASCII character.

Consider the following example:

```
'Two bells'(7,7)' in a null-terminated ASCII string.'(0)
```

For More Information:

* On the CHAR data type (Section 2.1.3)
* On the ASCII chart (Appendix A)

2.6.1 PACKED ARRAY OF CHAR Types

User-defined packed arrays of characters with specific lower and upper bounds provide a method of specifying fixed-length character strings. The string's lower bound must equal 1. The upper bound establishes the fixed length of the string.

The following example shows a declaration of a character string variable of twenty characters:

```
VAR
    My_String : PACKED ARRAY[1..20] OF CHAR;
```

NOTE

If the upper bound of the array exceeds 65,535, if the PACKED reserved word is not used, or if the array's components are not byte-sized characters, the compiler does not treat the array as a character string.

To assign values to fixed-length character strings, you can use a character-string constant (or an expression that evaluates to a character string). When assigning into fixed-length strings, the compiler adds blanks to extend a string shorter than the maximum characters declared. If you specify a string longer than the maximum characters declared, an error occurs. You can also use an array constructor as long as you specify characters for every component of the array as specified in the declaration. Consider the following.

```
VAR
    States : PACKED ARRAY[1..20] OF CHAR
             VALUE 'Hello';                              {Is legal}
    States : PACKED ARRAY[1..20] OF CHAR
             VALUE [1:'H';2:'e';3:'l';4:'l';5:'o']   {Generates
                                                       an error}
    States : PACKED ARRAY[1..20] OF CHAR
             VALUE [1:'H';2:'e';3:'l';4:'l';5:'o';
             OTHERWISE ' ']                          {Is legal,
                                                      but awkward}
```

For More Information:

For information on arrays, see Section 2.4.1.

2.6.2 VARYING OF CHAR Types

The VARYING OF CHAR user-defined types are a VAX Pascal extension
that provides a way of declaring variable-length character strings with
a compile-time maximum length. If you require portable code, use the
STRING predefined schema types to specify variable-length character
strings. VARYING OF CHAR types have the following form:

VARYING[upper-bound] OF [[attribute-list]] CHAR

upper-bound
An integer in the range from 1 through 65,535 that indicates the length of
the longest possible string.

attribute-list
One or more identifiers that provide additional information about the
VARYING OF CHAR string component.

You can assign string constants to VARYING OF CHAR variables from
length 0 to the specified upper-bound. The compiler allocates enough
storage space to hold a string of the maximum length. A VARYING OF
CHAR variable with length 0 is the empty string (' '). You can only use
character-string constants (or expressions that evaluate to character strings)
to assign values to variables of these types; you cannot use standard array
constructors. Also, you can initialize a character string to the empty string
(' '), as follows:

```
VAR
    String1 : VARYING[10] OF CHAR VALUE '';
```

The VARYING OF CHAR variable is stored as though it were a record with two fields, as follows:

```
RECORD
    LENGTH   : [WORD] 0..upper-bound;  {Length of current string}
    BODY     : PACKED ARRAY[1..upper-bound] OF CHAR;  {Current string}
    END;
```

You can access the LENGTH and BODY predeclared identifiers as you would access fields of a record. For instance, to determine the maximum length of a VARYING OF CHAR variable, you can use the SIZE predeclared function and the BODY predeclared identifier, as follows:

```
VAR
    String1 : VARYING[10] OF CHAR VALUE 'Wolf';
{In the executable section: }
Max_Length := SIZE( string1.BODY );
WRITELN( Max_Length );                    {writes '10'}
```

To determine the current length of a VARYING OF CHAR variable, you can use the LENGTH predeclared function. From the previous example, the result of LENGTH(String1) is the same as String1.LENGTH.

You can refer to individual array components as you would individual components of any array, as follows:

```
String1[8] := 'L';
```

You cannot specify an index value that is greater than the length of the current string. VAX Pascal does not pad remaining characters in the current string with blanks (' '). If you specify an index that is greater than the current length of the string, an error occurs.

For More Information:

- On arrays (Section 2.4.1)
- On attributes (Chapter 10)
- On the SIZE predeclared function (Section 8.66)
- On the LENGTH predeclared function (Section 8.41)

2.6.3 STRING Schema Type

The STRING predefined schema provides a way of declaring variable-length character strings. The compiler stores STRING data as though it were stored in the following schema definition:

```
TYPE
    STRING( Capacity : INTEGER ) = VARYING[Capacity] OF CHAR;
```

The syntax of the discriminated schema is as follows:

```
STRING( CAPACITY )
```

CAPACITY

An integer in the range 1..65,535 that indicates the length of the longest possible string.

To use the predefined STRING schema, you provide an upper bound as the actual discriminant. Consider the following example:

```
VAR
    Short_String : STRING( 5 );      {Maximum length of 5 characters}
    Long_String  : STRING( 100 );   {Maximum length of 100 characters}
```

You can assign string constants to STRING variables from length 0 to the specified upper bound. The compiler allocates enough storage space to hold a string of the maximum length. A STRING variable with length 0 is the empty string (′′). To provide values for variables of this type, you must use character-string constants (or expressions that evaluate to character strings); you cannot use array constructors. Also, you can initialize a character string to the empty string (′′), as follows:

```
VAR
    Short_String : STRING( 5 ) VALUE ′′;
```

You can access the CAPACITY predeclared identifier as you would a schema discriminant, and you can access the LENGTH and BODY predeclared identifiers as you would access fields of a record. The CAPACITY identifier allows you to access the actual discriminant of the STRING schema; the LENGTH identifier allows you to access the current length of the string object; and, the BODY identifier contains the current string object, including whatever is in memory up to the capacity of the discriminated schema. Consider the following example:

```
VAR
    String1 : STRING( 10 ) VALUE 'Wolf';
{In the executable section: }
WRITELN( String1.CAPACITY );   {prints '10'}
WRITELN( String1.LENGTH );     {prints '4'}
```

The value String1.BODY contains the four-character string 'Wolf' followed by whatever is currently stored in memory for the remaining 6 characters.

To determine the current length of a STRING variable, you can use the LENGTH predeclared function. The result of LENGTH(String1) is the same as String1.LENGTH.

You can refer to individual STRING components as you would individual components of any array, as follows:

```
String1[5] := 't';
```

The compiler does not pad remaining characters in the current string with blanks (' '). If you specify an index that is greater than the current length of the string an error occurs. Consider the following example:

```
VAR
    String1 : STRING( 10 ) VALUE 'Wombat';
    x        : CHAR;
{In the executable section:}
x := String1[9];        {Generates an error}
x := String1.BODY[9];   {Provides whatever is in memory there}
x := String1[5];        {Is legal}
String1[9] := 'X';      {Generates an error}
```

For More Information:

*   On schema types (Section 2.5)

*   On arrays (Section 2.4.1)

*   On the SIZE predeclared function (Section 8.66)

2.7 Predefined Structured and Schema Types

The following sections discuss additional structured and schema types that are predefined by VAX Pascal for your use.

2.7.1 TEXT Type

The TEXT predefined type is a file containing sequences of characters with special markers (end-of-line and end-of-file) added to the file. Although each character of a TEXT file is one file component, the end-of-line marker allows you to process the file line-by-line, if you choose. The TEXT type has the following form:

[[attribute-list]] TEXT

attribute-list
One or more identifiers that provide additional information about the file components.

For More Information:

- On the FILE type (Section 2.4.4)
- On TEXT files (Section 9.5)
- On INPUT and OUTPUT identifiers (*VAX Pascal Reference Supplement for VMS Systems*)

2.7.2 TIMESTAMP Type

The TIMESTAMP predefined type is used in conjunction with the GETTIMESTAMP procedure or with the DATE or TIME functions. GETTIMESTAMP initializes a variable of type TIMESTAMP; DATE and TIME function results are of type TIMESTAMP.

For More Information:

- On the DATE and TIME functions (Section 8.18)
- On the layout of the GETTIMESTAMP type (Section 8.34)

2.8 Static and Nonstatic Types

Static types are types whose objects can be fully described at compile time. For instance, the variables a and b are derived from static types in the following example:

```
VAR
   a : INTEGER;
   b : ARRAY[1..10] OF INTEGER;
```

Nonstatic types are types whose objects potentially cannot be fully described at compilation time (the type has a component that can be a run-time value). Nonstatic types include the following types:

- Discriminated and undisriminated schema types
- Any type that contains a nonstatic component or index type

Nonstatic types require storage allocation to hold information about the type at run time. This storage, called the **control part**, includes information that cannot be determined until execution time; VAX Pascal needs this information to allocate and to access variables and record fields of this type.

Consider the following nonstatic types:

```
TYPE
                            {Template is nonstatic:}
    Template( Upper : INTEGER ) = ARRAY[1..Upper] OF INTEGER;
    a = ^Template;                  {a's base type is nonstatic}
    b = Template( 5 );              {b is nonstatic}
    My_Subrange( x, y : INTEGER ) = x..y;
                                    {c is nonstatic:}
    c = ARRAY[My_Subrange( j, k ), My_Subrange( l, m )] OF INTEGER;
    d = ARRAY[1..10] OF Template( 5 ); {d is nonstatic}
    e = RECORD                      {e is nonstatic}
       f1 : TEMPLATE( 5 );
       END;
    f = SET OF My_Subrange( 10, 20 );  {f is nonstatic}
```

Do not confuse static and nonstatic types with automatic and static variable allocation.

For More Information:

* On automatic variable allocation (Section 10.2.4)

* On static variable allocation (Section 10.2.35)

* On storage representation of nonstatic types (the *VAX Pascal Reference Supplement for VMS Systems*)

2.9 Type Compatibility

The following sections discuss the two forms of type compatibility: structural and assignment compatibility.

2.9.1 Structural Compatibility

Two types are structurally compatible only if they have the same allocation size and the same type structure. VAX Pascal requires that the type of a variable passed to a routine as an actual parameter be structurally compatible with the type of the corresponding formal variable parameter. VAX Pascal also checks the structural compatibility of the base types when a pointer expression is assigned to a pointer variable. Structural compatibility does not apply to nonstatic types.

Two ordinal types are structurally compatible only if they have the same base type and the same allocation size.

If two ordinal types are components of packed structured types, they are structurally compatible only if the ranges of values they describe have identical upper and lower bounds.

In general, each real type is structurally compatible only with itself. However, because REAL and SINGLE are synonymous, they are structurally compatible with each other.

For two structured types to be structurally compatible, they must have the same allocation size, and both must be packed or both unpacked. The following conditions also affect structural compatibility:

- If both types are record types, they must have the same number of fields, and the types of corresponding fields must be structurally compatible and identically positioned. If the record types have variant parts, the corresponding variants must have identical case labels written in the same order. The types of the fields within corresponding variants must be structurally compatible.

- If both types are array types, the types of their components must be structurally compatible. The index types must have identical base types and identical upper and lower bounds.

- If both types are VARYING OF CHAR types, their maximum lengths must be equal. The lengths of the current values of the VARYING OF CHAR strings do not affect structural compatibility.

- If two components of packed structured types are set types, their base types must have identical upper and lower bounds.

- If both types are set types, file types, or pointer types, their base types must be structurally compatible. Because of the possibility that a pointer type can be defined in terms of itself, the VAX Pascal compiler begins the test for the structural compatibility of two pointer types by assuming that they are compatible. Next, the compiler tests the two base types for structural compatibility. If within the base type, the compiler encounters the same pointer types it is testing, it still follows the original assumption that the pointer types are compatible. If the base types prove to be structurally compatible, then the two pointer types are judged to be structurally compatible.

For More Information:

- On attributes that affect size and structure: ALIGNED, POS, READONLY, UNALIGNED, UNSAFE, VOLATILE, and WRITEONLY (Chapter 10)
- On ordinal types (Section 2.1)
- On real types (Section 2.2)

- On pointer types (Section 2.3)
- On structured types (Section 2.4)
- On default sizes (*VAX Pascal Reference Supplement for VMS Systems*)

2.9.2 Assignment Compatibility

Assignment compatibility rules apply to the types of values used to initialize variables, the types of expressions assigned to variables with the assignment operator (:=), and the types of actual parameters passed to formal value parameters.

Table 2–3 shows the contexts in which the type of an expression is assignment compatible with the type of a variable or a formal parameter.

Table 2–3: Assignment Compatibility

Type of Variable	Type of Assignment-Compatible Expression
INTEGER	INTEGER
UNSIGNED	UNSIGNED INTEGER
CHAR	CHAR
Subrange	Base type of the subrange
REAL, SINGLE	REAL, SINGLE, UNSIGNED, INTEGER
DOUBLE	DOUBLE, REAL, SINGLE, UNSIGNED, INTEGER
QUADRUPLE	QUADRUPLE, DOUBLE, REAL, SINGLE, UNSIGNED, INTEGER
PACKED ARRAY OF CHAR	CHAR, PACKED ARRAY OF CHAR with the same or smaller length, VARYING or STRING string whose current length is equal to or less than the packed array
VARYING OF CHAR	CHAR, PACKED ARRAY OF CHAR, VARYING, STRING, string whose current value does not exceed the maximum length of the variable or parameter
STRING	CHAR, PACKED ARRAY OF CHAR, VARYING, STRING, string whose current value does not exceed the maximum length of the variable or parameter
Pointer	Pointer to a structurally compatible type

Two record types or two array types are assignment compatible if they are structurally compatible. When you assign one record variable to another, or one array variable to another, the VAX Pascal compiler does not check for out-of-range assignments to record fields or array components; such assignments do not result in an error message, even if subrange checking is enabled at compile time.

A set expression is assignment compatible with a set variable if the set's base types are compatible. In addition, all elements of the set expression must be included in the range of the variable's base type.

Note that assignment operations are not allowed on objects of file types or structured types that have file components.

Two discriminated schema types are assignment compatible if they are of the same type family and if their actual discriminant values are identical. A dereferenced pointer to an undiscriminated schema type is actually referencing a discriminated schema object whose discriminants were specified in a call to the NEW function. Although STRING is a schema, the rules in Table 2–3 take precedence.

For More Information:

- On attributes that affect assignment compatibility: POS, READONLY, and UNSAFE (Chapter 10)
- On ordinal types (Section 2.1)
- On real types (Section 2.2)
- On pointer types (Section 2.3)
- On structured types (Section 2.4)
- On schema types (Section 2.5)
- On string types (Section 2.6)

Chapter 3

The Declaration Section

The declaration section contains declarations or definitions of constants, labels, user-defined data types, variables, and user-defined functions and procedures. In addition, only modules can contain initialization and finalization sections. Each appears in a subsection introduced by VAX Pascal reserved words.

This chapter discusses the following topics:

- The CONST section (Section 3.1)
- The LABEL section (Section 3.2)
- The TO BEGIN DO section (Section 3.3)
- The TO END DO section (Section 3.4)
- The TYPE section (Section 3.5)
- The VALUE section (Section 3.6)
- The VAR section (Section 3.7)

These sections appear after the header and before the executable section (if any). The TO BEGIN DO and TO END DO sections may appear only in modules and can appear only once within a module.

The remaining sections can appear in programs, modules, functions, or procedures; they can appear more than once and in any order in a single declaration section. If you use one kind of section more than once in a declaration section, be sure to declare types, variables, and constants before you use them in subsequent sections.

For More Information:

- On user-defined procedures and functions (Chapter 6)
- On program structure (Chapter 7)
- On modules (Section 7.3)

3.1 The CONST Section

The CONST section defines symbolic constants by associating identifiers with compile-time expressions; it has the following form:

```
CONST
    {constant-identifier = constant-expression};...
```

constant-identifier
The identifier of the symbolic constant being defined.

constant-expression
Any legal compile-time expression.

Once a constant identifier is associated with an expression, the identifier retains the value of that expression throughout the scope in which it was declared. You can change the value only by changing the definition in the CONST section.

Consider the following example:

```
TYPE
    array_type1 = ARRAY[1..10] OF INTEGER;
CONST
    Year        = 1984;
    Tiny        = 1.7253;
    Month       = 'November';
    Initial     = 'P';
    Lie         = FALSE;
    Untruth     = Lie;
    Almost_Pi   = 22.0/7.0;
    array_const =
        array_type1[1..3,5 : 1;   4,6 : 2;   7..9 : 3;   10 : 7];
```

For More Information:

- On expressions (Section 4.1)
- On constructors (Section 2.4)

3.2 The LABEL Section

A **label** is a tag that makes an executable statement accessible to a GOTO statement. The LABEL section declares labels and has the following form:

```
LABEL
    {label},...;
```

label

A decimal integer between 0 and 9999 (as an extension, between 0 and MAXINT), or a symbolic name. When declaring several labels, you can specify them in any order. The declaration and the occurrence of the label must be at the same level in the program.

A label can appear only once within the scope of the label declaration. It can precede any executable statement in the program. Use a colon (:) to separate the label from the statement it precedes. Labels can be accessed only by GOTO statements.

Consider the following example:

```
LABEL
    marker, 5;
{In the executable section: }
IF a <= 150 THEN GOTO 5
ELSE GOTO marker;
    .
    .
    .
5: a := a + 1;
    .
    .
    .
marker: WHILE x < 20 DO {Statement...}
```

For More Information:

For information on the GOTO statement, see Section 5.6.

3.3 The TO BEGIN DO Section

The TO BEGIN DO section allows you to specify a statement, in a module, that is to be executed before the executable section of the main program; it has the following form:

```
TO BEGIN DO statement;
```

statement
A VAX Pascal statement.

The TO BEGIN DO section can only appear in modules, can only appear once in a module, and must appear as the last section in the declaration section. (If appearing together, the TO BEGIN DO section must precede the TO END DO section at the end of the declaration section.)

Consider the following example:

```
MODULE x( INPUT, OUTPUT );
VAR
    Debug : BOOLEAN;
PROCEDURE Test(...); {Executable section...}

TO BEGIN DO
    BEGIN
    WRITE('Debug Module x?    ');
    READLN( Debug );
    END;
END.
```

As a general rule, if a program or module inherits an environment file, the initialization section in the inherited module must be executed before the initialization section in the program or module that inherited it. If a module or program inherits more than one module that contains an initialization section, the order of execution of the inherited modules cannot be determined.

Consider the following example:

```
[ENVIRONMENT( 'Mod1' )]  MODULE Mod1;
VAR
    i : INTEGER;
TO BEGIN DO
    i := 5;

{In a separate compilation unit:}
[INHERIT( 'Mod1' )]  MODULE Mod2;
VAR
    j : INTEGER;
TO BEGIN DO
    j := i + 1;   {First execute code in Mod1 for correct results}
```

Figure 3-1 illustrates the order of execution of initialization and finalization sections. Each circle is a module that contains both a TO BEGIN DO and a TO END DO section and each arrow indicates the order of inheritance for the environment files.

Figure 3–1:   Order of Execution for TO BEGIN DO and TO END DO Sections

Order of Execution:

TO BEGIN DO	TO END DO
First	Third
Second	Second
Third	First

ZK–1321A–GE
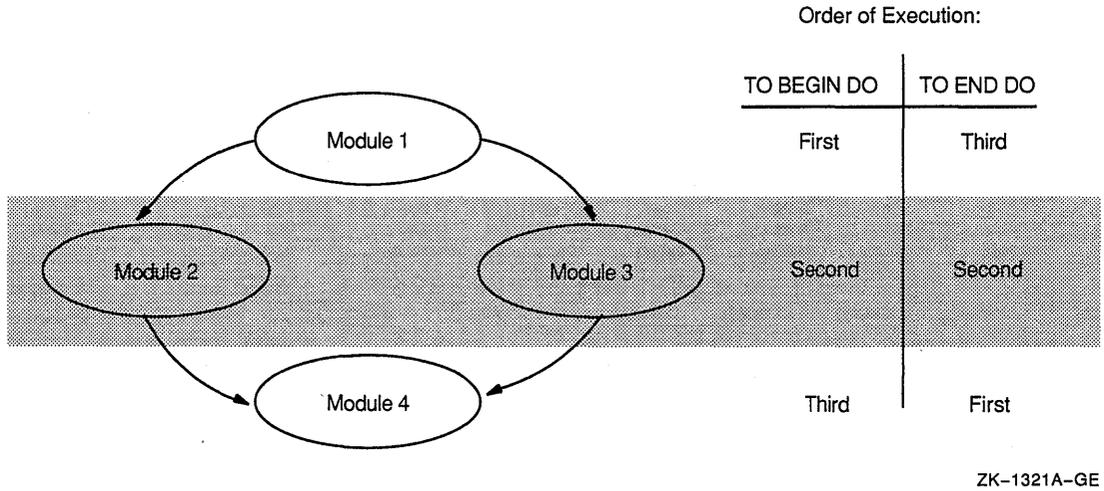
The execution order for initialization and finalization sections in Modules 2 and 3 cannot be determined. The headers the modules in Figure 3–1 are as follows:

```
[ENVIRONMENT]  MODULE Mod1; ...

[ENVIRONMENT, INHERIT( 'Mod1' )]  MODULE Mod2; ...

[ENVIRONMENT, INHERIT(' Mod1' )]  MODULE Mod3; ...

[INHERIT( 'Mod2', 'Mod3' )]       MODULE Mod4; ...
```

For More Information:

*   On modules (Section 7.3)

*   On environment files (Section 7.3.1.1)

3.4   The TO END DO Section

The TO END DO section allows you to specify a statement, in a module, that is to be executed after the executable section of the main program; it has the following form:

TO END DO statement;

statement
A VAX Pascal statement.

The TO END DO section can only appear in modules, can only appear once in a module, and must appear as the last section in the declaration section. (If appearing together, the TO END DO section must come after the TO BEGIN DO section at the end of the declaration section.)

As a general rule, if a compilation unit inherits an environment file, the finalization section in the inheriting compilation unit must be executed before the finalization section in the inherited compilation unit. Also, if more than one module with a finalization section inherits a single module, the order of finalization of the inheriting modules cannot be determined. Figure 3–1 illustrates an example of the order of execution of TO END DO sections.

Consider the following example:

```
MODULE File_Output;
VAR
    Out_File : TEXT;
    t        :, TIMESTAMP;
PROCEDURE Test(...); {Executable section...}

TO BEGIN DO
    OPEN( Out_File, 'foo.dat' );
    END;

TO END DO
    BEGIN
    GETTIMESTAMP( t );
    WRITELN( 'foo.dat closed at', TIME( t ) );
    CLOSE( Out_File )
    END;
END.
```

For More Information:

- On modules (Section 7.3)

- On environment files (Section 7.3.1.1)

3.5 The TYPE Section

The TYPE section introduces the name and set of values for a user-defined type or schema declaration; it has the following form:

TYPE
$\left\{ \begin{array}{l} \text{type-identifier} = \text{[[attribute-list]] type-denoter} \\ \text{schema-declaration} \end{array} \right\}$
[[VALUE initial-state-specifier]]};...

type-identifier
The identifier of the type being defined.

attribute-list
One or more identifiers that provide additional information about the type-denoter.

type-denoter
Any legal VAX Pascal type syntax.

schema-declaration
The declaration of a schema type.

initial-state-specifier
A compile-time expression that is assignment compatible with a variable of the TYPE identifier being defined. VAX Pascal initializes all variables declared to be of this type with the constant value or values provided (unless there is an overriding initial-state specifier in the variable declaration).

Pascal requires that all user-defined type identifiers (except base types of pointers) be defined before they are used in the definitions of other types. A base type must be defined before the end of the TYPE section in which it is first mentioned.

The following rules apply to the use of initial-state specifiers on data types:

* You must initialize a type with a compile-time expression of an assignment-compatible type. Scalar types require scalar constants; structured types require constant constructors.

* You cannot initialize file types.

* The predeclared function ZERO can be used to initialize an entire type (except file types) to binary zero.

* The constant identifier NIL or a call to the ZERO function are the only values with which you can initialize a pointer type.

Consider the following example:

```
TYPE
   Ptr_to_Movie = ^Movie;    {Movie is defined later}
   Name  = PACKED ARRAY[1..20] OF CHAR; {Defined before used}
   Movie = RECORD
            Title, Director : Name;
            Year  : INTEGER;
            Stars : FILE OF Name;
            Next  : Ptr_to_Movie;
            END;
```

Consider the following examples of type definitions:

```
TYPE
    Days_of_Week = ( Sun, Mon, Tues, Wed, Thurs, Fri, Sat )
                    VALUE Mon;
    Array_Template( Upper_Bound : INTEGER ) =
                    ARRAY [1..Upper_Bound] OF INTEGER;
VAR                     {Declaring variables of user-defined types:}
    week1, week2, week3, week4 : Days_of_Week; {Initial value: Mon}
    week5 : Days_of_Week VALUE Sun;            {Initial value: Sun}
    array_type2 : array_template( x ); {x is a run-time expression}
```

For More Information

* On data types (Chapter 2)

* On schema types (Section 2.5)

* On pointers (Section 2.3)

* On attributes (Chapter 10)

3.6 The VALUE Section

If you choose, you can use the VALUE section as a VAX Pascal extension that initializes ordinal, real, array, record, set, and string variables. (If you require portable code, use the VALUE reserved word in either TYPE definitions or VAR declarations.) The exact format of an initialization depends on the type of the variable being initialized. The VALUE section has the following form:

```
VALUE
    {variable-identifier := constant-expression};...
```

variable-identifier
The name of the variable to be initialized. You cannot specify a list of variable identifiers. You can initialize a variable or variable component only once in the VALUE section. Any variables appearing in the VALUE section must appear in a previous VAR section.

constant-expression
Any constant expression that is assignment compatible with the variable identifier.

Unlike other declaration sections, the VALUE section can appear only in a program or module declaration section. You cannot use the VALUE declaration section in procedures or functions. If you wish to initialize variables in procedures and functions, use an initial-state specifier (by using the VALUE reserved word in either the TYPE or VAR section).

You can assign values to complete structured variables or to a single component of that variable.

For More Information:

- On data types (Chapter 2)
- On expressions (Section 4.1)

3.7 The VAR Section

The VAR section declares variables and associates each variable with an identifier, a type, and optionally an initial value; it has the following form:

```
VAR
    {{variable-identifier},... : [[attribute-list]] type-denoter
```

$$[[\left\{ \begin{array}{l} := \\ \text{VALUE} \end{array} \right\} \text{ initial-state-specifier }]] \};...$$

variable-identifier
The identifier of the variable being declared.

attribute-list
One or more identifiers that provide additional information about the variable.

type-denoter
Any legal VAX Pascal type syntax.

initial-state-specifier
Any constant expression that is assignment compatible with the variable identifier. The variable is initialized to this expression.

You can combine several identifiers in the same variable declaration if the variables are of the same type and are being initialized either with the same value or not at all.

Consider the following example:

```
TYPE
    Hours_Worked = ARRAY[1..10] OF INTEGER;
VAR
    Answer, Rumor : BOOLEAN;
    Temp          : INTEGER VALUE 60;
    Grade         : 'A'..'D';
    Weekly_Hours  : Hours_Worked VALUE [1..3 : 7; OTHERWISE 5];
```

The following rules apply to the use of initial-state specifiers on variables:

- You must initialize a variable with a constant expression of an assignment-compatible type. Scalar variables require scalar constants; structured variables require constant constructors.

- You cannot initialize file variables.

- You can use the predeclared function ZERO to initialize all or part of a variable (except file variables and components) to binary zero.

- The constant identifier NIL or a call to the ZERO function are the only values with which you can initialize a pointer variable.

A reference to a variable consists of the variable's use in one of the following situations:

- The variable or one of its components is passed as a VAR, %REF, or %DESCR parameter. The reference lasts throughout the call to the corresponding routine.

- The variable or one of its components is used on the left side of an assignment statement. The reference lasts throughout the execution of the statement.

- The variable or one of its components is accessed by a WITH statement. The reference lasts throughout the execution of the statement.

The existence of a variable reference sometimes prohibits certain operations from being performed on the variable. Such restrictions are noted throughout this manual.

For More Information:

- Constructors (Section 2.4)

- On data types (Chapter 2)

- On attributes (Chapter 10)

- On the ZERO function (Section 8.89)

- On pointers and NIL (Section 2.3)

- On the assignment and WITH statements (Chapter 5)

Chapter 4

Expressions and Operators

This chapter discusses the following:

- Expressions (Section 4.1)
- Operators (Section 4.2)
- Data type conversions (Section 4.3)

4.1 Expressions

VAX Pascal expressions consist of one or more operands that result in a single value. If the expression contains more than one operand, the operands are separated by operators. Operands include numbers, strings, constants, variables, and function designators. Operators include arithmetic, relational, logical, string, set, and typecast operators.

VAX Pascal recognizes two forms of expressions: **constant** expressions and **run-time** expressions. Constant expressions result in a value at the time you compile your program. These expressions can include constants, constant identifiers, operators, and some predeclared functions. Constant expressions cannot include the following:

- Variable references
- Schema discriminants
- Bound identifiers from conformant parameters
- Calls to user-defined functions
- Calls to EOF and EOLN predeclared functions
- Constructors of schema types or of types containing schema components

Run-time expressions can only result in a value at the time you execute your program. These expressions can include variables, predeclared functions, user-declared functions, and everything that a constant expression cannot contain.

When you form an expression, the operands must be of the same data type. Under some circumstances, the compiler performs data type conversions and allows you to form an expression with operands of different types.

VAX Pascal does not evaluate expressions contained within a single statement in a predictable order. Also, the compiler does not always evaluate all expressions in a single statement if the correct execution of the statement can be determined by evaluation of fewer expressions. For instance, some IF statement conditions can be determined TRUE or FALSE by only evaluating one of the Boolean expressions in the condition. Do not write code that depends on the evaluation order of expressions, and, in some cases, on the evaluation of all expressions in a single statement. If you require a predictable order of evaluation, you may wish to use the AND_THEN and OR_ELSE operators.

For More Information:

- On data type conversion (Section 4.3)
- On data types (Chapter 2)
- On evaluation of IF statement conditions (Section 5.7)
- On the AND_THEN and OR_ELSE logical operators (Section 4.2.3)

4.2 Operators

VAX Pascal provides several classes of operators. You can form complex expressions by using operators to combine constants, constant identifiers, variables, and function designators.

VAX Pascal also provides the assignment operator (:=) for use in assignment statements.

For More Information:

- Precedence of operators (Section 4.2.7)
- Assignment statement (Section 5.1)

4.2.1   Arithmetic Operators

An arithmetic operator provides a formula for calculating a value. Table 4–1 lists the arithmetic operators that you can use, in combination with numeric operands, to perform an arithmetic operation.

Table 4–1:   Arithmetic Operators

Operator	Example	Result
+	A+B	Sum of A and B
–	A–B	B subtracted from A
*	A*B	Product of A and B
**	A**B	A raised to the power of B
/	A/B	A divided by B
DIV	A DIV B	Result of A divided by B truncated toward zero
REM	A REM B	Remainder of A divided by B
MOD	A MOD B	Modulus of A with respect to B

The +, –, *, ** Operators:

Addition, subtraction, multiplication, and exponentiation operators can be used on integer, unsigned, real, DOUBLE, and QUADRUPLE operands. These operators produce a result of the same type as the values. In exponentiation operations, if the data types of the operands are not the same, the operand of the less-precise type is converted and the result is of the more-precise type.

When you use a negative integer as an exponent, the exponentiation operation may yield unexpected results. Table 4–2 shows the defined results of integers raised to the power of negative integers.

Table 4–2: Results of Negative Exponents

Base	Exponent	Result
0	Negative or 0	Error
1	Negative	1
–1	Negative and odd	–1
–1	Negative and even	1
Any other integer	Negative	0

For example, the expression $1^{(-3)}$ equals 1; $(-1)^{(-3)}$ equals –1; $(-1)^{(-4)}$ equals 1; and $3^{(-3)}$ equals 0.

The / Operator:

The division operator (/) can be used on integer, unsigned, real, DOUBLE, and QUADRUPLE operands. The division operator always produces a real result. This result may reflect some loss of precision as the compiler converts integer and unsigned operands to their real equivalents. With one or more operands of higher precision, the result is of the higher-precision type.

The DIV, REM, MOD Operators:

The DIV, REM, and MOD operators can be used only on integer and unsigned operands. DIV divides one integer or unsigned operand by the other, producing an integer or unsigned result. DIV truncates toward zero any remaining fraction and does not round the result. For example, the expression 23 DIV 12 equals 1, and (–5) DIV 3 equals –1.

REM returns the remainder after dividing the first operand by the second. Thus, 5 REM 3 evaluates to 2. Similarly, 3 REM 3 evaluates to 0 and (–4) REM 3 evaluates to –1.

MOD returns the modulus of the first operand with respect to the second. The result of the operation A MOD B is defined only when B is a positive integer. This result is always an integer between 0 and B–1. The modulus of A with respect to B is computed as follows:

- If A is greater than B, B is subtracted repeatedly from A until the result is a nonnegative integer less than B.

- If A is less than B and not negative, the result is A.

- If A is less than zero, B is added repeatedly to A until the result is a nonnegative integer less than B.

For example, 5 MOD 3 equals 2, (–4) MOD 3 equals 2, and 2 MOD 5 equals 2.

When both operands are positive, the REM and MOD operators return the same result. For example, 28 REM 5 equals 3 and 28 MOD 5 equals 3. However, when the first operand is negative, REM produces a negative or zero result, while MOD produces a positive or zero result. For example, (–42) REM 8 equals –2 and (–42) MOD 8 equals 6.

Enabling subrange checking ensures that a MOD operation is legal by verifying at run time that B is a positive integer.

Note that the use of negative integer and real number constants as operands in MOD and exponentiation operations may not produce the results you expect because the minus sign (–) is actually a negation operator. For example, the expression –2.0**2 is equivalent to the expression –(2.0**2) and produces the result –4.0. Therefore, you should enclose a negative constant in parentheses to make sure that it is interpreted as you intend. The expression (–2.0)**2 produces the result 4.0.

Table 4–3 lists the result types of arithmetic operations with operands of various types.

Table 4–3: Result Types of Arithmetic Operators

Operator	Type of Operands	Result Type
+ - * **	INTEGER, UNSIGNED, REAL, DOUBLE, QUADRUPLE	Same as the operands if both are of the same type; otherwise, the operand of the lower-ranked type is converted and the result is of the higher-ranked type.
/	INTEGER, UNSIGNED, REAL, DOUBLE, QUADRUPLE	One of the real types—REAL if the operands are of type REAL (or SINGLE) or a lower-ranked type; otherwise, the operand of the lower-ranked type is converted and the result is of the higher-ranked type.
DIV REM MOD	INTEGER and UNSIGNED	INTEGER if both operands are of type INTEGER; UNSIGNED if the operands are of mixed types or are both UNSIGNED; otherwise, an error occurs.

For More Information:

- On integers (Section 2.1.1)
- On real numbers (Section 2.2)
- On "more precise" and "less precise" operands, and on type conversions (Section 4.3)
- On using the CHECK attribute and SUBRANGE option for MOD run-time checking (Section 10.2.7)

4.2.2 Relational Operators

A relational operator tests the relationship between two ordinal, real, DOUBLE, or QUADRUPLE expressions and returns a Boolean result. If the relationship holds, the result is TRUE; otherwise, the result is FALSE. Table 4–4 lists the relational operators that you can apply to arithmetic operands. You can also apply some of the relational operators to string operands and to set operands.

Table 4–4: Relational Operators

Operator	Example	Result
=	A = B	TRUE if A is equal to B
<>	A <> B	TRUE if A is not equal to B
<	A < B	TRUE if A is less than B
<=	A <= B	TRUE if A is less than or equal to B
>	A > B	TRUE if A is greater than B
>=	A >= B	TRUE if A is greater than or equal to B

Note that operators designated with two characters must appear in the order specified and cannot be separated by a space.

For More Information:

- On relational operators in string expressions (Section 4.2.4)
- On relational operators in set expressions (Section 4.2.5)
- On the BOOLEAN data type (Section 2.1.4)

4.2.3 Logical Operators

A logical operator evaluates one or more Boolean expressions and returns a Boolean value. The logical operators are listed in Table 4–5.

Table 4–5: Logical Operators

Operator	Example	Result
AND	A AND B	TRUE if both A and B are TRUE
OR	A OR B	TRUE if either A or B is TRUE, or if both are TRUE
NOT	NOT A	TRUE if A is FALSE, and FALSE if A is TRUE
AND_THEN	A AND_THEN B	TRUE if both A and B are TRUE; forces left-to-right evaluation order with short circuiting
OR_ELSE	A OR_ELSE B	TRUE if either A or B is TRUE, or if both are TRUE; forces left-to-right evaluation order with short circuiting

The AND, AND_THEN, OR, and OR_ELSE operators combine two conditions to form a compound condition. The NOT operator reverses the value of a single condition so that if A is TRUE, NOT A is FALSE, and vice versa.

Normally, the compiler does not guarantee the evaluation order for logical operations. The AND_THEN and OR_ELSE operators force the compiler to evaluate an expression from left to right, stopping when the overall result can be determined (also called **short circuiting**). The following example forces the compiler to verify that an array element is within index bounds before evaluating the element's contents:

```
IF ( i < 11 ) AND_THEN ( Array_A[i] = 0 ) THEN
   WRITELN( 'Index bounds are legal and element contained 0' );
```

The following examples show logical expressions and their Boolean results:

```
    { Expressions:                      Results: }
( 4 > 3 ) AND ( 18 = 3 * 6 )          {TRUE}
( 3 > 4 ) OR  ( 18 = 3 * 6 )          {TRUE}
NOT ( 4 <> 5 )                        {FALSE}
( i < 11 ) AND_THEN ( Array_A[i] = 0 ) {Not known}
p = NIL  OR_ELSE  p^ = 0              {Not known}
```

Boolean variables and functions can be used as operands in logical expressions. Consider the following example:

```
Flag AND ODD( i )
```

Suppose that Flag is a Boolean variable and ODD(i) is a function that returns TRUE if the value of the integer variable i is odd and FALSE if the value of i is even. Both operands, Flag and ODD(i), must be TRUE for the expression to be TRUE.

For More Information:

*   On precedence of operators (Section 4.2.7)
*   On Boolean data type (Section 2.1.4)

4.2.4 String Operators

A string operator concatenates or compares character-string expressions. The result of the operation is either a string or a Boolean value. Table 4–6 lists the string operators.

Table 4–6: String Operators

Operator	Example	Result
+	A+B	String that is the concatenation of strings A and B
=	A=B	TRUE if strings A and B have equal ASCII values
<>	A<>B	TRUE if strings A and B have unequal ASCII values
<	A<B	TRUE if ASCII value of string A is less than that of string B
<=	A<=B	TRUE if ASCII value of string A is less than or equal to that of string B
>	A>B	TRUE if ASCII value of string A is greater than that of string B
>=	A>=B	TRUE if ASCII value of string A is greater than or equal to that of string B

With the plus sign (+), you can concatenate any combination of STRING and VARYING character strings, packed arrays of characters, and single characters.

The result of a string comparison depends on the ordinal value in the ASCII character set of the corresponding characters in the strings. For example:

```
'motherhood' > 'cherry pie'
```

This relational expression is TRUE because lowercase 'm' comes after lowercase 'c' in the ASCII character set. If the first characters in the strings are the same, VAX Pascal searches for differing characters, as in the following:

```
'string1' < 'string2'
```

This expression is TRUE because the digit 1 precedes the digit 2 in the ASCII character set.

The relational operators are legal for character strings of different lengths as well as for character strings of the same lengths. The shorter of the two character strings is padded with blanks for the comparison. The following two strings, for instance, result in a value of TRUE:

```
'John' < 'Johnny'
'abc' = 'abc   '
```

The EQ, NE, GE, GT, LE, and LT predeclared routines make string comparisons that are similar to the relational operators, but these routines do not pad strings of unequal length with blanks. Instead, they halt string comparison when they detect unequal lengths.

Enabling bounds checking causes the length of all character strings to be checked at run time for illegal operations.

For More Information:

- On Boolean data types (Section 2.1.4)
- On string data types (Section 2.6)
- On the ASCII character set (Appendix A)
- On the CHECK attribute and the BOUNDS option for run-time character-string checking (Section 10.2.7)
- On the EQ, NE, GE, GT, LE, and LT routines (Chapter 8)

4.2.5 Set Operators

A set operator forms the union, intersection, difference, or exclusive-OR of two sets, compares two sets, or tests an ordinal value for inclusion in a set. Its result is either a set or a Boolean value. Table 4–7 lists the set operators.

Table 4–7: Set Operators

Operator	Example	Result
+	A+B	Set that is the union of sets A and B
*	A*B	Set that is the intersection of sets A and B
–	A–B	Set of those elements of set A that are not also in set B
=	A=B	TRUE if set A is equal to set B
<>	A<>B	TRUE if set A is not equal to set B
<=	A<=B	TRUE if set A is a subset of set B
>=	A>=B	TRUE if set B is a subset of set A
IN	C IN B	TRUE if C is an element of set B

Most set operators require both operands to be set expressions. The IN operator, however, requires an ordinal expression as its first operand and a set expression as its second operand. The ordinal expression must be of the same type as the set's base type. For example:

```
2*3 IN [1..10]
```

The result of this IN operation is TRUE because 2 * 3 evaluates to 6, which is a member of the set [1..10].

The XOR predeclared routine can return the set of elements that do not appear in both sets.

For More Information:

- On the SET data type (Section 2.4.3)
- On the Boolean data type (Section 2.1.4)
- On the XOR function (Section 8.88)

4.2.6 Type Cast Operator

Normally, VAX Pascal associates each variable with one type: the type with which the variable was declared. In some systems' programming applications, you can perform operations more efficiently by relaxing VAX Pascal's strict type-checking rules. VAX Pascal provides the type cast operator for this purpose.

The type cast operator changes the context in which you can use a variable or an expression of a certain data type. The actual representation of the object being cast is not altered by the type cast operator. VAX Pascal overrides the type only for the duration of one operation. It has one of the following forms:

$$\left\{ \begin{array}{l} \text{variable-identifier} \\ \text{(expression)} \end{array} \right\} \ :: \ \text{type-identifier}$$

The type cast operator (::) separates the name of a variable or an expression in parentheses from its target type, the type to which it is being cast. The operator "alters" the type of the cast object at that point only. The compiler assumes that a type cast will not affect the object at any other point in the program. If the type cast is likely to affect the object elsewhere, you should declare the object with the VOLATILE attribute.

Once you cast a variable or an expression, the object has all the properties of its target type during the execution of the operation in which the type cast operator appears. A variable and its target type must have the same allocation size. Therefore, you cannot cast a conformant-array parameter, although you can cast a fixed-size component of a conformant-array parameter. A schema variable or parameter cannot be type cast since it does not have a size that is known at compile-time.

When you cast an expression in parentheses, the value of that expression can be either truncated on the left or padded on the left with zeros, so that the allocation size of the expression's value and its target type become the same. The type of a cast expression cannot be VARYING OF CHAR, a conformant-array parameter, or a discriminated schema. In addition, the target type of a cast expression cannot be VARYING OF CHAR or a discriminated schema.

Consider the following example:

```
TYPE
    F_Float = PACKED RECORD
                Frac1 : 0..127;
                Expo  : 0..255;
                Sign  : BOOLEAN;
                Frac2 : 0..65535;
                END;
VAR
    A : REAL;
{In the executable section:}
A::F_Float.Expo := A::F_Float.Expo + 1;
```

In this example, the record type F_Float illustrates the layout of an F_floating real number. The real variable A is cast as a record of this type, allowing you to access the fields containing the mantissa, exponent, sign, and fraction of A. Adding 1 to the field containing the exponent gives the same result as multiplying A by 2.0.

For More Information:

- On data types (Chapter 2)
- On the VOLATILE attribute (Section 10.2.41)
- On conformant-array parameters (Section 6.3.6.1)

4.2.7 Precedence of Operators

The operators in an expression establish the order in which VAX Pascal combines the operands. The compiler performs operations with higher-precedence operators before operations with lower-precedence operators. Table 4–8 lists the order of operator precedence, from highest to lowest (operators on the same line are of equal precedence).

Table 4–8: Precedence of Operators

Operators	Precedence
::	Highest
NOT	
**	
*, /, DIV, REM, MOD, AND, AND_THEN	

Table 4–8 (Cont.): Precedence of Operators

Operators	Precedence
+, –, OR, OR_ELSE, unary +, unary –	
=, <>, <, <=, >, >=, IN	Lowest

In VAX Pascal, operators of equal precedence (such as plus and minus) are combined from left to right within the expression.

You must use parentheses for correct evaluation of an expression that combines relational operators. Consider the following expression:

```
a<=x AND b<=y
```

Without parentheses, this expression is interpreted as A<= (X AND B) <=Y. The logical operator AND requires its operands X and B to be Boolean expressions and returns a Boolean result, which is then used as an operand in evaluating one of the relational operators (<=). This operation causes an error because you cannot use relational operators with Boolean operands. You can modify the expression with parentheses as follows:

```
( a<=x ) AND ( b<=y )
```

In the rewritten expression, the compiler combines the Boolean values of the two relational expressions with the AND operator.

You can use parentheses in an expression to force a particular order for combining the operands. For example:

Expression	Result
8 * 5 DIV 2–4	16
8 * 5 DIV (2–4)	–20

The compiler evaluates the first expression according to normal precedence rules.

First, 8 is multiplied by 5 and the result (40) is divided by 2. Then 4 is subtracted to get 16. The parentheses in the second expression, however, force the subtraction of 4 from 2 (yielding –2) to be performed before the division of 40 by –2. The result is –20.

Parentheses can help to clarify an expression. For instance, you could write the first example as follows:

```
( ( 8 * 5 ) DIV 2 ) -4
```

The parentheses eliminate any confusion about how the compiler associates the operands in the expression.

The desired results of your program should not depend on the order of subexpression evaluation. Unless you use the AND_THEN or OR_ELSE operators, the compiler does not guarantee the order in which subexpressions and complex expressions are evaluated. In fact, if the result of an expression can be determined without complete evaluation, VAX Pascal may only partially evaluate some logical operations.

Usually the order of evaluation does not prevent the correct result from being produced. However, order of evaluation is very important when you write logical operations involving function designators that have side effects. (A side effect is an assignment to a nonlocal variable or to a variable parameter within a function block.)

For example, the following IF statement contains two function designators for function f:

```
IF f( a ) AND f( b ) THEN {Statement...}
```

The compiler can evaluate these two function designators in any order. Regardless which function designator the compiler evaluates first, if the result is FALSE, the other function designator does not have to be evaluated.

Suppose that function f assigns the value of its parameter to a nonlocal variable. Because you cannot know which function designator was evaluated first, you cannot be sure of the value of the nonlocal variable after the IF statement is performed.

For More Information:

- On expressions (Section 4.1)
- On the AND_THEN or OR_ELSE logical operators (Section 4.2.3)
- On user-defined functions (Chapter 6)
- On optimization, compiler switches, and order of evalutation (*VAX Pascal Reference Supplement for VMS Systems*)

4.3 Type Conversions

Since VAX Pascal is a strongly typed language, you cannot normally treat a value of one type as though it were of a different type, as you can in many languages. For example, you cannot assign the character '1' to a variable of type INTEGER, because '1' is not an integer constant but a character constant. However, there are times when it makes sense to combine values of two different types because the values have some aspect in common. For example, suppose you wish to add a value of type REAL to a value of type INTEGER. This operation is legal because the value of type INTEGER is converted to its equivalent value of type REAL before the operation is performed. The result of the operation is of type REAL.

In VAX Pascal, values are converted from one type to another when the conversion is required for an operation, an assignment, or a formal/actual parameter association. Before any type conversion, the arithmetic types are ranked as follows, from lowest to highest:

- INTEGER
- UNSIGNED
- REAL or SINGLE
- DOUBLE
- QUADRUPLE

Similarly, the character types are also ranked as follows, from lowest to highest:

- CHAR
- PACKED ARRAY OF CHAR
- STRING or VARYING OF CHAR strings

When values of two different arithmetic or character types are combined in an expression, the lower-ranked operand is converted to its equivalent in the higher-ranked type. The result of an operation in which conversion occurs is always of the higher-ranked type.

Conversions to values of type UNSIGNED are never checked for overflow. When combined with other unsigned values, negative integer values are converted to large unsigned values by the calculation of the modulus with respect to 2^{32}.

A special case of conversion can occur when you attempt to assign an expression of type VARYING OF CHAR or STRING strings to a variable of type PACKED ARRAY OF CHAR or if you try to pass a string expression to a formal value parameter of type PACKED ARRAY OF CHAR. If the varying-length string has less than or exactly the same number of components as the packed array, the varying-length string is converted to a packed array of characters before the assignment is made and padded with blanks as necessary. If you attempt to perform this assignment with a varying-length string that has more components than the packed array, a run-time error occurs.

For More Information:

For information on data types, see Chapter 2.

Chapter 5

Statements

VAX Pascal statements specify actions to be performed and appear in executable sections. This chapter discusses the following statements:

- Assignment statement (Section 5.1)
- CASE statement (Section 5.2)
- Compound statement (Section 5.3)
- Empty statement (Section 5.4)
- FOR statement (Section 5.5)
- GOTO statement (Section 5.6)
- IF statement (Section 5.7)
- Procedure call (Section 5.8)
- REPEAT statement (Section 5.9)
- WHILE statement (Section 5.10)
- WITH statement (Section 5.11)

When coding, separate statements with a semicolon (;). Since the semicolon is not syntactically part of a statement, it is not included in the syntax examples in this chapter.

5.1 Assignment Statement

The assignment statement uses an assignment operator (:=) to assign a value to a variable or to a function identifier. An assignment statement has the following form:

variable-access := expression

variable-access

An identifier, array component, record component, pointer dereference, pointer-function dereference, or file buffer.

expression

A run-time expression whose type is assignment compatible with the type of the variable. The value of the expression is the value assigned to the variable.

You cannot assign values to a variable of a record type with variants if you allocated this variable using the NEW procedure. You can assign values to a field of such a record variable.

Consider the following example:

```
VAR
    x      : INTEGER;
    y, z   : RECORD
        f1 : real;
        f2 : integer
        END;
{In the executable section:}
x := 1;  {type of expression is same type as the variable}
y := z;  {variables are assignment compatible}
Func_Return_Ptr_To_Integer( 3 )^ := 19;
```

For More Information:

- On the NEW procedure (Section 8.50)

- On assigning constructor values to structured variables (Section 2.4)

- On assignment compatibility (Section 2.9.2)

5.2 CASE Statement

The CASE statement causes one of several statements to be executed. Execution depends on the value of an ordinal expression called the **case selector**. A CASE statement has the following form:

```
CASE case-selector OF
    [[{{case-label-list};...: statement};...]]
    [[ [[;]] OTHERWISE {statement};... ]]
    [[;]]
END
```

case-selector

An expression of an ordinal type.

case-label-list

One or more case labels of the same ordinal type as the case selector, separated by commas. A case label can be a single constant expression, such as 1, or a range of expressions, such as 5..10.

statement

Any statement to be executed depending on the values of both the case-selector and the case-label.

You can specify case labels in any order within the case label list. Each case label can appear only once within a given CASE statement.

At run time, the system evaluates the case selector expression and chooses which statement to execute. If the value of the case selector does not appear in the case label list, the system executes the statement in the OTHERWISE clause. If you omit the OTHERWISE clause, the value of the case selector must be equal to one of the case labels. If the value is not equal to a label, the CASE statement result is undefined.

Consider the following example:

```
CASE Age OF
   1..4   : School := 'preschool';    {Subranges}
   5..8   : School := 'elementary';
   9..13  : School := 'middle';
   14..18 : BEGIN
            School := 'high';
            WRITELN( 'Difficult years!' );
            END;                        {Compound statements}
   19     : School := 'reform';       {Single ordinal value}
   OTHERWISE  School := 'graduated'; {If 1 > Age > 18 ...}
   END;
```

For More Information:

- On ordinal values (Section 2.1)

- On using the CHECK attribute to check selectors at run time (Section 10.2.7)

5.3 Compound Statement

A compound statement groups a series of statements so that they can appea anywhere that language syntax calls for a single statement. A compound statement has the following form:

```
BEGIN
{statement};...
END
```

statement
Any VAX Pascal statement, including other compound statements.

The statements that make up the compound statement must be separated with semicolons (;), although the semicolon before the END delimiter is optional.

Consider the following example:

```
IF a < 10 THEN
    BEGIN          {A compound statement}
    x := 10;
    y := 20;
    z := x + y;
    END            {No semicolon in THEN clause before an ELSE}
ELSE z := 29;      {A single statement}
```

For More Information:

*   On program executable sections (Section 7.3)

*   On function and procedure executable sections (Section 6.1)

5.4 Empty Statement

The empty statement causes no other action to occur than the advancement of program flow to the next statement. To use the empty statement, place a semicolon where the language syntax calls for a statement.

Consider the following example:

```
CASE Alphabetic OF
    'A','E','I','O','U' : Alpha_Flag := Vowel;
                    'Y' : ;   {Empty statement as selector; no action}
    OTHERWISE Alpha_Flag := Consonant;
    END;
```

5.5 FOR Statement

The FOR statement is a looping statement that repeats execution of a statement according to the value of a control variable. The control variable assumes a value within a specified range or set. A FOR statement has one of the following forms:

FOR control-variable := initial-value $\left\{ \begin{array}{c} \text{TO} \\ \text{DOWNTO} \end{array} \right\}$ final-value DO

    statement

FOR control-variable IN set-expression DO
    statement

control-variable
The name of a previously declared variable of an ordinal type.

initial-value
final-value
Expressions that form a range and whose type is assignment compatible with the type of the control variable.

set-expression
An expression resulting in a value of SET type. The base type of the set must be assignment compatible with the control variable.

statement
Any VAX Pascal statement that does not change the value of the control variable.

At run time, the initial and final values or the set expression is evaluated before the loop body is executed. Execution or termination of the statement occurs in the following cases:

- In the TO form, VAX Pascal checks to see if the value of the control variable is less than or equal to the final value. If this condition is met, the control variable takes on the value of the initial value for the first loop iteration. During iterations, the control variable increments according to its data type. Looping ceases when the control variable is greater than the final value.

- In the DOWNTO form, VAX Pascal checks to see if the value of the control variable is greater than or equal to the final value. If this condition is met, the control variable takes on the value of the initial value for the first loop iteration. During iterations, the control variable decrements according to its data type. Looping ceases when the control variable is less than the final value.

- In the set expression form, VAX Pascal checks to see if the set expression is not the empty set. If this condition is met, the control variable takes on the value of one of the members of the set. Iterations occur for each member of the set; the selection order of members of the set is undefined. Looping stops after the loop body executes for each member of the set.

In both the TO and the DOWNTO forms, incrementation of the control variable depends on its type. For example, values expressed in type INTEGER increment or decrement in units of 1. Values expressed in type CHAR increment or decrement in accordance with the ASCII collating sequence.

After normal termination of the FOR statement, the control variable does not retain a value. You must assign a new value to this variable before you use it elsewhere in the program. If the FOR loop terminates with a GOTO statement, the control variable retains the last assigned value. In this case, you can use the variable again without assigning a new value.

Consider the following examples:

```
FOR Year := 1899 DOWNTO 1801 DO   {Print leap years in 1800's}
   IF ( Year MOD 4 ) = 0 THEN
      WRITELN( Year:4, ' is a leap year' );

FOR I IN Set1 DO  {Set2 members are successors of Set1 members}
      Set2 := Set2 + [I + 1];
```

For More Information:

- On ordinal values (Section 2.1)
- On sets (Section 2.4.3)

5.6 GOTO Statement

The GOTO statement causes an unconditional branch to a statement prefixed by a label. A GOTO statement has the following form:

GOTO label

label

An unsigned decimal integer or symbolic name that represents a statement label.

The GOTO statement must be within the scope of the label declaration. A GOTO statement that is outside a structured statement cannot jump to a label within that structured statement. A GOTO statement within a routine can branch to a labeled statement in an enclosing block only if the labeled statement appears in the block's outermost level. Consider the following example:

```
FOR I := 1 TO 10 DO
   BEGIN
   IF Real_Array[I] = 0.0 THEN
      BEGIN
      Result := 0.0;
      GOTO 10;        {Use GOTO to exit from loop}
      END;
   Result := Result + 1.0/Real_Array[I];   {Compute sum of inverses}
   END;

10: Invertsum := Result;
```

For More Information:

- On label declarations (Section 3.2)

- On exiting FOR loops using GOTO (Section 5.5)

5.7 IF Statement

The IF statement tests a Boolean expression and performs a specified action if the result of the test is TRUE. The ELSE clause, when it appears, executes only if the test condition results to FALSE. An IF statement has the following form:

IF boolean-expression THEN statement1 [[ELSE statement2]]

boolean-expression
Any Boolean expression.

statement1
The statement to be executed if the value of the Boolean expression is TRUE.

statement2
The statement to be executed if the value of the Boolean expression is FALSE.

If an IF statement contains an ELSE clause, the statement in the THEN clause cannot be followed with a semicolon (;), since that completes the IF statement and separates it from the following statement. The following examples contain correct code:

```
IF x > 10 THEN y := 4          IF x > 10 THEN BEGIN y := 4;
        ELSE y := 5;                              z := 5;
                                           END
                                    ELSE y := 5;
```

The ELSE clause always modifies the closest IF-THEN statement. Use caution to avoid logic errors in nested IF statements, as in the following:

```
IF A = 1 THEN      {First IF}
   IF B<>1 THEN    {Second IF}
      C := 1
ELSE               {Appears to modify first IF}
   C := 0;         {Actually modifies second IF}
```

VAX Pascal may not always evaluate all the terms of a Boolean expression if it can evaluate the entire expression based on the value of one term. Either do not write code that depends on actual evalution (or evaluation order) of Boolean expressions, or use the AND_THEN and OR_ELSE operators for a predictable order of evaluation.

For More Information:

- On Boolean expressions (Section 2.1.4)

- On forming and evaluating expressions (Section 4.1)

- On the AND_THEN and OR_ELSE logical operators (Section 4.2.3)

5.8 Procedure Call

Syntactically, a procedure call is a statement. A procedure call has the following form:

routine-identifier [[({actual-parameter},...)]]

routine-identifier
The name of a procedure or function.

actual-parameter
An expression that is of a type that is compatible with the type of the formal parameter, or the name of a procedure or function.

In VAX Pascal, you can use procedure-call syntax to call a function, even though function calls are usually considered to be expressions. If you do this, the compiler invokes the function but ignores the return value.

For More Information:

For information on procedures and functions, see Chapter 6.

5.9 REPEAT Statement

The REPEAT statement is a looping statement and executes one or more statements until a specified condition is true. A REPEAT statement has the following form:

```
REPEAT
    {statement};...
UNTIL expression
```

statement
Any VAX Pascal statement.

expression
Any Boolean expression.

VAX Pascal always executes a REPEAT statement for one iteration; iterations continue as long as the Boolean expression is FALSE. When specifying more than one statement as the loop body to a REPEAT statement, do not enclose the statements with the BEGIN and END reserved words. Multiple statements are legal in the REPEAT loop body.

Consider the following example:

```
REPEAT
    READ( x );      {Attempts to read at least one character}
    IF ( x IN ['0'..'9'] ) THEN
        BEGIN       {Keep count of numbers and increase total}
        Digit_Count := Digit_Count + 1;
        Digit_Sum := Digit_Sum + ORD( x ) - ORD( '0' );
        END
    ELSE
        Char_Count := Char_Count+1;    {Count characters}
UNTIL EOLN(INPUT); {Reads from default device until end of line}
```

For More Information:

For information on Boolean expressions, see Section 2.1.4.

5.10 WHILE Statement

The WHILE statement is a loop that executes a statement while a specified condition is true. A WHILE statement has the following form:

WHILE expression DO
  statement

expression
Any Boolean expression.

statement
Any VAX Pascal statement.

VAX Pascal checks the value of the Boolean expression before executing the loop body for the first time; if the expression is FALSE, the loop body is not executed. If the initial value is TRUE, loop iterations continue until the condition is FALSE. When specifying more than one statement as the loop body to a WHILE statement, enclose the statements with the BEGIN and END reserved words, since the syntax calls for a single statement to follow the DO reserved word. If you do not use a compound statement for the loop body, VAX Pascal executes the first statement following the DO reserved word as the loop body.

Consider the following examples:

```
WHILE NOT EOF( File1 ) DO  {If EOF from the start, the loop}
    READLN( File1 );       {    body is not executed.       }
WHILE NOT EOLN( INPUT ) DO
    BEGIN                  {Use compound statement:}
    READ( x );
    IF NOT ( x IN ['A'..'Z', 'a'..'z', '0'..'9'] )
    THEN
        Err := Err + 1;    {Count odd characters as errors}
    END;
```

For More Information:

*   On Boolean expressions (Section 2.1.4)
*   On compound statements (Section 5.3)

5.11 WITH Statement

The WITH statement provides an abbreviated notation for references to the fields of a record variable or to the formal discriminants of a discriminated schema type. A WITH statement has the following form:

WITH { $\left\{ \begin{array}{l} \text{record-variable} \\ \text{schema-variable} \end{array} \right\}$ },... DO statement

record-variable
The name of the record variable being referenced.

schema-variable
The name of the variable being referenced whose type is a discriminated schema type. This underlying type of the schema can be a record.

statement
Any VAX Pascal statement.

The WITH statement allows you to refer to the fields of a record or to a formal discriminant of a schema by their names alone, rather than by the record.field-identifier or schema-variable.formal-discriminant syntax. In effect, the WITH statement opens the scope so that references to field identifiers or to formal discriminants alone are unambiguous. When you access a variable using a WITH statement, the reference syntax lasts only throughout the execution of the statement.

Specifying more than one variable has the same effect as nesting WITH statements. Consider the following example:

```
{The record Dog is nested in the record Cat:}
WITH Cat, Dog DO    {Specify Cat before Dog}
   Bills := Bills + Cat_Vet + Dog_Vet;

WITH Cat DO   {This is equivalent to the previous WITH}
     WITH Dog DO
          Bills := Bills + Cat_Vet + Dog_Vet;
```

If you are specifying nested records, their variable names must appear in the order in which they were nested in the record type definition. If you are working with record and schema variables that are not nested, you can specify variable names in any order. If you specify record or schema variables whose field names or formal discriminants conflict with one another, VAX Pascal uses the last record or schema in the comma list. Consider the following example:

```
VAR
    x : STRING( 10 );
    y : STRING( 15 );

{In the executable section:}
WITH x, y DO
    WRITELN( CAPACITY );        {y.CAPACITY is used}

{The following is equivalent:}
WITH x DO
    WITH y DO
        WRITELN( CAPACITY );
```

For More Information:

•   On records (Section 2.4.2)

•   On schema types (Section 2.5)

Chapter 6

Procedures and Functions

Procedures and functions are subprograms. A **procedure** contains one or more statements to be executed once the procedure is called. A **function** contains one or more statements to be executed once the function is called; in addition, functions return a single value. This manual refers to functions and procedures collectively as **routines**.

This chapter discusses the following information about user-defined routines:

* Routine declarations (Section 6.1)
* Routine calls (Section 6.2)
* Parameters (Section 6.3)

In addition to user-defined routines, VAX Pascal also allows you to access external routines (routines that are globally available on your system, which may or may not be written in VAX Pascal) and routines that are predeclared by the compiler.

For More Information:

* On predeclared routines (Chapter 8)
* On calling external routines (*VAX Pascal Reference Supplement for VMS Systems*)

6.1 Routine Declarations

You must declare a routine before you call it. Routine declarations have the
following formats:

[[attribute-list]] PROCEDURE routine-identifier [[(formal-parameter-list)]];

$$\left\{\begin{array}{l} \text{[[declaration-section]] BEGIN \{statement\};... END} \\ \left\{\begin{array}{l} \text{EXTERN} \\ \text{EXTERNAL} \\ \text{FORTRAN} \\ \text{FORWARD} \end{array}\right\} \end{array}\right\}$$

[[attribute-list]] FUNCTION routine-identifier [[(formal-parameter-list)]]
    : [[attribute-list]] result-type-id;

$$\left\{\begin{array}{l} \text{[[declaration-section]] BEGIN \{statement\};... END} \\ \left\{\begin{array}{l} \text{EXTERN} \\ \text{EXTERNAL} \\ \text{FORTRAN} \\ \text{FORWARD} \end{array}\right\} \end{array}\right\}.$$

attribute-list
One or more identifiers that provide additional information about the
type-denoter.

routine-identifier
The name of the routine. If you use the routine-identifier within the routine
body (with the exception of assigning a value to the routine-identifier of a
function), the result is a recursive call to the routine. The routine-identifier
of a procedure can be redeclared in the procedure's declaration-section.
The routine-identifier of a function cannot be redeclared in the function's
declaration-section; however, it can be redeclared in any nested routines
within the function's declaration-section.

formal-parameter-list
A comma list of the routine's formal parameters. A procedure can have as
many as 255 formal parameters; depending on the function return value,
some functions are limited to 254 formal parameters. Optionally, you can
specify a mechanism specifier and an attribute list for each parameter.

declaration-section

A routine declaration section can include all sections except TO BEGIN DO, TO END DO, and VALUE sections. Data specified in this declaration section is local to the routine and to any nested routines; you can redeclare identifiers that are declared in an outer block. You cannot redeclare a formal parameter identifier to be a local variable in the routine.

By default, the system does not retain the values of local variables after it exits from a routine. Each call to a routine creates copies of the local variables. This means you can call a routine recursively without affecting the values held by the local variables at each activation of the routine. To preserve the value of a local variable (not the copy) from one call to the next, you must declare the local variable with the STATIC attribute.

statement

Any VAX Pascal statement. In a function executable section, there must be at least one statement of the following form:

routine-identifier := result

The routine-identifier is the name of the function. The result is a value of either an ordinal, real, structured, or pointer type that VAX Pascal returns when function is called. (This value cannot be a file type or a structured type with a file component.) This value must be of the same type as the result-type-id.

EXTERN
EXTERNAL
FORTRAN
FORWARD

Predeclared identifiers that direct VAX Pascal to find the body of the routine elsewhere. The EXTERN, EXTERNAL, and FORTRAN identifiers declare routines that are independently compiled by VAX Pascal or that are written in other languages. In VAX Pascal, these identifiers are equivalent. Although not part of the Pascal standard, many Pascal compilers only accept the FORTRAN identifier for external routines actually written in FORTRAN; if portability is a concern, you may wish to use FORTRAN only for external FORTRAN routines.

The FORWARD identifier declares a routine whose block is specified in a subsequent part of the same procedure and function section, allowing you to call a routine before you specify its routine body. As an extension, VAX Pascal will allow the body to be in a different declaration part. If the body and heading are specified in different procedure and function sections, a FORWARD declared function should not be used as an actual discriminant to a schema type.

When you specify the body of the routine in subsequent code, include only the FUNCTION or PROCEDURE predeclared identifier, the routine-identifier, and the body of the routine. Do not repeat the formal-parameter, the attribute-list, or the result-type-id.

result-type-id
The type specification of the function return value. The function's result must be of this data type. This type cannot be a file type or a structured type with a file component.

Consider the following example:

```
{Function body contained in subsequent code:}
FUNCTION Adder( Op1, Op2, Op3 : REAL ) : REAL; FORWARD;

PROCEDURE Introduction;
VAR
    a, b, c, z : REAL;     {Variables local to the procedure}
BEGIN
WRITELN( 'This is the Inventory Program Version 5.6.' );
WRITELN;
WRITELN( 'Press CTRL/H for help.  Press RETURN to continue.' );
a := 4.6;    b := 12.1;   c := 201.45;
z := Adder( a, b, c );     {Call the function Adder}
END;

{System_Routine_Tanh available with the operating system:}
FUNCTION System_Routine_Tanh( Angle : REAL ) : REAL; EXTERNAL;

FUNCTION Adder;   {Do not repeat attributes or parameters}
BEGIN
Adder := Op1 + Op2 + Op3;    {Assign a function return value}
END;
```

For More Information:

- On attributes (Chapter 10)
- On declaration sections (Chapter 3)
- On the scope of identifiers (Section 7.2)
- On parameters and passing mechanisms (Section 6.3)
- On recursive function calls (*VAX Pascal User Manual*)
- On calling external routines (*VAX Pascal Reference Supplement for VMS Systems*)
- On functions limited to 254 parameters (*VAX Pascal Reference Supplement for VMS Systems*)

6.2 Routine Calls

A routine call executes all statements in the body of the declared routine. You must declare a routine before you can call it. In addition, function calls return a single value. Syntactically, procedure calls are statements, and function calls are expressions. You can call routines in the executable section of a program or in the body of another routine. Routine calls have the following forms:

$$\text{procedure-identifier } [[(\{ \left[\begin{array}{l} \%\text{IMMED} \\ \%\text{REF} \\ \%\text{DESCR} \\ \%\text{STDESCR} \end{array} \right] \text{actual-parameter}\},...)]]$$

$$\text{function-identifier } [[(\{ \left[\begin{array}{l} \%\text{IMMED} \\ \%\text{REF} \\ \%\text{DESCR} \\ \%\text{STDESCR} \end{array} \right] \text{actual-parameter}\},...)]]$$

procedure-identifier
function-identifier
The declared routine identifier.

actual-parameter
The actual parameter whose data type matches the type of the corresponding formal parameter. Optionally, you can specify a passing mechanism for each parameter.

Syntactically, procedure calls are statements, and function calls are expressions. You can call a function anywhere that an expression of the declared result type is legal.

If the result of a function is irrelevant, you can call the function as a statement, in the same way that you call a procedure.

The scope of a routine identifier is the block in which it is declared, excluding any nested blocks that redeclare the same identifier.

Consider the following example:

```
VAR
    a, b, c, z : REAL;

PROCEDURE Introduction;
BEGIN
WRITELN( 'This is the Inventory Program Version 5.6.' );
WRITELN;
WRITELN( ' Press CTRL/H for help.  Press RETURN to continue.' );
END;
```

```
FUNCTION Adder( Op1, Op2, Op3 : REAL ) : REAL;
BEGIN
Adder := Op1 + Op2 + Op3;    {Assign a function return value}
END;

{In the executable section:}
Introduction;    {No parameters necessary in the call}
a := 3.14;       b := 14.78;       c := 112.456;
z := Adder( a, b, c );  {Function used as an expression
                         evaluating to a REAL value}
```

If a function returns a value of an array, record, or pointer type, you can index, select, or dereference the object at the time of the function call, without first assigning the function result to a variable. Consider the following:

```
TYPE
    Player_Rec = RECORD
        Wins       : INTEGER;
        Losses     : INTEGER;
        Percentage : REAL;
        END;
VAR
    Number : INTEGER;

FUNCTION Return_Player_Info( Player_Num : INTEGER ) : Player_Rec;
    {In the function body:}
    Return_Player_Info := Player_Rec[Wins: 3; Losses: 18;
                                     Percentage: 21/3];
{In the executable section:}
WRITELN( Return_Player_Info( Number ).Losses, 'losses is poor!');
```

For More Information:

*   On expressions (Section 4.1)

*   On parameters and passing mechanisms (Section 6.3)

6.3 Parameters

In VAX Pascal, there are two types of parameters: formal and actual parameters. A **formal parameter** (also called an argument) is located in the header of the routine declaration. You cannot redeclare a formal parameter in a routine's declaration section, but you can redeclare it in nested routines within the routine's declaration section.

The formal parameter establishes the **semantics**, the data type, and the required **passing mechanism** of the parameter. The general format of the formal parameter list is as follows.

$$[[(\{\begin{array}{l}\text{value-parameter-spec}\\\text{variable-parameter-spec}\\\text{routine-parameter-spec}\\\text{foreign-parameter-spec}\end{array}\};...)]]$$

The specific format of a formal parameter specification depends on the semantics (value, variable, routine, foreign) of the formal parameter you are declaring (conformant parameters also have a unique syntax).

Table 6–1 presents the VAX Pascal semantics for formal parameters.

Table 6–1: Formal Parameter Semantics

Parameter Type	Description
Value	Used only to provide input to the routine. After calling the routine, the value of the actual parameter remains unchanged.
Variable	Used to allow access to the actual parameter. If the routine makes changes to the value of the formal parameter, the value of the actual parameter changes accordingly.
Routine	Used to call another routine.
Foreign	Used to call a routine written in another language.

At run time, the formal parameter receives a value from the corresponding **actual parameter**, which is located in the routine call. The passing mechanism is the way in which the compiler passes the actual parameter value to the corresponding formal parameter. Table 6–2 presents the passing mechanisms supported by VAX Pascal.

Table 6–2: Parameter Passing Mechanisms

Mechanism	Description
By immediate value	The information passed to the formal parameter is the data.
By reference	The information passed to the formal parameter is the address of the data. By default, VAX Pascal passes all actual parameters by reference, except for conformant parameters and except when the formal parameter is an undiscriminated schema parameter.

Table 6–2 (Cont.): Parameter Passing Mechanisms

Mechanism	Description
By descriptor	The information passed to the formal parameter is the address of a descriptor of the data. By default, VAX Pascal passes all conformant parameters and undiscriminated schema parameters by descriptor.

The actual parameter must be of the same data type and passing mechanism as the corresponding formal parameter. VAX Pascal uses the default passing mechanism for each actual parameter depending on its data type and formal definition. If the called routine is external to VAX Pascal, you can specify an explicit passing mechanism that overrides the type and number of formal parameters.

For More Information:

*   On undiscriminated schema types (Section 2.5)

*   On descriptors (*VAX Pascal Reference Supplement for VMS Systems*)

6.3.1 Value Parameters

By the rules of value semantics defined by the Pascal standard, a formal value parameter represents a local variable within the called routine. When you specify value semantics, the address of the actual parameter is passed to the called routine, which then copies the value from the specified address to its own local storage. The routine then uses this copy. The copy is not retained when control returns to the calling block. Therefore, if the called routine assigns a new value to the formal parameter, the change is not reflected in the value of the actual parameter.

When you do not include a reserved word before the name of a formal parameter, you automatically cause VAX Pascal to use value semantics to pass data to that parameter. A formal value parameter has the following form:

$$
\{\text{identifier}\},...\ :\ [[\text{attribute-list}]] \left\{ \begin{array}{l} \text{type-identifier} \\ \text{undiscriminated-schema-name} \\ \text{conformant-parameter-syntax} \end{array} \right\}
$$

[[:= [[mechanism-specifier]] default-value]]

identifier
The name of the formal parameter. Multiple identifiers must be separated with commas.

attribute-list
One or more identifiers that provide additional information about the formal parameter.

type-identifier
The type identifier of the parameters in this section.

undiscriminated-schema-name
The name of an undiscriminated schema type.

conformant-parameter-syntax
The type syntax of a conformant array or a conformant VARYING parameter.

mechanism-specifier
The mechanism by which a default value is to be associated with the formal parameter.

default-value
A compile-time expression representing the default value for the formal parameter.

Any attributes associated with a formal parameter become attributes of the local variable. They do not affect the values that can be passed to the parameter; they affect the behavior of the formal parameter only within the routine block. When a formal parameter has the UNSAFE attribute, the types of the actual parameters passed to it are not checked for compatibility.

An actual value parameter must be an expression whose type is assignment compatible with the type of the corresponding formal parameter. Because there is no assignment compatibility for file variables, undiscriminated schema sets, and undiscriminated schema subranges, they can never be passed as value parameters. Also, the names of routines are not allowed as value parameters.

If necessary, the type of an actual parameter is converted to the type of the formal parameter to which it is being passed. In this case, VAX Pascal follows the same type conversion rules that it uses to perform any other assignment. You may, for example, pass an integer expression to a formal parameter of a real type. If an actual parameter has the UNSAFE attribute, no conversion occurs.

If you have a user-defined, formal parameter of an undiscriminated schema type, the corresponding actual parameter must be discriminated from the same schema type as that of the formal parameter.

When you pass a string expression to a formal value parameter of type STRING, the actual parameter's current length (not its declared maximum length) becomes both the maximum length and the current length of the formal parameter.

You can also use the attributes [CLASS_S], [CLASS_A], and [CLASS_NCA] on value parameters if a routine requires a specific type of descriptor for VAX Pascal to build. A [CLASS_A], [CLASS_NCA] or [CLASS_S] formal value parameter requires the actual value parameters to be passed with the by descriptor mechanism.

Consider the following examples:

```
VAR
    Old_Number, x, y : INTEGER;
FUNCTION Random( Seed : INTEGER ): INTEGER;    {Function body...}
PROCEDURE Alpha( a, b : INTEGER; c : CHAR );   {Procedure body...}

{In the executable section:}
New_Number := Random( Old_Number );
Alpha ( x+y, 11, 'G' );   {Actual parameters are integer
                           and character expressions}
```

For More Information:

- On blocks and scope (Section 7.2)
- On conformant parameters (Section 6.3.6)
- On default values for formal parameters (Section 6.3.8)
- On mechanism specifiers (Section 6.3.4)
- On the UNSAFE attribute (Section 10.2.39)
- On type conversions (Section 4.3)

6.3.2 Variable Parameters

By the rules of variable semantics defined by the Pascal standard, a formal variable parameter represents another name for a variable in the calling block. It is preceded by the reserved word VAR. When you specify variable semantics, the address of the actual parameter is passed to the called routine. In contrast to value semantics, the called routine directly accesses the actual parameter. Thus, the routine can assign a new value to the formal parameter during execution and the changed value is reflected immediately in the calling block (the value of the actual parameter changes).

VAX Pascal uses variable semantics to pass data to a formal parameter, often called a formal VAR parameter, and has the following form:

VAR {identifier},... : [[attribute-list]] $\left\{ \begin{array}{l} \text{type-identifier} \\ \text{undiscriminated-schema-name} \\ \text{conformant-parameter-syntax} \end{array} \right\}$

   [[:= [[mechanism-specifier]] default-value]]

identifier
The name of the formal parameter. Multiple identifiers must be separated with commas.

attribute-list
One or more identifiers that provide additional information about the formal parameter.

type-identifier
The type identifier of the parameters in this parameter section.

undiscriminated-schema-name
The name of an undiscriminated schema type.

conformant-parameter-syntax
The type syntax of a conformant array or a conformant VARYING parameter.

mechanism-specifier
The mechanism by which a default value is to be associated with the formal parameter. A mechanism specifier can be used only on a declaration for an external routine.

default-value
A compile-time expression representing the default value for the parameter. A default value can be used only on an external routine.

When you use variable semantics, the actual parameter must be a variable or a component of an unpacked structured variable (you can pass an entire packed structure); no expressions are allowed unless the formal parameter has the READONLY attribute. The type of a variable passed to a routine must be structurally compatible with the type of the corresponding formal parameter, except for schema parameters. For a formal parameter that is an undiscriminated schema type, the type of the variable must be discriminated from the same type as that of the formal parameter (they must be of the same schema type family). For a formal parameter that is a discriminated schema type, the type of the variable must be of the same type family and must have equivalent actual discriminants.

The names of routines are never allowed as variable parameters. In addition, you must use variable semantics when passing a file variable as an actual parameter. Also, you cannot pass the tag field of a variant record to a formal VAR parameter.

Consider the following example:

```
VAR  My_String : VARYING [20] OF CHAR VALUE 'Harry Hayes';
PROCEDURE Name( VAR A_String : VARYING [String_Size] OF CHAR ); {Body}

{In the executable section:}
Name( My_String );
WRITELN( 'The new name is ', My_String );
```

This example declares a procedure, Name, which returns a new name through the formal parameter A_String. Procedure Name modifies the value of A_String; since My_String is passed by variable semantics, upon completion of the routine the modified value is reflected in the variable My_String.

In VAX Pascal, certain attributes in a routine declaration or a routine call affect the rules of compatibility between actual and formal VAR parameters. These rules also apply to the corresponding components of structured types and to the base types of pointer types used as formal parameters. The attributes that result in rule changes are the alignment, POS, READONLY, size, UNSAFE, VOLATILE, and WRITEONLY attributes.

You can also use the attributes [CLASS_S], [CLASS_A], and [CLASS_NCA] on variable parameters if a routine requires a specific type of descriptor for VAX Pascal to build. A [CLASS_A], [CLASS_NCA] or [CLASS_S] formal variable parameter requires the actual variable parameters to be passed with the by descriptor mechanism.

For More Information:

- On blocks and scope (Section 7.2)
- On conformant parameters (Section 6.3.6)
- On default values for formal parameters (Section 6.3.8)
- On mechanism specifiers (Section 6.3.4)
- On attributes and parameter compatibility (Chapter 10)
- On type conversions (Section 4.3)

6.3.3 Routine Parameters

To write a routine that invokes another routine whose effect is not determined until the program is executed, use routine parameters. To declare a procedure or a function as a formal parameter to another routine, you must include a complete routine heading in the formal parameter list. You can also associate a foreign mechanism specifier and a default value with a formal procedure or function parameter.

The following examples show formal routine parameter sections in procedure and function declarations:

```
PROCEDURE Apply( FUNCTION Operation( Left, Right : REAL ) : REAL;
                 VAR Result : REAL );
FUNCTION Copy( PROCEDURE Get_Char( VAR c : CHAR );
               PROCEDURE Put_Char( i : CHAR ) ) : BOOLEAN;
```

The identifiers listed as formal parameters to a formal procedure or function parameter are not accessible outside the routine declaration; they indicate the number and kind of actual parameters necessary. You refer to these identifiers only when you use nonpositional syntax to call a routine parameter.

In the previous example, the formal parameter list of Get_Char informs the compiler that Copy must pass one character parameter to Get_Char using variable semantics. Copy does not refer explicitly to the formal parameter C unless it calls Get_Char using nonpositional syntax.

To pass a routine as an actual parameter, the formal parameter list of the routine being passed and the routine specified as the formal parameter must be congruent. Two formal parameter lists are congruent if they have the same number of sections and if the sections in corresponding positions meet any of the following conditions:

- Both are value parameter sections containing the same number of parameters. The types of parameters must either be compatible or be equivalent conformant parameters.

- Both are variable parameter sections containing the same number of parameters. The types of the parameters must either be compatible or be equivalent conformant parameters. Any attributes associated with a formal variable parameter affect the kinds of actual parameters that can be passed to it.

- Both are procedure parameter sections having either congruent formal parameter lists or no formal parameters.

- Both are function parameter sections having either congruent formal parameter lists or no formal parameters, and having compatible result types.

- Both are foreign parameter sections having the same mechanism specifier and the same number of parameters, and whose types must be compatible.

- If one formal parameter list has a LIST attribute on its last parameter section, the other formal parameter list must also have this attribute.

The following program shows a function declaration that includes two functions as formal parameters:

```
VAR
    Costs, Pay, Fedtax, Food : REAL;
    Housing                  : INTEGER;

FUNCTION Income( Salary, Tax : REAL ) : REAL;   {Function body...}

FUNCTION Expenses( Rent : INTEGER; Grocery : REAL ) : REAL;
    {Function body...}

FUNCTION Budget( FUNCTION Credit( Earnings, UStax : REAL ) : REAL;
  FUNCTION Debit( Housing : INTEGER; Eat : REAL ) : REAL ) : REAL;
    VAR    Deduct : REAL;
    BEGIN {FUNCTION Budget}
    Deduct := Debit( Eat := Food, Housing := Housing );
    Budget := Credit( Pay, Fedtax ) - Deduct;
    END;

{In the executable section:}
Costs := Budget( Income, Expenses );
```

When the function Budget is called, the function Income is passed to the formal function parameter Credit, and the function Expenses is passed to the formal function parameter Debit. When Credit is called, the program-level variables Pay and Fedtax are substituted for Credit's formal parameters, Earnings and UStax. In the call to Debit, nonpositional syntax is used to associate Debit's formal parameters Housing and Eat with the program-level variables Housing and Food. Note that the names of program-level variables do not conflict with formal parameters of routine parameters.

The presence of the ASYNCHRONOUS and UNBOUND attributes in routine declarations causes additional requirements to be imposed on the routines that can legally be passed as actual parameters.

For More Information:

- On routine headings (Section 6.1)
- On positional syntax (Section 6.3.7)
- On default values for formal parameters (Section 6.3.8)
- On mechanism specifiers (Section 6.3.4)
- On conformant parameters (Section 6.3.6)
- On attributes (Chapter 10)

6.3.4 Foreign Parameters

When declaring an external routine (one written in a language other than Pascal) that is called by a VAX Pascal routine, you must specify not only the correct semantics but the correct mechanism as well. To allow you to obtain these passing mechanisms, VAX Pascal provides foreign mechanism specifiers and the passing mechanism attributes. Table 6–3 gives the method specifier or attribute used for each passing mechanism.

Table 6–3: Specifiers and Attributes for Passing Mechanisms

Passing Mechanism	Specifiers and Attributes
By immediate value	%IMMED or [IMMEDIATE]
By reference	%REF or [REFERENCE]
By descriptor	%DESCR or %STDESCR

The foreign mechanism specifier %IMMED, %REF, %DESCR or %STDESCR precedes a formal parameter in the declaration of an external routine. If the formal parameter does not represent a routine, the mechanism specifier must precede the parameter name. If the formal parameter represents a routine, the specifier must precede the reserved word PROCEDURE or FUNCTION in the parameter declaration.

In addition, it is possible to use the passing mechanism attributes [IMMEDIATE] or [REFERENCE] in a formal parameter's attribute list to obtain the same behavior as %IMMED or %REF, respectively.

When calling an external routine, you must make sure that you pass actual parameters by the mechanism stated or implied in the routine declaration. VAX Pascal allows you to use the foreign mechanism specifiers %IMMED, %REF, %DESCR, and %STDESCR before an actual parameter in a routine call. (Passing-mechanism attributes are valid only on formal

parameters.) When a mechanism specifier appears in a call, it overrides the type, semantics, mechanism specified, and even the number of parameters in the formal parameter declaration. Thus, type checking is suspended for the parameter association to which the specifier applies.

The passing of an expression to a foreign mechanism parameter implies foreign value semantics: the calling block makes a copy of the actual parameter's value and passes this copy to the called routine. The copy is not retained when control returns to the calling block. Foreign value semantics differs from value semantics in that the calling block, not the called routine, makes the copy.

The passing of a variable to a foreign mechanism parameter (except a parameter with the %IMMED or [IMMEDIATE] specifier) implies foreign variable semantics: the variable itself is passed.

A compile-time warning occurs if the compiler must convert the value of an actual parameter variable to make it match the type of a foreign mechanism parameter. In that case, the compiler passes a copy of the converted value by foreign value semantics, using the specified mechanism. You can eliminate this warning by enclosing the actual parameter variable in parentheses; by doing so, you prevent the compiler from interpreting the actual parameter as a variable. The compiler takes the same action, whether or not it produces a warning message.

Mechanism specifiers on formal parameters produce the following results:

- A %REF or [REFERENCE] formal parameter requires actual parameters to be passed by reference. %REF or [REFERENCE] implies variable semantics unless the actual parameter is an expression; in that case, it implies foreign value semantics.

- An %IMMED or [IMMEDIATE] formal parameter requires actual parameters to be passed with the by immediate value mechanism and always implies value semantics. %IMMED or [IMMEDIATE] cannot be used on formal parameters of type VARYING, or on conformant array and conformant VARYING parameters.

- A %DESCR formal parameter requires actual parameters to be passed with the by descriptor mechanism and interprets the semantics as %REF or [REFERENCE] does.

- A %STDESCR formal parameter requires actual parameters to be passed with the by string descriptor mechanism. An actual parameter variable of type PACKED ARRAY OF CHAR implies variable semantics. An actual parameter expression of either type PACKED ARRAY OF CHAR or type VARYING OF CHAR implies foreign value semantics. You cannot use %STDESCR on formal procedure and function parameters.

Because the semantics are implicit in the mechanism, a formal parameter cannot be declared with both the reserved word VAR and a mechanism specifier.

Also, when passing an actual parameter to a formal foreign parameter, the VAX Pascal compiler checks for type compatibility when an external routine is called. However, at the time of the declaration, a formal parameter passed by immediate value that does not represent a routine is checked to ensure that it can be stored in 32 or fewer bits. A formal parameter passed by immediate value that does represent a routine must be declared with the UNBOUND attribute.

Special considerations arise when a function that has no formal parameters of its own (or that has defaults that are being used for all its formal parameters) and is passed as a formal parameter to another routine. The appearance of the function identifier in an actual parameter list could indicate the passing of either the address of the function or the function result. In VAX Pascal, the address of the function is passed by default. Therefore, to cause the function result to be passed, you must enclose the function identifier in parentheses.

Consider the following example:

```
p( %IMMED f );    {Address of function f is passed}
p( %IMMED (f) );  {Result of function f is passed}
```

For More Information:

- On conformant parameters (Section 6.3.6)

- On type conversions (Section 4.3)

- On attributes (Chapter 10)

- On calling external routines (*VAX Pascal Reference Supplement for VMS Systems*)

- On compiler messages (*VAX Pascal Reference Supplement for VMS Systems*)

6.3.5 Schema Parameters

VAX Pascal provides a method of processing schematic arrays, records, sets, subranges, and STRINGs with potentially different actual discriminants. To do this, you can use undiscriminated schema parameters.

An undiscriminated schema formal parameter is a type name that represents a specific schema family. The actual discriminants are determined each time you pass a corresponding actual parameter. The actual discriminants are available within the routine through the formal parameter.

You can use undiscriminated schema formal parameters when declaring value and variable parameters. When you use a formal, undiscriminated schema parameter instead of a conformant parameter or a type identifier, a call to the routine provides actual parameters that are discriminated from the same schema type family. However, when using value semantics, there are two exceptions, as follows:

- You cannot pass schema sets and schema subranges as value parameters, since there is no assignment compatibility for undiscriminated sets and for undiscriminated subranges.

- You can pass a varying-length string expression to a formal STRING parameter (the actual parameter does not have to be of the STRING type).

Consider the following example:

```
TYPE
    Array_Template( lbnd, hbnd : INTEGER ) =
        ARRAY[lbnd..hbnd] OF INTEGER;
VAR
    Even_Numbers : Array_Template( 1, 30 );
    Odd_Numbers  : Array_Template( 1, 60 );

PROCEDURE Print_Array( Array_To_Print : Array_Template );
    VAR
        i : INTEGER;
    BEGIN
    WRITELN( 'The maximum number of elements is ',
            Array_To_Print.hbnd );
    WRITELN;
    FOR I := LOWER( Array_To_Print ) TO UPPER( Array_To_Print ) DO
        WRITELN( Array_To_Print[i] );
    END;

{In the executable section:}
Print_Array( Even_Numbers );
Print_Array( Odd_Numbers );
```

All passing mechanism specifiers and attributes (%REF, %IMMED, %DESCR, %STDSCR, [REFERENCE], [IMMEDIATE], [CLASS_S], [CLASS_A], [CLASS_NCA]) are illegal on parameters of nonstatic types.

For More Information:

- On schema types (Section 2.5)

- On the STRING predefined schema type (Section 2.6.3)

6.3.5.1 Schema Parameter Sections

When you specify more than one formal schema parameter of the same schema type in a single parameter section, there are additional programming considerations.

If you specify more than one formal parameter (separated by commas) of a single, user-defined, undiscriminated schema type, the corresponding actual parameters must be discriminated from the same type as the formal parameter (they must be of the same schema type family), and the actual schema parameters must have equivalent actual discriminant values. Consider the following example:

```
TYPE
    Array_Template( Upper_Bound : INTEGER )
                 = ARRAY[1..Upper_Bound] OF INTEGER;
VAR
    Actual_1, Actual_2 : Array_Template( 10 );
    Actual_3           : Array_Template( 20 );

    PROCEDURE Schema_Proc1( Only_One : Array_Template ); {Body...}
    PROCEDURE Schema_Proc2( One, Two : Array_Template ); {Body...}

{In the executable section:}
Schema_Proc1( Actual_1 );                {Legal}
Schema_Proc1( Actual_3 );                {Legal}
Schema_Proc2( Actual_1, Actual_2 );      {Legal}
Schema_Proc2( Actual_1, Actual_3 );      {Illegal}
```

When using value semantics, if you specify more than one formal parameter (separated by commas) of an undiscriminated STRING type, the corresponding actual parameters must have equivalent current values (not necessarily equivalent maximum lengths). Consider the following example:

```
VAR
    One_String, Two_String : STRING( 15 );
    Three_String           : STRING( 20 );

PROCEDURE Test_Strings( One, Two : STRING ); {Body...}

{In the executable section:}
Test_Strings( 'a', 'b' );        {Legal}
Test_Strings( 'a', 'bb' );       {Illegal}
One_String := 'Hello';
Two_String := 'Hello there';
Three_String := 'olleH';
Test_Strings( One_String, Two_String );   {Illegal}
Test_Strings( One_String, Three_String ); {Legal}
```

When using variable semantics, if you specify more than one formal parameter (separated by commas) of an undiscriminated STRING type, the corresponding discriminated, STRING, actual parameters must have an equivalent maximum length.

For More Information:

- On user-defined schema types (Section 2.5)
- On the STRING predefined schema type (Section 2.6.3)

6.3.6 Conformant Parameters

VAX Pascal provides a method of processing arrays and character strings with potentially different maximum lengths. To do this, you can use conformant array parameters or conformant varying parameters.

A conformant parameter is a syntax that represents a set of types that are identical except for their bounds. The bounds of a conformant parameter are determined each time a corresponding actual parameter is passed. The bounds of an actual parameter are available within the routine through identifiers declared in the conformant parameter. A conformant parameter can appear only within a formal parameter list.

You can use conformant parameters when declaring value, variable, and foreign mechanism parameters. When you use a conformant parameter instead of a type identifier in a formal parameter declaration, a call to the routine can provide static and nonstatic arrays, VARYING OF CHAR strings, and discriminated strings (of the STRING schema family) of any size.

In addition, two conformant parameters are equivalent if they have indexes of the same ordinal type and components that either are compatible or are equivalent conformant parameters. They must also have the same number of dimensions and both must be packed or unpacked.

6.3.6.1 Conformant Array Parameters

The syntax for a conformant array has the following form:

```
ARRAY[{lower-bound-identifier..upper-bound-identifier :
    [[attribute-list]] index-type-identifier};...] OF [[attribute-list]]
    ⎰ type-identifier                ⎱
    ⎱ conformant-parameter-syntax ⎰

PACKED ARRAY[lower-bound-identifier..upper-bound-identifier :
    [[attribute-list]] index-type-identifier] OF [[attribute-list]] type-identifier
```

lower-bound-identifier
An identifier that represents the lower bound of the conformant array's index.

upper-bound-identifier
An identifier that represents the upper bound of the conformant array's index.

attribute-list
One or more identifiers that provide additional information about the conformant array.

index-type-identifier
The type identifier of the index, which must denote an ordinal type.

type-identifier
The type identifier of the array components, which can denote any type.

To specify the range and type of the index, you must use type identifiers that represent predefined or user-defined ordinal types. The identifiers that represent the index bounds can be thought of as READONLY value parameters, implicitly declared in the procedure declaration.

Unless the conformant array is packed, the component can be either a type identifier or another conformant parameter; therefore, only the last dimension of a conformant parameter can be packed. For example, the following is illegal because the component of the packed array in this example is another conformant parameter:

```
PACKED ARRAY[l1..u1: INTEGER; l2..u2: INTEGER] OF CHAR
```

However, the following is allowed because only the last component is packed:

```
ARRAY[l1..u1: INTEGER] OF PACKED ARRAY[l2..u2: INTEGER] OF CHAR
```

Consider the following example:

```
TYPE
   Workdays = 1..31;
   Feb_Days = 1..28;
   Mar_Days = 1..31;
VAR
   Feb_Arr             : ARRAY[Feb_Days] OF INTEGER;
   Mar_Arr             : ARRAY[Mar_Days] OF INTEGER;
   Feb_Total, Mar_Total : INTEGER;

FUNCTION Inventory( VAR Amt_Sold :
   ARRAY[First_Day..Last_Day : Workdays] OF INTEGER ) : INTEGER;
```

```
{In executable section:}
{Amt_Sold : ARRAY[1..28] OF INTEGER...}
Feb_Total := Inventory( Feb_Arr );

{Amt_Sold : ARRAY[1..31] OF INTEGER...}
Mar_Total := Inventory( Mar_Arr );
```

The formal parameter Amt_Sold can have index values from 1 to 31 to indicate the number of workdays in each month. Thus, an actual parameter passed to Amt_Sold could be an array whose index type is either Feb_Days or Mar_Days. Using a conformant parameter in this example allows you to write a general-purpose routine that sums the components of Amt_Sold and returns the monthly inventory total to the calling block.

For More Information:

* On ordinal types (Section 2.1)

* On arrays (Section 2.4.1)

* On attributes (Chapter 10)

6.3.6.2 Conformant VARYING Parameter

The syntax for a conformant VARYING string has the following form:

VARYING [upper-bound-identifier] OF [[attribute-list]] CHAR

attribute-list
One or more identifiers that provide additional information about the conformant VARYING string.

upper-bound-identifier
An identifier that represents the upper bound of the conformant VARYING OF CHAR string's index. The type of the upper bound identifier is always integer.

The upper bound identifier specifies the maximum length of the VARYING OF CHAR string and must denote an integer. The upper bound identifier that represents the maximum length can be thought of as a READONLY value parameter, implicitly declared in the procedure declaration.

When you pass a string expression to a value conformant varying-length parameter, the length of the actual parameter's current value parameter (not its declared maximum length) becomes both the current length and the maximum length of the formal parameter. When you pass either a conformant VARYING OF CHAR string variable or a discriminated STRING variable to a VAR conformant varying-length parameter, the declared maximum length of the actual parameter becomes the maximum length of the formal parameter.

The following example shows how to declare and use a VARYING OF CHAR conformant parameter:

```
VAR
    Short_String : VARYING[40] OF CHAR := PAD( '', '-', 40 );
    Long_String  : VARYING[80] OF CHAR := PAD( '', '-', 80 );

PROCEDURE Dashed_Line( VAR String : VARYING[Len] OF CHAR ); {Body...}

{In the executable section:}
Dashed_Line( Short_String );
Dashed_Line( Long_String );
```

In this example, note that Len is not a previously declared identifier but is instead an additional implicit parameter defined by the procedure declaration. The upper bound of the conformant parameter String is established by the declared maximum length of the actual parameter passed to it when the procedure Dashed_Line is called. The first call to Dashed_Line passes a 40-character string, so Len has the value 40. The second call passes an 80-character string, so Len has the value 80.

For More Information:

• On VARYING OF CHAR data types (Section 2.6.2)

• On attributes (Chapter 10)

6.3.6.3 Conformant Parameter Sections

When you specify more than one conformant parameter of the same type in a single parameter section, there are additional programming considerations.

When using value semantics, if you specify more that one formal parameter (separated by commas) of a single, user-defined, conformant parameter, the corresponding actual parameters must have either of the following characteristics:

• Equivalent current lengths (in the case of passing a string expression to a conformant PACKED array parameter of type CHAR or to a conformant VARYING OF CHAR)

• Indexes that are equivalent and of the same ordinal type, the same number of dimensions, and components that are compatible (in the case of passing a static or nonstatic array to a conformant array parameter)

Consider the following example:

```
VAR
    Actual_1, Actual_2 : ARRAY[1..10] OF INTEGER;
    Actual_3          : ARRAY[5..10] OF INTEGER;

PROCEDURE TestArr( One, Two : ARRAY [L1..U1 : INTEGER] OF INTEGER );
          {Procedure body...}
PROCEDURE TestStr( One, Two : PACKED [L2..U2 : INTEGER] OF CHAR );
          {Procedure body...}

{In the executable section:}
TestArr( Actual_1, Actual_2);       {Legal}
TestArr( Actual_2, Actual_3);       {Illegal}
TestStr( 'ABC', 'XYZ' );            {Legal}
TestStr( 'HELLO', 'GOODBYE' );      {Illegal}
```

When using variable semantics, if you specify more than one formal parameter (separated by commas) of a single, user-defined, conformant parameter, the corresponding actual variable parameters must be of the same type.

For More Information:

*   On ordinal types (Section 2.1)
*   On static and nonstatic types (Section 2.8)

6.3.7 Parameter Association

In most cases, a routine call must pass exactly one actual parameter for each formal parameter. The actual parameter is either listed explicitly in the routine call or supplied by means of a default value in the routine declaration.

One way of establishing the correspondence between actual and formal parameters is to give the parameters in each list the same position. That is, the association of actual and formal parameters proceeds from left to right, item by item, through both lists. This form of association is called **positional syntax**.

Another way of establishing correspondence is to specify the formal parameter name and the actual parameter being passed to it. In VAX Pascal, you can associate an actual with a formal parameter using the assignment operator (:=). The actual parameters in the call do not have to appear in the same order as the formal parameters appeared in the declaration. This form of association is called **nonpositional syntax**.

You can use both positional and nonpositional actual parameters in the same call. However, after you specify one parameter in nonpositional syntax, all remaining parameters must be in nonpositional syntax (all parameters in positional syntax must be at the front of the list).

Consider the following example:

```
PROCEDURE Compute_Sum( x, y : INTEGER; VAR z : INTEGER ); {Body...}

{In the executable section:}
{Positional syntax:}
Compute_Sum( Quantity + 6, 15, Total );

{Nonpositional syntax:}
Compute_Sum( z := Total, x := Quantity + 6, y := 15 );

{Both syntaxes:}
Compute_Sum( Quantity + 6, z := Total, y := 15 );
```

For More Information:

*   On using the LIST and TRUNCATE attributes to specify variable-length parameter lists (Sections 10.2.22 and 10.2.36)
*   On scope (Section 7.2)

6.3.8   Default Formal Parameters

VAX Pascal allows you to supply default values for formal parameters. Using default parameter values, you do not need to pass actual parameters. Also, you can specify an actual parameter in the position of a formal parameter whose default value you want to override. To specify a default value, use the following format:

parameter-spec := [[mechanism-specifier]] constant-expression;

parameter-spec
The parameter specification. The syntax for a parameter specification depends on its semantics.

mechanism-specifier
The mechanism by which VAX Pascal associates a default value with a formal parameter. You can only specify a mechanism-specifier on a foreign routine.

constant-expression
A constant expression representing the default value for the formal parameter.

This default value, plus the optional mechanism specifier, must be a legal actual parameter for the kind of formal parameter with which the default is associated. Table 6–4 shows when VAX Pascal allows a default value.

Table 6–4: Default Values on Formal Parameters

Formal Parameter Type	External Routine	VAX Pascal Routine
Variable	Requires foreign mechanism specifier	Error
Value	Allowed	Allowed
Routine	Requires foreign mechanism specifier	Error

When you declare a formal parameter with a default value, you can either omit it from the routine call or, if you use positional syntax, you can indicate its position with a comma.

Consider the following example:

```
FUNCTION Net_Pay( Hours    : INTEGER;
                  Tax      : REAL := 0.05;
                  Rate     : REAL;
                  Fica     : REAL := 0.07;
                  Overtime : INTEGER )  : REAL;   {Body...}
{In the executable section:}
{Nonpositional syntax:}
Take_Home_Year := Take_Home_Year +
                  Net_Pay ( Overtime := Overtime_Week,
                            Rate := Pay_Rate,
                            Hours := Hours_Week );
{Positional syntax:}
Take_Home_Year := Take_Home_Year +
                  Net_Pay( Hours_Week, , Pay_Rate, ,
                           Overtime_Week );
```

The formal parameters Tax and Fica are given the default values 0.05 and 0.07, respectively.

You can override a formal parameter's default value by associating the formal parameter with an actual parameter in a routine call. For example, if you wanted to replace the default value of the formal parameter Tax in the previous example for one call, you could call Net_Pay as follows:

```
Take_Home_Year:= Take_Home_Year +
   Net_Pay( Hours_Week, 0.06, Pay_Rate, , Overtime_Week );
```

As a result of this routine call, the default value of Tax would be replaced by the value 0.06 supplied in the actual parameter list.

For More Information:

• On specifying passing mechanisms (Section 6.3.4)

• On postional and nonpositional syntax of parameters (Section 6.3.7)

Chapter 7

Program Structure and Scope

This chapter describes the following:

- Blocks (Section 7.1)
- Scope of Identifiers (Section 7.2)
- Modules and Programs (Section 7.3)

7.1 Blocks

A block is a declaration section and an executable section. Programs, modules, and routines are structured in blocks. A declaration section can contain routine blocks nested within the outer program or module block; routine blocks can also be nested within other routines.

The declaration section contains data definitions and declarations, and nested routine declarations that are local to the enclosing block. The executable section contains the statements that specify the block's actions. You can cause an exit from a block with the last executable statement of the block, which causes normal termination, or with a GOTO statement, which transfers control to an outer block.

For More Information:

- On declaration sections (Chapter 3)
- On routine declarations (Section 6.1)
- On the GOTO statement (Section 5.6)

7.2 Scope of Identifiers

The scope of an identifier is the part of the program in which the identifier has a particular meaning. In VAX Pascal, the scope of an identifier is the block in which it is defined or declared, including nested blocks (but excluding any nested blocks that redeclare the same identifier). Outside its scope and assuming that it is not declared elsewhere, an identifier has no meaning; in this case, attempts to use the identifier generate an error.

All VAX Pascal identifiers observe the following scope rules:

- An identifier can be declared only once within a particular scope.

- A previously declared identifier can be redeclared in a nested block.

- An identifier declared in the main program or module block is accessible in all nested blocks (except where it is redeclared).

Figure 7–1 illustrates the scope of identifiers that appear in several blocks in a program. VAX Pascal scope rules make the following statements about Figure 7–1 true:

- Variable identifiers A and B are declared at the outer level of the example. Scope rules make them accessible throughout the example. In the main program, Level1a, and Level1b, identifiers A and B represent integers. In Level2, A still represents an integer variable, but B is redeclared as a Boolean variable.

- Type identifier C and variable identifiers D and E are declared at the next lower lever, Level1a. Scope rules make them accessible only in this block. They cannot be accessed from the higher-level main program. They cannot be accessed from a lower-level block because Level1a contains no nested routine.

- Formal parameter identifiers V, U, and T are declared at the same level, Level1b. They cannot be redeclared within this block. They could be redeclared as local identifiers in the nested block of Function Level2.

- Procedure identifier Level1a is declared in the outermost block of the example. This identifier could be redeclared within its own declaration section.

- Function identifier Level2 is declared in the next-highest level, Level1b. It cannot be declared within this block. Level2 could be redeclared as a local identifier within a nested block.

Figure 7–1: Scope of Identifiers

```
PROGRAM Scope ( INPUT, OUTPUT );

VAR
     a, b : INTEGER;
     .
     .
     .

     PROCEDURE Level1a ( z, y : INTEGER );

          TYPE
              c = ARRAY[1 .. 35] OF CHAR;

          VAR
              d, e : c ;
              .
              .
              .

          END;        {End procedure Level1a}

     PROCEDURE Level 1b (v, u : CHAR; VAR t : INTEGER );
              .
              .
              .

              FUNCTION Level2: CHAR;

                   VAR
                       b : BOOLEAN;
                       .
                       .
                       .

                   END;    {End function Level2}
              .
              .
              .

          END;        {End procedure Level1b}
     {Executable section of program Scope:}
              .
              .
              .
```
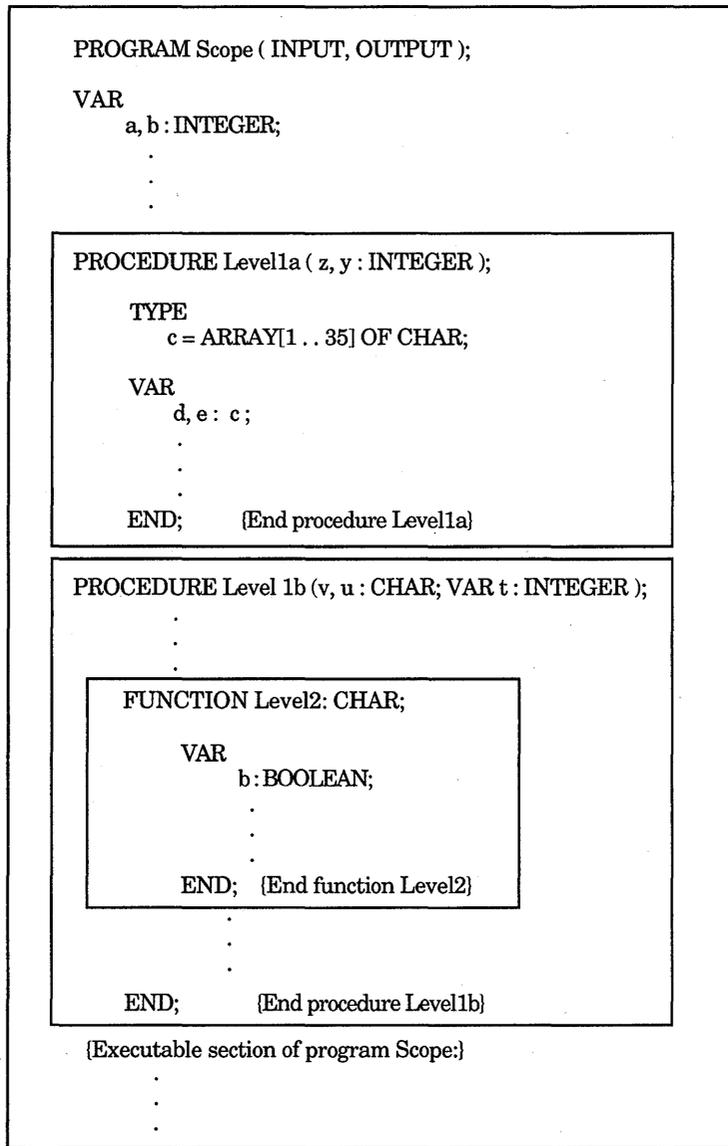
ZK–1112A–GE

For More Information:

- On scope of routine identifiers (Section 6.1)
- On scope of formal parameters (Section 6.3)
- On scope of labels and GOTO statements (Section 5.6)

7.3 Modules and Programs

A **module** is a set of instructions that can be compiled, but not executed, by itself. Module blocks contain only a declaration section and TO BEGIN DO and TO END DO sections. A **program** is a set of instructions that can be compiled and executed by itself. Program blocks contain a declaration and an executable section. A **compilation unit** is a unit of VAX Pascal code that can be compiled independently; the term compilation unit refers to either a program or a module.

Each module and program must be in a separate file; you cannot place multiple modules (or a module and a program) in the same file. You can compile modules and a program together or separately (the syntax of compilation depends on the operating system command-line interpreter you are using).

The formats for compilation units are as follows:

```
[[attribute-list]] PROGRAM comp-unit-identifier [[({file-identifier},...)]];
    [[declaration-section]]
    BEGIN
    {statement};...
    END.
```

```
[[attribute-list]] MODULE comp-unit-identifier [[({file-identifier},...)]];
    [[declaration-section]]
    [[TO BEGIN DO statement;]]
    [[TO END DO statement;]]
    END.
```

The module syntax of VAX Pascal is slightly different than that of Extended Pascal. However, the concepts in both languages are the same.

attribute-list
One or more identifiers that provide additional information about the compilation unit.

comp-unit-identifier

Specifies the name of the program or module. The identifier appears only in the heading and has no other purpose within the compilation unit.

file-identifier

Specifies the names of any file variables associated with the external files used by the compilation unit.

declaration-section

A VAX Pascal declaration section.

statement

A VAX Pascal statement.

The program or module heading includes all information preceding the program or module block. If your program contains any input or output routines, you must list all the external file variables that you are using in the compilation unit's heading. File variables listed in a heading must also be declared locally in the block, except for the predeclared file variables INPUT and OUTPUT. The INPUT identifier corresponds to a predefined external file that accepts input from the default device (usually, your terminal); the OUTPUT identifier corresponds to a predefined external file that sends output to the default device (usually, your terminal). If you redeclare INPUT and OUTPUT in a nested block, you lose access to the default input and output devices.

Consider the following example:

```
PROGRAM Write_Var(OUTPUT);    {Header}
VAR                           {Declaration section}
   Number : INTEGER VALUE 3;
BEGIN                         {Executable section}
WRITELN( Number );            {Writes 3 to the default device}
END.
```

For More Information:

* On INPUT and OUTPUT (Section 9.5)

* On compilation and command-line syntax (*VAX Pascal Reference Supplement for VMS Systems*)

* On the TO BEGIN DO section (Section 3.3)

* On the TO END DO section (Section 3.4)

7.3.1 Compilation Units and Data Sharing

When dividing code into programs and modules, you may want to share declarations among compilation units. The following sections discuss ways of sharing data.

7.3.1.1 Environment Files

Environment files contain the definitions and declarations, from the outermost level of declaration section of a compilation unit, that can be shared with other compilation units that inherit the environment file. Environment files exist in a form that the compiler can process more easily. The following example shows how to use environment files:

```
{Contained in one file:}
[INHERIT ('File_Name')]
PROGRAM a( INPUT, OUTPUT );
BEGIN
READ( Amount );
Calc;
WRITELN( 'Purchase Amount:  ', Amount:10:2)
WRITELN( '                +  ', Tax:10:2)
WRITELN( 'Pay This Total:   ', Total:10:2)
END.

{Contained in a separate file:}
[ENVIRONMENT('File_Name')]
MODULE B;         {Keep all global data in one module}
CONST             {Compile this unit first to create file}
    Rate = 0.06;
VAR
    Amount, Total, Tax: REAL;
PROCEDURE Glf; BEGIN...END;
PROCEDURE Calc;
    BEGIN
    Tax := Amount * Rate;
    Total := Tax + Amount;
    Glf;
    END;
END.
```

Since the declarations in module B compose the environment file to be inherited by another compilation unit, you must compile module B first to create the environment file. When compiling module B, VAX Pascal creates an environment file by the name of File_Name, with a default file type of .PEN (for Pascal Environment).

The environment file contains declaration information about the constant Rate; about the variables Amount, Total, and Tax; and about the procedures Glf and Calc. If there are identifiers in a compilation unit that you do not want to be included in an environment file, use the HIDDEN attribute.

When you compile the program A, which contains the INHERIT attribute, VAX Pascal uses the specified environment file to allow the program access to the data and routines declared in the module. Variables that are inherited from an environment file are not newly created variables, but are the same variables that were allocated storage by the declaring compilation unit.

A compilation unit may create only one environment file, but may inherit multiple files that must have been created by earlier compilations. Declarations from inherited files are not included in any environment files created by the compilation unit. An environment file must have been created by a version of the compiler that is compatible with the version that is compiling the compilation unit.

The identifiers in the outermost level of the declaration section of a compilation unit and all inherited identifiers must be unique. However, VAX Pascal allows the following exceptions to the redeclaration rules:

- A variable identifier can be multiply declared if all declarations of the variable have the same type, and all but one declaration at most are external.

- A procedure identifier can be multiply declared if all declarations of the procedure have congruent parameter lists, and all but one declaration at most are external.

- A function identifier can be multiply declared if all declarations of the function have congruent parameter lists and identical result types, and all but one declaration at most are external.

Consider the following example:

```
{In one compilation unit:}
[ENVIRONMENT('EXTERN.PEN')] MODULE Mod1;
[EXTERNAL] PROCEDURE Inst; EXTERN;
END.

{In another compilation unit:}
[INHERIT('EXTERN.PEN')] MODULE Mod2;
[GLOBAL] PROCEDURE Inst;  {Body...}
END.
```

For More Information:

- On the ENVIRONMENT attribute (Section 10.2.12)

- On the INHERIT attribute (Section 10.2.19)

- On the HIDDEN attribute (Section 10.2.16)

- On VAX Pascal versions and environment file compatibility (Appendix C)
- On declaration sections (Chapter 3)

7.3.1.2 Global and External Identifiers

Global and external identifiers are accessible to other compilation units (even to compilation units written in other languages) that make up one executable unit. The GLOBAL attribute allows a declared identifier to be globally accessible by other compilation units; the EXTERNAL attribute specifies that another compilation unit allocates storage for the data or routine. The compiler does not check to make sure that global and external identifiers are declared as being of the same data type; you must ensure that the data types are compatible.

You cannot use global and external names to share the declarations of user-defined types. However, you can use the VALUE attribute to share global and external literals.

The following example shows how to use global and external identifiers:

```
{Contained in one file:}
PROGRAM a( INPUT, OUTPUT );
VAR
    Amount, Total, Tax : [EXTERNAL] REAL;   {Defined elsewhere}
[EXTERNAL] PROCEDURE Calc; EXTERNAL;        {Defined elsewhere}
[GLOBAL]   PROCEDURE Glf; {Body...}         {Available outside}
BEGIN
READ( Amount );
Calc;
WRITELN( 'Purchase Amount:  ', Amount:10:2)
WRITELN( '                +  ', Tax:10:2)
WRITELN( 'Pay This Total:   ', Total:10:2)
END.

{Contained in a separate file:}
MODULE B;
CONST
    Rate = 0.06;
VAR
    Amount, Total, Tax: [GLOBAL] REAL;  {Available outside}
[EXTERNAL] PROCEDURE Glf; EXTERNAL;     {Defined elsewhere}
[GLOBAL] PROCEDURE Calc;                {Available outside}
    BEGIN
    Tax := Amount * Rate;
    Total := Tax + Amount;
    Glf;
    END;
END.
```

The file containing the module can be compiled separately from the file containing the program.

For More Information:

- On the GLOBAL attribute (Section 10.2.15)
- On the EXTERNAL attribute (Section 10.2.13)

DECLIT AA VAX L369D

VAX PASCAL reference manual

Predeclared Routines

VAX Pascal supplies predeclared procedures and functions that perform various commonly used operations. You do not have to declare these routines in order to call them from your code.

In this chapter, the routines are presented in alphabetical order. Also in this chapter, the term "arithmetic types" refers to those data types that can be used in arithmetic operations: INTEGER, UNSIGNED, and the real types.

In some sections of this manual, reference is made to entire categories of routines. Table 8–1 lists the routines in each category.

Table 8–1: Predeclared Routine Categories

Category	Category Description and Routines
Allocation size	Routines that provide information about the amount of storage allocated for variables and for components of various types: BITNEXT, BITSIZE, NEXT, and SIZE
Arithmetic	Routines that perform mathematical computations: ABS, ARCTAN, COS, EXP, LN, MAX, MIN, SIN, SQR, SQRT, UAND, UNOT, UOR, UXOR, and XOR
Character-string	Routines that manipulate character strings: BIN, DEC, FIND_MEMBER, FIND_NONMEMBER, GE, GT, HEX, INDEX, LE, LENGTH, LT, NE OCT, PAD, READV, STATUSV, SUBSTR, UDEC, and WRITEV
Component position	Routines that provide information about the offset of record components: BIT_OFFSET and BYTE_OFFSET

Table 8–1 (Cont.): Predeclared Routine Categories

Category	Category Description and Routines
Date-time	Routines that provide information on the calendar date and time: CLOCK DATE, GETTIMESTAMP, and TIME
Dynamic allocation	Routines that provide for the creation and use of pointer variables: ADDRESS, DISPOSE, IADDRESS, and NEW
Input and Output	Routines that you use for I/O; these routines are not described in this chapter
Low-level	Routines that allow for parallel processes and for asynchronous routines to operate in a real-time or multitasking environment: ADD_INTERLOCKED, CLEAR_INTERLOCKED, FIND_FIRST_BIT_CLEAR, FIND_FIRST_BIT_SET, and SET_INTERLOCKED
Ordinal	Routines that provide information on the ordered sequence of values: PRED, SUCC, LOWER, and UPPER
Parameter	Routines that give information about variable-length parameter lists: ARGUMENT, ARGUMENT_LIST_LENGTH, and PRESENT
Privileged	Routines that manipulate privileged hardware registers: MFPR and MTPR
Transfer	Routines that convert an actual parameter to data of another type: CHR, DBLE, INT, ORD, PACK, QUAD, ROUND, SNGL, TRUNC, UINT, UNPACK, UROUND, and UTRUNC
Miscellaneous	CARD, CREATE_DIRECTORY, DELETE_FILE, ESTABLISH, EXPO, HALT, ODD, RENAME_FILE, REVERT, UNDEFINED, and ZERO

For More Information:

- On ordinal types (Section 2.1)
- On arrays and records (Section 2.4)
- On pointers (Section 2.3)
- On data types (Chapter 2)
- On parameter lists (Section 6.3)
- On input and output routines (Section 9.6)
- On data storage (*VAX Pascal Reference Supplement for VMS Systems*)

3.1 ABS Function

The ABS function returns a value (of the same data type as the specified parameter) that is the absolute value of the parameter.

ABS(x)

The parameter x can be of any arithmetic type.

3.2 ADD_INTERLOCKED Function

The ADD_INTERLOCKED function adds the value of an expression to the value of a variable, stores the newly computed value in the variable, and returns an integer value: −1 if the new value is negative, 0 if it is zero, and 1 if it is positive.

ADD_INTERLOCKED(e, v)

The type of the expression e must be assignment compatible with that of the variable v. The variable v must be an integer or an unsigned subrange; v must have an allocation size of two bytes and must be aligned on a word boundary. The type of e must be assignment compatible with that of v.

Note that unless the type of v is an integer subrange that includes negative values, the result of the ADD_INTERLOCKED function is never −1.

Overflow and subrange checking are never performed on the ADD_INTERLOCKED operation, even if these options are in effect for the rest of the function or compilation unit.

3.3 ADDRESS Function

The ADDRESS function returns a pointer value that is the address of the parameter.

ADDRESS(x)

The parameter x can be a variable of any type except a component of a packed structured type. A compile-time warning results if x is a formal VAR parameter, a component of a formal VAR parameter, or a variable that does not have the READONLY or VOLATILE attribute.

A pointer can only refer to a VOLATILE variable or a variable allocated by the NEW procedure.

For More Information:

- Pointer data type (Section 2.3)
- On the VOLATILE attribute (Section 10.2.41)
- On the NEW procedure (Section 8.50)

8.4 ARCTAN Function

The ARCTAN function returns a real value that expresses in radians the arctangent of the specified parameter.

ARCTAN(x)

The parameter x can be an INTEGER or REAL type.

8.5 ARGUMENT Function

The ARGUMENT function specifies an argument in a variable-length parameter list that was created using the LIST attribute.

ARGUMENT(parameter-name, n)

The parameter-name argument specifies the name of a parameter declared with the LIST attribute. The parameter n specifies a positive integer value that identifies the argument. The first argument in a list is always 1. An error occurs if the value supplied for n is less than 1, or exceeds the ARGUMENT_LIST_LENGTH parameter (which indicates the total number of arguments).

If the LIST parameter is a value parameter, ARGUMENT indicates the corresponding value in the argument list. If the LIST parameter is a VAR parameter, ARGUMENT is a reference to the corresponding variable in the argument list.

Also, you can use the IADDRESS function with the ARGUMENT function to return the address of a selected argument.

For More Information:

- On variable-length parameter lists (the example in Section 8.6)
- On parameters (Section 6.3)
- On the LIST attribute (Section 10.2.22)
- On the IADDRESS function (Section 8.37)

8.6 ARGUMENT_LIST_LENGTH Function

The ARGUMENT_LIST_LENGTH function returns an integer value representing the number of arguments in a variable-length parameter list that was created using the LIST attribute.

ARGUMENT_LIST_LENGTH(parameter-name)

The parameter-name argument specifies the name of the parameter declared with the LIST attribute.

When creating a variable-length parameter list, you can place the LIST attribute on only the last formal parameter. When you call the routine, you can specify any number of actual parameters, or arguments, that correspond to the last formal parameter declared with LIST. Consider the following:

```
PROGRAM Show_Arg( OUTPUT );
                    {Ax corresponds to any number of char. arguments}
PROCEDURE Variable_Write( Fl : VARYING[len] OF CHAR;
                          Ax : [LIST] CHAR );
   VAR
      i : INTEGER;
   BEGIN
   WRITE( Fl, ', ' );
   {For however many arguments there are:}
   FOR i := 1 TO ARGUMENT_LIST_LENGTH( Ax ) DO
      WRITE( ARGUMENT( Ax, i ) );              {Write an argument}
   WRITELN;
   END;
{In the executable section:}
Variable_Write( ' hello', '*' );  {One argument: Writes ' hello, *'}
Variable_Write( ' hello','s','a','i','l','o','r','!' );
                    {Seven arguments: Writes ' hello, sailor!'}
```

For More Information:

- On parameters (Section 6.3)
- On the LIST attribute (Section 10.2.22)

8.7 BIN Function

The BIN function returns a character-string value that is the binary equivalent of the specified parameter. The return value is compatible with all other string types.

BIN(x[[, length[[, digits]]]])

The parameter x is the expression to be converted. This parameter must have a size that is known at compile time; it cannot be VARYING OF CHAR, a conformant parameter, or a schema type.

Two optional integer parameters specify the length of the resulting string and the minimum number of significant digits to be returned. If you specify a length that is too short to hold the converted value, the resulting string is truncated on the left.

If you omit the optional parameters, the bit width of the converted parameter value determines the string length and the number of significant digits. By default, the number of significant digits is the minimum number of characters necessary to express all the bits of the converted parameter. This default length is one character more than the default number of digits, which causes a leading blank to be included in the resulting string when both parameters are omitted. Consider the following example:

```
TYPE
    Month_Dates = SET OF 0..31;
VAR
    Days_Of_Rain :   Month_Dates;
{In the executable section:}
Days_Of_Rain := [1, 2, 6, 10, 12, 14, 18, 22, 25, 30];
Result := BIN (Days_Of_Rain, 32);
{Returns '01000010010001000101010001000110', 32 characters}
```

The binary representation is from right to left, with the leftmost bit being bit 31 and the rightmost bit being bit 0.

For More Information:

- On character strings (Section 2.6)
- On conformant parameters (Section 6.3.6)

8.8   BITNEXT Function

The BITNEXT function returns an integer value that indicates the number of bits that would be allocated for one component of the specified type in a packed array or if the specified variable appeared as a cell in a packed array.

BITNEXT(x)

The parameter x can be a variable or any type identifier.

Cells in a packed array are affected by any alignment attributes placed on them. By default, they are unaligned if less than 32 bits in size; otherwise, they are byte aligned. Therefore, the size returned includes the actual size

of the type or variable in addition to trailing space required to ensure proper alignment.

The BITNEXT and BITSIZE functions return the same bit size for a given type or variable, except where the components of the packed array are padded to ensure proper alignment.

For More Information:

*   On examples of return values for this function (Table 8–2)
*   On packed arrays (Section 2.4)

8.9 BIT_OFFSET Function

The BIT_OFFSET function returns an INTEGER value that represents the bit position of a field in a record.

BIT_OFFSET(t, f)

The parameter t can be of any record type or variable, and the parameter f can be any field contained in that record.

For More Information:

For information on records, see Section 2.4.2.

8.10 BITSIZE Function

The BITSIZE function returns an integer value that indicates the number of bits that would be allocated for one field of the specified type in a packed record or if the specified variable appeared as a field in a packed record.

BITSIZE(x)

The parameter x can be a variable or any type identifier.

Fields in a packed record are not affected by any alignment attributes placed on subsequent fields. Therefore, the size returned indicates the actual size of the type or variable.

The BITNEXT and BITSIZE functions return the same bit size for a given type or variable, except where the components of the packed array are padded to ensure proper alignment.

For More Information:

- On possible return values for this function (Table 8–2)
- On packed arrays (Section 2.4)

8.11 BYTE_OFFSET Function

The BYTE_OFFSET function returns an integer value that represents the byte position of a field in a record.

BYTE_OFFSET(t, f)

The parameter t can be of any record type or variable, and the parameter f can be any field contained in that record.

For More Information:

For information on records, see Section 2.4.2.

8.12 CARD Function

The CARD function returns an integer value indicating the number of components that are currently elements of the set expression.

CARD(s)

The parameter s must be a set expression.

For More Information:

For information on sets, see Section 2.4.3.

8.13 CHR Function

The CHR function returns a char value whose ordinal value in the ASCII character set is the parameter, provided such a character exists.

CHR(x)

The parameter x must be of type INTEGER or UNSIGNED and have a value from 0 to 255.

For More Information:

For information on the ASCII character set, see Appendix A.

8.14 CLEAR_INTERLOCKED Function

The CLEAR_INTERLOCKED function assigns the value FALSE to the
parameter and returns the original Boolean value of the parameter.

CLEAR_INTERLOCKED(b)

The parameter b must be a variable of type BOOLEAN. The variable does
not have to be aligned; therefore, it can be a field of a packed record.

8.15 CLOCK Function

The CLOCK function returns an integer value indicating the amount of
central processor time (in milliseconds) used by the current process. This
function does not have a parameter list. The result of CLOCK includes the
amount of central processor time allocated to all previously executed images.

8.16 COS Function

The COS function returns a real value that represents the cosine of the
specified parameter.

COS(x)

The parameter x can be an INTEGER or REAL type, and is expressed in
radians.

8.17 CREATE_DIRECTORY Procedure

The CREATE_DIRECTORY procedure creates a new directory or
subdirectory.

CREATE_DIRECTORY(file-name [[, error-return]])

The file-name parameter must be a directory name, and optionally can
contain a device name. The error return parameter is optional, and will
return an error recovery code if specified.

For More Information:

- On device, directory, and file specifications (*VAX Pascal Reference Supplement for VMS Systems*)

- On error recovery codes (*VAX Pascal Reference Supplement for VMS Systems*)

8.18 DATE and TIME Functions

The DATE and TIME functions provide a standard way of returning a character-string value that indicates the calender date and time. The return value is compatible with all string types.

```
DATE( t )
TIME( t )
```

The parameter t is a variable of the predeclared type TIMESTAMP. You can either call the GETTIMESTAMP procedure to initialize parameter t before you pass t to either DATE or TIME, or you can construct your own TIMESTAMP object.

The size of the function's return value depends on the string length that is normally returned by your system for either date or time data. Consider the following:

```
VAR
    Time_Var : TIMESTAMP;
    The_Time, The_Date : STRING( 23 );
{In the executable section:}
GETTIMESTAMP( Time_Var );
The_Date := DATE( Time_Var );
The_Time := TIME( Time_Var );
WRITELN( The_Date, The_Time );   {Writes: 1-FEB-1989 14:20:25.98 }
```

For More Information:

For information on the GETTIMESTAMP predeclared procedure, see Section 8.34.

8.19 DATE and TIME Procedures

The DATE and TIME procedures write the date and the time to their parameters.

DATE(str)
TIME(str)

The parameter str must be of type PACKED ARRAY[1..11] OF CHAR. After execution of the procedure, the parameter str contains either the date or the time. If the day of the month is a 1-digit number, the leading zero does not appear in the result; that is, a space appears before the date string. The time is returned in 24-hour format.

For More Information:

For information on standard ways to obtain the date and the time, see Section 8.18.

8.20 DBLE Function

The DBLE function converts the parameter and returns its DOUBLE equivalent.

DBLE(x)

The parameter x must be of an arithmetic type. The value of x must not be too large to be represented by a double-precision number.

For More Information:

For information on precision and support for the DOUBLE data type, see the *VAX Pascal Reference Supplement for VMS Systems*.

8.21 DEC Function

The DEC function returns a character-string value that is the decimal equivalent of the specified parameter. The return value is compatible with all other string types.

DEC(x[[, length[[, digits]]]])

The parameter x is the expression to be converted. DEC can take a parameter of any type except VARYING OF CHAR, conformant parameters, or schema types. DEC requires x to be 32 bits or less in length.

Two optional integer parameters specify the length of the resulting string and the minimum number of significant digits to be returned. If you specify a length that is too short to hold the converted value, the resulting string is truncated on the left. If you do not specify values for the optional parameters, a default length and a default minimum number of significant digits is used. The defaults are 11 characters for the length and 10 characters for the minimum number of digits. Because the default length is 1 greater than the number of significant digits, positive numbers will be preceded by a blank and negative numbers will be preceded by a minus sign. Consider the following:

```
VAR
    Account : INTEGER;
{In the executable section:}
Account := 16#F;
WRITELN( DEC( Account, 8, 7 ) );
```

The value of the integer variable Account is converted to its decimal equivalent (16) and, in this example, printed in eight columns: seven digits, and one leading blank.

For More Information:

- On character strings (Section 2.6)
- On conformant parameters (Section 6.3.6)

8.22 DELETE_FILE Procedure

The DELETE_FILE procedure deletes one or more files.

DELETE_FILE(file-name [[, error-return]])

The file-name specification can contain an explicit device and directory name, plus it must contain a file name, a file type or extension, and a version number. If you omit either the directory or device name, VAX Pascal uses the directory you are working in at the time of program execution. The error return parameter returns an error recovery code if specified.

For More Information:

- On device, directory, file, type or extension, and version number specifications (*VAX Pascal Reference Supplement for VMS Systems*)
- On error recovery codes (*VAX Pascal Reference Supplement for VMS Systems*)

8.23 DISPOSE Procedure

The DISPOSE procedure deallocates memory for a dynamic variable.

DISPOSE(p [[, t1,...,tn]])

The parameter p is a pointer variable. The t parameters are constant expressions that match the corresponding t parameter used in the call to the NEW procedure that allocated the memory. If you use t parameters in a call to NEW, you must specify the same t parameters in the call to DISPOSE. If you allocated memory using d parameters, just specify the pointer variable to the corresponding DISPOSE call.

The DISPOSE procedure deallocates the object to which the pointer variable points. You cannot call DISPOSE more than once for the same dynamic variable. Consider the following:

```
DISPOSE( Ptr );   {Ptr^ is distroyed; Ptr becomes undefined}
```

For More Information:

* On the pointer data type (Section 2.3)
* On the NEW procedure (Section 8.50)

8.24 EQ Function

The EQ function returns a Boolean value that specifies if the parameters are equal according to the ASCII values of the strings' characters.

EQ(str1, str2)

The parameters str1 and str2 must be character-string expressions. If the EQ function detects unequal string lengths, it stops comparison and returns FALSE. Consider the following:

```
VAR
    Match : BOOLEAN;
{In the executable section:}
Match := EQ( 'exit   ', 'exit' );   {Returns FALSE; unequal lengths}
Match := EQ( 'exit', 'exit' );      {Returns TRUE}
```

For More Information:

For information on string data types, see Section 2.6.

8.25 ESTABLISH Procedure

The ESTABLISH procedure specifies a condition handler that executes if your program generates operating-system events.

ESTABLISH(function-identifier)

The function-identifier parameter must be the name of a function that has the ASYNCHRONOUS attribute. The function passed to ESTABLISH must have two formal array parameters.

For More Information:

*   On the ASYNCHRONOUS attribute (Section 10.2.2)
*   On error and report processing (*VAX Pascal Reference Supplement for VMS Systems*)

8.26 EXP Function

The EXP function returns a real value that represents the exponent of the specified parameter (it represents e^x).

EXP(x)

The parameter x can be an INTEGER or REAL type.

8.27 EXPO Function

The EXPO function returns the integer exponent of the floating-point representation of the parameter.

EXPO(x)

The parameter x can be of any real type.

For More Information:

For information on precision and support for real numbers, see the *VAX Pascal Reference Supplement for VMS Systems*.

3.28 FIND_FIRST_BIT_CLEAR Function

The FIND_FIRST_BIT_CLEAR function locates the first bit in a Boolean array whose value is 0 and returns an integer value that specifies the index into the array.

FIND_FIRST_BIT_CLEAR(vector [[, start-index]])

The vector parameter is a variable of type PACKED ARRAY OF BOOLEAN with an INTEGER index type. The optional start-index parameter must be an INTEGER expression that indexes the element at the point at which the search starts. The starting index must be greater than or equal to the vector's lower bound, and less than or equal to 1 plus the vector's upper bound; otherwise, a range violation occurs. If omitted, the starting index defaults to the vector's first element.

The FIND_FIRST_BIT_CLEAR function returns a value indexing the first element containing the value 0. If no bit is 0, the result is 1 plus the vector's upper bound. If the vector or the indexed part of the vector has a size of 0, the result is start-index.

8.29 FIND_FIRST_BIT_SET Function

The FIND_FIRST_BIT_SET function locates the first bit in a Boolean array whose value is 1 and returns an integer value that specifies the index into the array.

FIND_FIRST_BIT_SET(vector [[, start-index]])

The vector parameter is a variable of type PACKED ARRAY OF BOOLEAN with an INTEGER index type. The optional start-index parameter must be an expression of an INTEGER type that indexes the element at the point at which the search starts. The starting index must be greater than or equal to the vector's lower bound, and less than or equal to 1 plus the vector's upper bound; otherwise, a range violation occurs. If omitted, the starting index defaults to the vector's first element.

The FIND_FIRST_BIT_SET function returns an integer value indexing the first element containing the value 1. If no bit is 1, the result is 1 plus the vector's upper bound. If the vector or the indexed part of the vector has a size of 0, the result is start-index. Consider the following.

```
VAR
    Boo : PACKED ARRAY [0..31] OF BOOLEAN;
{In the executable section:}
Boo::INTEGER := 128;
WRITELN( FIND_FIRST_BIT_SET( BOO ) );
```

For More Information:

For information on the type-cast operator (::), see Section 4.2.6.

8.30 FIND_MEMBER Function

The FIND_MEMBER function locates the first character in a string that is a member of a specified set and returns an integer value indicating the position of the character in the string; the function returns 0 if the characters in the string were not members of the set.

FIND_MEMBER(string, char-set)

The string parameter is a string value, and char-set is a value of type SET OF CHAR.

For More Information:

*   On string types (Section 2.6)
*   On sets (Section 2.4.3)

8.31 FIND_NONMEMBER Function

The FIND_NONMEMBER function locates the first character in a string that is not a member of a specified set and returns an integer value indicating the position of the character in the string; the function returns 0 if the characters in the string were all members of the set.

FIND_NONMEMBER(string, char-set)

The string parameter is a string value, and char-set is a value of type SET OF CHAR.

For More Information:

*   On string types (Section 2.6)
*   On sets (Section 2.4.3)

8.32 GE Function

The GE function returns a Boolean value that specifies if the first parameter is greater than or equal to the second parameter, according to the ASCII values of the strings' characters.

GE(str1, str2)

The parameters str1 and str2 must be character-string expressions. VAX Pascal does not pad shorter strings with blanks. Consider the following:

```
VAR
    Match : BOOLEAN;
    Test  : STRING(8) VALUE 'ENTRANCE';
{In the executable section:}
Match := GT( 'exit', 'exit' );   {Returns TRUE}
Match := GT( Test, 'EXIT' );     {'N' less-than 'X': Returns FALSE}
```

For More Information:

For information on string data types, see Section 2.6.

8.33 GT Function

The GT function returns a BOOLEAN value that specifies if the first parameter is greater than the second parameter, according to the ASCII values of the strings' characters.

GT(str1, str2)

The parameters str1 and str2 must be character-string expressions. VAX Pascal does not pad shorter strings with blanks. Consider the following:

```
VAR
    Match : BOOLEAN;
    Test  : STRING( 8 ) VALUE 'ENTRANCE';
{In the executable section:}
Match := GT( 'exit', 'exit' );   {Returns FALSE}
Match := GT( Test, 'EXIT' );     {'N' less-than 'X': Returns FALSE}
```

For More Information:

For information on string data types, see Section 2.6.

8.34 GETTIMESTAMP Procedure

The GETTIMESTAMP procedure initializes its parameter for use with the DATE and TIME functions.

GETTIMESTAMP(t [[, str]])

The parameter t is a variable of the TIMESTAMP type, which is a predeclared record type. The TIMESTAMP data type is as follows:

```
TIMESTAMP = PACKED RECORD
    DATEVALID, TIMEVALID   : BOOLEAN;
    YEAR                   : INTEGER;
    MONTH                  : 1..12;
    DAY                    : 1..31;
    HOUR                   : 0..23;
    MINUTE                 : 0..59;
    SECOND                 : 0..59;
    HUNDREDTH              : 0..99;
        {64-bit VMS binary time:}
    BINARY_TIME            : [QUAD] RECORD L1,L2:INTEGER END;
    DAY_OF_WEEK            : 1..7;    {1 is Monday and 7 is Sunday}
    END;
```

The parameter str is a string type that represents a date or both a date and time. The following rules apply to the specification of the str parameter:

- If you do not specify the parameter str, the GETTIMESTAMP procedure initializes the variable to be the date and time at execution of the procedure.

- If you specify an invalid date, the GETTIMESTAMP procedure sets the date to be January 1, 1. If you omit the date, this procedure uses the current date. If you specify an invalid time or if you omit the time, it sets the time to be midnight.

Consider the following:

```
VAR
     Time_Var : TIMESTAMP;
     The_Time, The_Date : STRING( 23 );
{In the executable section:}
GETTIMESTAMP( Time_Var );    {Get current date and time}
GETTIMESTAMP( Time_Var, 'TOMORROW' ); {Midnight tomorrow}
GETTIMESTAMP( Time_Var, '22-Nov-1988:12:30:15.15');
GETTIMESTAMP( Time_Var, '22-Nov-1988' ); {Midnight at that date}
GETTIMESTAMP( Time_Var, '41-Nov-1988:999:999:999.99' );
                        {Midnight on January 1, 1}
```

For More Information:

For information on the DATE and TIME functions, see Section 8.18.

8.35 HALT Procedure

The HALT procedure uses operating system resources to stop execution of your program unless you have written a condition handler (using the ESTABLISH procedure) that enables continued execution.

For More Information:

- On the ESTABLISH procedure (Section 8.25)
- On environment-specific behavior of HALT (*VAX Pascal Reference Supplement for VMS Systems*)

8.36 HEX Function

The HEX function returns a character-string value that is the hexadecimal equivalent of the specified parameter. The return value is compatible with all other string types.

HEX(x[[, length[[, digits]]]])

The parameter x is the expression to be converted. This parameter must have a size that is known at compile time; it cannot be VARYING OF CHAR, a conformant parameter, or a schema type.

Two optional integer parameters specify the length of the resulting string and the minimum number of significant digits to be returned. If you specify a length that is too short to hold the converted value, the resulting string is truncated on the left. If you do not specify values for the optional parameters, a default length and a default number of significant digits is used. By default, the number of significant digits is the minimum number of characters necessary to express all the bits of the converted parameter. This default length is one character more than the default number of digits, which causes a leading blank to be included in the resulting string when both parameters are omitted. Consider the following.

```
VAR
    p : ^Rec;
{In the executable section:}
Digits := 8;
NEW( p );
Result := HEX( p, 10, Digits );
```

In this example, the HEX function returns a string of 10 characters
containing the hexadecimal equivalent of the value of the pointer variable p.
The string has eight significant digits, as specified by the value of the actual
parameter Digits.

8.37 IADDRESS Function

The IADDRESS function returns an integer value that refers to the address
of either a VOLATILE parameter or a routine, and does not generate
compile-time warnings (as does the ADDRESS function). The IADDRESS
function is commonly used for constructing arguments for system services of
the VMS operating system.

IADDRESS(x)

The parameter x can be of any type except a component of a packed
structured type or a routine name.

NOTE

The VAX Pascal compiler automatically assumes that all pointers
refer either to dynamic variables allocated by the NEW procedure
or to variables that have the VOLATILE attribute; therefore, you
should use utmost caution when using the IADDRESS function.
This function does not generate compile-time warnings.

Consider the following example:

```
VAR
    Real_Addr : INTEGER;
    Real_Var  : [VOLATILE] REAL;
{In the executable section:}
Real_Addr := IADDRESS( Real_Var );   {Returns address of Real_Var}
WRITELN( 'The address of Real_Var is', Real_Addr );
```

For More Information:

* On the VOLATILE attribute (Section 10.2.41)

* On the ADDRESS function (Section 8.3)

* On the NEW procedure (Section 8.50)

8.38 INDEX Function

The INDEX function searches a string for a specified substring and returns an integer value that either indicates the location of the substring or the status of the search.

INDEX(string, substring)

INDEX requires two character-string expressions as parameters: a string to be searched and a substring to be found.

The search ends as soon as the first occurrence of the substring is located. If the substring is found, INDEX returns the string component that contains the first letter of the substring. If the substring is not found, INDEX returns the value 0. If the substring is an empty string, INDEX returns the value 1. If the string to be searched is an empty string, INDEX returns the value 0 unless the substring is also empty; in which case, INDEX returns the value 1. Consider the following example:

```
The_String  := 'The Pilgrims landed at Plymouth Rock';
Substring   := 'Plymouth Rock';
Position    := INDEX( The_String, Substring ); {Returns 24}
Substring   := 'Mayflower';
Position    := INDEX( The_String, Substring ); {Returns 0}
```

For More Information:

For information on character strings, see Section 2.6.

8.39 INT Function

The INT function converts the parameter and returns its INTEGER equivalent.

INT(x)

The parameter x must be of an ordinal type.

No error results if x is of type UNSIGNED and has a value greater than MAXINT. In that case, the value of x is converted to its equivalent as a 32-bit integer by subtracting 2^{32} from it. For example, INT(3604928157) returns the value −690,039,139, which is the negative integer with the same 32-bit representation as the unsigned integer value 3,604,928,157.

For More Information:

For information on range and support for the UNSIGNED data type, see the *VAX Pascal Reference Supplement for VMS Systems.*

8.40 LE Function

The LE function returns a Boolean value that specifies if the first parameter is less than or equal to the second parameter, according to the ASCII values of the strings' characters.

LE(str1, str2)

The parameters str1 and str2 must be character-string expressions. VAX Pascal does not pad shorter strings with blanks.

The expression LE(Str1, Str2) is equivalent to the following:

```
( LENGTH( Str1 ) < LENGTH( Str2 ) )   OR   ( Str1 <= Str2 )
```

Consider the following example:

```
VAR
    Match : BOOLEAN;
    Test  : STRING( 8 ) VALUE 'ENTRANCE';
{In the executable section:}
Match := LE( 'exit', 'exit' );    {Returns TRUE}
Match := LE( Test, 'EXIT' );      {'N' less-than 'X': Returns TRUE}
```

For More Information:

For information on string data types, see Section 2.6.

8.41 LENGTH Function

The LENGTH function returns an integer value that is the length of a specified string expression.

LENGTH(str)

LENGTH requires a character-string expression as a parameter.

For More Information:

For information on character strings, see Section 2.6.

8.42 LN Function

The LN function returns a real value that represents the natural logarithm of the specified parameter.

LN(x)

The parameter x can be an INTEGER or REAL type. The value of x must be greater than zero.

8.43 LOWER Function

The LOWER function returns the lower bound for ordinal types, SET base types, and array indexes.

LOWER(x [[, n]])

The parameter x is a type identifier or variable of an ordinal, SET, or ARRAY type. The parameter n is an integer constant that denotes a dimension of x, if x is an array. If x is an array and if you omit the parameter n, VAX Pascal uses the default value 1. If x is an array, LOWER returns the lower bound of the nth dimension of x. If x is an ordinal type, LOWER returns the lower bound or smallest value. If x is a SET, LOWER returns the lower bound of the SET base type.

For More Information:

For examples of the LOWER function, see Section 8.83.

8.44 LT Function

The LT function returns a Boolean value that specifies if the first parameter is less than the second parameter, according to the ASCII values of the strings' characters.

LT(str1, str2)

The parameters str1 and str2 must be character-string expressions. VAX Pascal does not pad shorter strings with blanks. Consider the following:

```
VAR
    Match : BOOLEAN;
    Test  : STRING( 8 ) VALUE 'ENTRANCE';
{In the executable section:}
Match := LT( 'exit', 'exit' );    {Returns FALSE}
Match := LE( Test, 'EXIT' );      {'N' less-than 'X': Returns TRUE}
```

For More Information:

For information on string data types, see Section 2.6.

8.45 MAX Function

The MAX function returns a value (the same type as that of the parameters) that is the maximum value of a specified list of parameters.

MAX(x1,...,xn)

The parameters can be any arithmetic type, but must all be of the same type.

8.46 MFPR Function

The MFPR function returns an unsigned value that is the value of a VAX internal processor register.

MFPR(ipr-register-expression)

The ipr-register-expression parameter is an expression compatible with the UNSIGNED type.

For More Information:

For information on running in kernel mode or on using the MFPR function, see the *VAX Pascal Reference Supplement for VMS Systems*.

8.47 MIN Function

The MIN function returns a value (of the same type as that of the parameters) that is the minimum value of a specified list of parameters.

MIN(x1,...,xn)

The parameters can be any arithmetic type, but must all be of the same type.

8.48 MTPR Procedure

The MTPR procedure assigns a value into a VAX internal processor register.

MTPR(ipr-register-expression, source-expression);

The ipr-register-expression and source-expression parameters are expressions compatible with the unsigned type. VAX Pascal stores the value specified by source-expression into the internal processor register specified by the ipr-register-expression.

For More Information:

For information on running in kernel mode or on using the MTPR procedure, see the *VAX Pascal Reference Supplement for VMS Systems.*

8.49 NE Function

The NE function returns a Boolean value that specifies if the parameters are not equal according to the ASCII values of the strings' characters.

NE(str1, str2)

The parameters str1 and str2 must be character-string expressions. VAX Pascal does not pad shorter strings with blanks. Consider the following:

```
VAR
    Match : BOOLEAN;
{In the executable section:}
Match := NE( 'exit   ', 'exit' );   {Returns TRUE}
Match := NE( 'exit', 'exit' );      {Returns FALSE}
```

For More Information:

For information on string data types, see Section 2.6.

8.50 NEW Procedure

The NEW procedure allocates memory for the dynamic variable to which a pointer variable refers. The value of the newly allocated variable is set to the initial value of the base type if defined; otherwise, the value of the variable is undefined.

NEW(p [[, $\left\{ \begin{array}{l} t1,...,tn \\ d1,...,dn \end{array} \right\}$]])

The parameter p is a pointer variable.

The parameters t1,...,tn are constant expressions of an ordinal type that represent nested tag-field values, where t1 is the outermost variant.

If the object of the pointer is a non-schema record type with variants, then you have two ways of allocating memory. If you do not specify t parameters, VAX Pascal allocates enough memory to hold any of the variants of the record. If you do specify t parameters, then VAX Pascal allocates enough memory to hold only the variant or variants that you specify.

Since the t parameters cause VAX Pascal to allocate memory for the variant alone and not for the whole record, you cannot assign or evaluate the record as a whole; you can assign and evaluate only the individual fields. Also, a call to NEW sets the tag fields of a variant record.

The parameters d1,...,dn are compile-time or run-time ordinal values that must be the same type as the formal discriminants of the object.

If the object of the pointer is of an undiscriminated schema type, you must specify a d parameter for each of the formal discriminants of the schema type. The d parameters discriminate the schema type in much the same way as actual discriminants in a discriminated schema. VAX Pascal bases the size of the allocation on the value of the d parameters.

If the object is a schema record type, then you must use d parameters; you cannot use t parameters or a combination of the syntaxes. If the schema record type contains a variant (which depends on one of the formal discriminants) then the d parameter discriminates the schema, determines the variant, and allows VAX Pascal to compute the necessary size of the allocation.

NOTE

If you specify t parameters to the NEW procedure, you must specify the same t parameters to the DISPOSE procedure that deallocates memory for the corresponding variable.

Consider the following examples:

```
TYPE
    Meat_Type = ( Fish, Fowl, Beef );
    Beef_Portion = ( Oz_10, Oz_16, Oz_32 );
    Var_Record = RECORD
        CASE Entree : Meat_Type OF
            Fish : ( Fish_Type : ( Salmon, Cod, Perch, Trout );
                     Lemon : BOOLEAN );
            Fowl : ( Fowl_Type : ( Chicken, Duck, Goose );
                     Sauce : ( Orange, Cherry, Raison ));
            Beef : ( Beef_Type : ( Steak, Roast, Prime_Rib );
                CASE size : Beef_Portion OF
                        Oz_10, Oz_16 : ( Beef_Veg : ( Pea, Mixed ));
                Oz_32        : ( Stomach_Cure :
                                        Bicarb, Antacid, None )));
        END;
    The_Schema( Upper_Bound : INTEGER )
                = ARRAY [1..Upper_Bound] OF INTEGER;
VAR
    To_Int         : ^INTEGER;
    To_Var_Record : ^Var_Record;
    To_Schema      : ^The_Schema;
    Bound          : INTEGER VALUE 32;
```

```
{In the executable section:}
NEW( To_Int );   {Memory for To_Int^ allocated but not initialized}

NEW( To_Var_Record, Fish ); {Memory allocated only for Fish variant}
DISPOSE( To_Var_Record, Fish ); {Specify Fish to DISPOSE}
NEW( To_Var_Record, Beef, Oz_32 ); {Allocates more memory this time}
DISPOSE(To_Var_Record, Beef, Oz_32 );

NEW( To_Schema, Bound ); {Allocation for undisc. schema object}
DISPOSE( To_Schema );
```

For More Information:

- On variant records (Section 2.4.2.1)

- On schema types (Section 2.5)

- On the DISPOSE procedure (Section 8.23)

.51 NEXT Function

The NEXT function returns an integer value that indicates the number of bytes that would be allocated for one component of the specified type in an unpacked array or if the specified variable appeared as the cell in an unpacked array.

```
NEXT( x )
```

The parameter x can be a type identifier or variable.

Cells in an unpacked array are affected by alignment attributes and, by default, are byte aligned. Therefore, the size returned includes the actual size of the type or variable, in addition to trailing space required to ensure proper alignment.

If a variable that is not allocated to an integral number of bytes is passed to NEXT, the number of bits are rounded down to the nearest byte and then the number of bytes are returned.

For More Information:

- On examples of return values for this function (Section 8.66)

- On arrays (Section 2.4.1)

8.52 OCT Function

The OCT function returns a character-string value that is the octal equivalent of the specified parameter. The return value is compatible with all other string types.

```
OCT( x[[, length[[, digits]] ]] )
```

The parameter x is the expression to be converted. This parameter must have a size that is known at compile time; it cannot be VARYING OF CHAR, a conformant parameter, or a schema type.

Two optional integer parameters specify the length of the resulting string and the minimum number of significant digits to be returned. If you specify a length that is too short to hold the converted value, the resulting string is truncated on the left. By default, the number of significant digits is the minimum number of characters necessary to express all the bits of the converted parameter. This default length is one character more than the default number of digits, which causes a leading blank to be included in the resulting string when both parameters are omitted. Consider the following:

```
Int_Var := 427;
Result  := OCT( Int_Var, 10, 3 );  {Returns '        653'}
```

For More Information:

For information on character strings, see Section 2.6.

8.53 ODD Function

The ODD function returns a Boolean value that indicates if the parameter is odd.

```
ODD( x )
```

The parameter x must be of type INTEGER or UNSIGNED. The function returns TRUE if the value of x is odd and FALSE if the value of x is even.

8.54 ORD Function

The ORD function returns an integer value that is the position of the parameter in the ordered sequence of values of the parameter's type.

```
ORD( x )
```

The parameter x must be of an ordinal type. Note that the ordinal value of an INTEGER object is the integer itself. If x is of type UNSIGNED, its value must not be greater than MAXINT.

8.55 PACK Procedure

The PACK procedure copies components of an unpacked array variable to a packed array variable.

PACK(a, i, z)

The parameter a is an unpacked array. The parameter i is a value to indicate the starting value of the index of a. The parameter z is a packed array of the same component type as a.

The number of components in parameter a must be greater than or equal to the number of components in z. The PACK procedure assigns the components of a, starting with a[i], to the array z, starting with z[low-bound], until all the components in z are filled.

In general, when specifying i, keep in mind that the upper bound of a (that is, n) must be greater than or equal to i + v−u, where v is the upper bound of z and u is the lower bound of z. That is, ORD(n) must be greater than or equal to ORD(i) + ORD(v) − ORD(u). Consider the following:

```
VAR
   a : ARRAY[1..25] OF 0..15;
   p : PACKED ARRAY[1..20] OF 0..15;
   i : INTEGER;
{In the executable section:}
FOR i := 1 TO 20 DO
   READ ( a[i] );
PACK( a, 5, p );   {a[1] through a[4] are not used}
PACK( a, 1, p );   {a[21] through a[25] are not used}
```

For More Information:

For information on arrays and packed arrays, see Section 2.4.

8.56 PAD Function

The PAD function returns a character-string value, of the specified size, that contains padded fill characters. The return value is compatible with all other string types.

PAD(str, fill, size)

The parameter str is a character-string value to be padded; the parameter fill is a value of type CHAR to be used as the fill character; and, the parameter size is an integer value indicating the size of the final string.

This string is composed of the original string followed by the fill character, which is repeated as many times as is necessary to extend the string to its specified size. The final size must be greater than or equal to the length of the string to be padded.

For More Information:

For information on character strings, see Section 2.6.

8.57 PRED Function

The PRED function returns the value preceding the parameter according to the parameter's data type.

PRED(x)

The parameter x can be of any ordinal type; however, there must be a predecessor value for x in the type.

8.58 PRESENT Function

The PRESENT function returns a Boolean value that indicates whether the actual argument list of a routine contains an argument that corresponds to a formal parameter. (The PRESENT function is usually used to supply a default value or to take a default action when the argument for a parameter is omitted.)

PRESENT(parameter-name)

The parameter-name argument is the name of a formal parameter with the TRUNCATE attribute. The parameter-name must be the name of a formal parameter of the function from which PRESENT is called, or from a subroutine of that function. The function result indicates whether the argument list of the containing routine specifies an actual argument corresponding to an optional parameter.

Parameters that do not have the TRUNCATE attribute and also do not follow a parameter with the TRUNCATE attribute in the formal parameter list, are allowed; in their case, the PRESENT function always returns TRUE.

Default parameters are considered to be present in the argument list, and the PRESENT function returns TRUE when passed the name of a parameter with a default value.

For More Information:

* On examples using PRESENT and TRUNCATE (Section 10.2.36)
* On parameters (Section 6.3)

8.59 QUAD Function

The QUAD function converts the parameter and returns its QUADRUPLE equivalent.

QUAD(x)

The parameter x must be of an arithmetic type.

For More Information:

For information on precision and support for the QUADRUPLE data type, see the *VAX Pascal Reference Supplement for VMS Systems*.

8.60 READV Procedure

The READV procedure reads characters from a character-string expression and assigns them to parameters in the READV call. The behavior of READV is analogous to that of READLN; the character string is analogous to a one-line file.

READV(str, {variable-identifier[[: radix-specifier]]},... [[, ERROR := error-recovery]])

The parameter str is the string to be read. The parameter variable-identifier is the variable to be assigned a value from str. The parameter radix-specifier can be BIN, OCT, or HEX. You can read a variable of any type by using a radix specifier except a type that contains a file component. The error-recovery parameter indicates the action to be taken in case of an error. (An error occurs at run time if values have not been assigned to all the parameters listed in the READV procedure call before the end of the character string is reached.)

```
TYPE
    Color = ( Yellow, Red, Blue );
VAR
    Paint, Paint2 : Color;
    Month : VARYING[5] OF CHAR;
    Real_Var : REAL;
    Read_String : VARYING[17] OF CHAR;

{In the executable section:}
Read_String := 'Red July 26.33805';

READV( Read_String, Paint, Month, Real_Var );
{Paint contains Red, Month contains 'July', and Real_Var contains 26.33805}

READV( Read_String, Paint, Month, Real_Var, Paint2 );
{Error: end of string reached after assigning to Real_Var}

READV( Read_String, Paint, Month ); {Legal: '26.33805' is not used}
READV( Read_String, Real_Var, Paint, Month ); {Error: Red is not REAL}
```

For More Information:

- On input and output (Chapter 9)

- On character strings (Section 2.6)

- On error recovery codes (*VAX Pascal Reference Supplement for VMS Systems*)

8.61 RENAME_FILE Procedure

The RENAME_FILE procedure renames a file.

RENAME_FILE(old-file-name, new-file-name [[, error-return]])

The parameter old-file-name specifies the names of one or more files whose specifications are to be changed. New-file-name provides the new file specification to be applied. The error-return parameter contains an error recovery code if specified.

For More Information:

For information on file specifications or on error processing, see the *VAX Pascal Reference Supplement for VMS Systems*.

8.62 REVERT Procedure

The REVERT procedure cancels a condition handler activated by the ESTABLISH procedure. This procedure does not have a parameter list.

For More Information:

For information on error processing, see the *VAX Pascal Reference Supplement for VMS Systems*.

8.63 ROUND Function

The ROUND function converts the value of the parameter by rounding the fractional part of the value, and returns its integer equivalent.

ROUND(x)

The parameter x must be of type REAL, SINGLE, DOUBLE, or QUADRUPLE. The value of x must not be too large to be represented by an integer.

8.64 SET_INTERLOCKED Function

The SET_INTERLOCKED function assigns the value TRUE to the parameter and returns its original Boolean value.

SET_INTERLOCKED(b)

The parameter b must be a variable of type BOOLEAN. The variable does not have to be aligned; therefore, it can be a field of a packed record.

8.65 SIN Function

The SIN function returns a real value that represents the sine of the specified parameter.

SIN(x)

The parameter x can be an INTEGER or REAL type, and is expressed in radians.

8.66 SIZE Function

The SIZE function returns an integer value that indicates the possible or actual number of bytes that are allocated for a specified data type or variable.

SIZE(x[[,t1,...,tn]])

The parameter x can be a type identifier or variable. If x is a type identifier, then SIZE returns an integer value which indicates the number of bytes that would be allocated for a variable or record field of type x. If x is a variable, then SIZE returns an integer value that indicates the number of bytes that are allocated for that variable.

In the case where the parameter x is a variant record variable or variant type identifier, SIZE returns an integer value that indicates the number of bytes that are allocated (for a variant record variable) or would be allocated (for a variant type identifier) for both the fixed portion of the record and the largest variant. In addition, you can supply additional parameters t1 through tn that correspond to the case labels of the record. The SIZE routine returns an integer value that indicates the number of bytes that would be allocated by the NEW procedure for a dynamic variable of the specified variant.

If a variable that is not allocated to an integral number of bytes is passed to SIZE, the number of bits will be rounded up to the nearest byte and then the number of bytes will be returned.

Table 8–2 presents values returned by the alignment routines if the routines accepted objects of the specified data types.

Table 8–2: Return Values of Alignment Predeclared Routines

Type or Variable	Size in Bits		Size in Bytes	
	BITNEXT	BITSIZE	NEXT	SIZE
[BIT(1)] BOOLEAN	1^1	1	1^2	1^3
0..25 (subrange)	5^1	5	4^4	4^4
[BYTE] 0..255 (byte)	8	8	1	1
[BYTE, ALIGNED(2)] 0..225	32^5	8	4^5	1

[1]By default, the variable is unaligned in a packed context.
[2]By default, the variable is byte aligned in an unpacked context.
[3]SIZE rounds up to the nearest byte for the bit-sized objects.
[4]Subranges assume the size of their base types in an unpacked context.
[5]Extra space is needed to fulfill alignment requirements.

Table 8–2 (Cont.):   Return Values of Alignment Predeclared Routines

Type or Variable	Size in Bits		Size in Bytes	
	BITNEXT	BITSIZE	NEXT	SIZE
First element of: PACKED ARRAY [1..10] OF 0..25	5	5	0^6	1^3
PACKED ARRAY [1..10] OF 0..25	56^7	56^7	7	7

[3]SIZE rounds up to the nearest byte for the bit-sized objects.
[6]NEXT rounds down to the nearest byte for bit-sized objects.
[7]Items larger than 32 bits must be allocated in an integral number of bytes.

For More Information:

For information on data types, see Chapter 2.

8.67 SNGL Function

The SNGL function converts the parameter and returns its real equivalent.

SNGL(x)

The parameter x must be of an arithmetic type. The value of x must not be too large to be represented by a single-precision number.

8.68 SQR Function

The SQR function returns a value (of the same type of the parameter) that represents the square of the specified parameter.

SQR(x)

The parameter x can be of any arithmetic type.

8.69 SQRT Function

The SQRT function returns a real value that represents the square root of the specified parameter.

SQRT(x)

The parameter x can be of an INTEGER, UNSIGNED, or REAL type. If the
value of x is less than zero, an error occurs.

8.70 STATUSV Function

The STATUSV function returns an integer value that specifies the status of
the last READV or WRITEV procedure completed. STATUSV does not have
any parameters.

If you have an asynchronous system trap (AST) routine condition handler
written in your program that uses READV and WRITEV, the call of
STATUSV in your main program may not return the results you expected if
an AST occurred between the READV/WRITEV procedure and the STATUSV
procedure. Consider the following:

```
VAR
    Vary_Src   : VARYING [20] OF CHAR;
    Int_Result : INTEGER;
{In the executable section:}
Vary_Src := '255';
READV( Vary_Src, Int_Result, ERROR := CONTINUE );
IF STATUSV <> 0 THEN   {0 means READV executed successfully}
    WRITELN( 'Error in READV' );
```

For More Information:

* On the READV procedure (Section 8.60)
* On the WRITEV procedure (Section 8.87)
* On character strings (Section 2.6)

8.71 SUBSTR Function

The SUBSTR function returns a substring (from a string specified as a
parameter) that is of the specified starting point and length. The return
value is compatible with all other string types.

SUBSTR(str, start, length)

The parameter str is a character string value; the parameter start is an
integer value that indicates the starting position of the substring. The
parameter length is an integer value that indicates the length of the
substring.

The following rules apply to the use of the SUBSTR function:

* The value of the starting position must be greater than 0.

- The value of the length must be greater than or equal to 0.
- There must be enough characters following the starting position to construct a substring of the specified length.

Consider the following:

```
Original_String := 'This is the original string';
Start_Position := 13;
Substring_Length := 15;
New_String := SUBSTR( Original_String, Start_Position,
                      Substring_Length );
{New_String contains 'original string'}
```

For More Information:

For information on character strings, see Section 2.6.

8.72 SUCC Function

The SUCC function returns the value that succeeds the parameter according to the parameter's data type.

SUCC(x)

The parameter x can be of any ordinal type; however, there must be a successor value for x in the type.

8.73 TIME Function

See Section 8.18.

8.74 TIME Procedure

See Section 8.19.

8.75 TRUNC Function

The TRUNC function converts the value of the parameter by truncating the fractional part of the value and returns its integer equivalent.

TRUNC(x)

The parameter x must be of type REAL, SINGLE, DOUBLE, or QUADRUPLE. The value of x must not be too large to be represented by an integer.

8.76 UAND Function

The UAND function returns an unsigned value that represents a binary logical AND operation on each corresponding pair of bits of the specified parameters.

UAND(u1, u2)

The parameters u1 and u2 must be of type UNSIGNED. Consider the following example:

```
Result := UAND( 16#FF9, 16#703 ); {Returns 16#701}
```

For More Information:

For information on specifying extended-digit notation, see Section 2.1.1.

8.77 UDEC Function

The UDEC function returns a character-string value that is the unsigned decimal equivalent of the specified parameter. The return value is compatible with all other string types.

UDEC(x[[, length[[, digits]]]])

The parameter x is the expression to be converted. This parameter must have a size that is known at compile time; it cannot be VARYING OF CHAR, a conformant parameter, or a schema type. Parameter x must be 32 bits or less in length.

Two optional integer parameters specify the length of the resulting string and the minimum number of significant digits to be returned. If you specify a length that is too short to hold the converted value, the resulting string is truncated on the left. If you do not specify values for the optional parameters, a default length and a default minimum number of significant digits is used. The defaults are 11 characters for the length and 10 characters for the minimum number of digits. Consider the following:

```
VAR
    Account : INTEGER;
{In the executable section:}
Account := 3;
WRITELN( UDEC( Account ) );
```

For More Information:

- On conformant parameters (Section 6.3.6)
- On VARYING OF CHAR (Section 2.6.2)

8.78 UINT Function

The UINT function converts the value of the parameter and returns its unsigned equivalent.

UINT(x)

The parameter x must be of an ordinal type.

No error results if x is of type INTEGER and has a negative value. In that case, the internal representation of x is returned as an unsigned number.

For More Information:

For information on range and support for the UNSIGNED data type, see the *VAX Pascal Reference Supplement for VMS Systems.*

8.79 UNDEFINED Function

The UNDEFINED function returns a Boolean value that specifies whether the parameter contains a reserved operand.

UNDEFINED(x)

The parameter x must be a variable of type REAL, SINGLE, DOUBLE, or QUADRUPLE. The function returns TRUE if x contains a value that has been reserved by the system or machine architecture. If x does not contain a reserved value, the function returns FALSE. If x contains a reserved operand and if you attempt to use x in arithmetic computations, an error results.

For More Information:

For information on operands reserved by the operating system or architecture, see the *VAX Pascal Reference Supplement for VMS Systems.*

8.80 UNOT Function

The UNOT function returns an unsigned value that represents a binary logical NOT operation on each bit of the specified parameter.

UNOT(u)

The u parameter must be an expression of type UNSIGNED. Consider the following example:

```
Result := UNOT( 16#FF9 );   {Returns 16#FFFFF006}
```

For More Information:

For information on specifying extended-digit notation, see Section 2.1.1.

8.81 UNPACK Procedure

The UNPACK procedure copies components of a packed array to an unpacked array variable.

UNPACK(z, a, i)

The parameter z is a packed array. The parameter a is an unpacked array variable. The parameter i is the starting value of the index of a.

The number of components in parameter a must be greater than or equal to the number of components in z. The UNPACK procedure assigns the components of z, starting with z[low-bound], to the array a, starting with a[i], until all the components in z are used.

In general, when specifying i, keep in mind that the upper bound of a (that is, n) must be greater than or equal to i + v–u, where v is the upper bound of a and u is the lower bound of a. That is, $ORD(n)$ must be greater than or equal to $ORD(i) + ORD(v) - ORD(u)$.

Normally, you cannot pass the individual components of a packed array to formal VAR parameters; you must unpack the array first. Consider the following:

```
VAR
    p : PACKED ARRAY[1..10] OF CHAR;
    a : ARRAY[1..10] OF CHAR;
    i : INTEGER;
PROCEDURE Process_Components( VAR Ch : CHAR ); {Body...}
```

```
{In the executable section:}
READ( p );
UNPACK( p, a, 1);
FOR i := 1 TO 10 DO
    Process_Components( a[i] ); {Pass each component to procedure}
```

For More Information:

For information on arrays and packed arrays, see Section 2.4.1.

8.82 UOR Function

The UOR function returns an unsigned value of a binary logical OR operation on the corresponding pair of bits of two specified parameters.

UOR(u1, u2)

The u1 and u2 parameters must be of type UNSIGNED. Consider the following example:

```
Result := UOR( 16#FF9, 16#703 );   {Returns 16#FFB}
```

For More Information:

For information on specifying extended-digit notation, see Section 2.1.1.

8.83 UPPER Function

The UPPER function returns the upper bound for ordinal types, SET base types, and array indexes.

UPPER(x [[, n]])

The parameter x is a type identifier or variable of an ordinal, SET, or ARRAY type. The parameter n is an integer constant that denotes a dimension of x, if x is an array. If x is an array and if you omit the parameter n, VAX Pascal uses the default value 1. If x is an array, UPPER returns the upper bound of the nth dimension of x. If x is an ordinal type, UPPER returns the upper bound or largest value. If x is a SET, UPPER returns the upper bound of the SET base type.

Consider the following example.

```
TYPE
    A_Schema( a, b ) = a..a+b;
VAR
    x : A_Schema( 5, 10 );
{In the executable section:}
WRITELN( UPPER( BOOLEAN ) );        {Writes TRUE}
WRITELN( LOWER( x ) );              {Writes 5}
WRITELN( UPPER( x ) );              {Writes 15}
```

8.84 UROUND Function

The UROUND function converts the value of the parameter and returns its unsigned equivalent by rounding the fractional part of the value.

UROUND(x)

The parameter x must be of type REAL, SINGLE, DOUBLE, or QUADRUPLE.

No error results if the value of x is negative or greater than 4,294,967,295. In that case, the unsigned result is the rounded parameter value MOD 4,294,967,296.

For More Information:

For information on range and support of the UNSIGNED data type, see the *VAX Pascal Reference Supplement for VMS Systems*.

8.85 UTRUNC Function

The UTRUNC function converts the parameter and returns its unsigned equivalent by truncating the fractional part of the value.

UTRUNC(x)

The parameter x must be of type REAL, SINGLE, DOUBLE, or QUADRUPLE.

No error results if the value of x is negative or greater than 4,294,967,295. In that case, the unsigned result is the truncated parameter value MOD 4,294,967,296.

For More Information:

For information on range and support of the UNSIGNED data type, see the *VAX Pascal Reference Supplement for VMS Systems*.

8.86 UXOR Function

The UXOR function returns an unsigned value of a binary logical exclusive-OR operation on the corresponding pair of bits of two specified parameters.

```
UXOR( u1, u2 )
```

The u1 and u2 parameters must be of type UNSIGNED.

```
Result := UXOR( 16#FF9, 16#703 ); {Returns 16#8FA}
```

For More Information:

For information on specifying extended-digit notation, see Section 2.1.1.

8.87 WRITEV Procedure

The WRITEV procedure writes characters to a character-string variable of type VARYING OF CHAR or discriminated STRING, by converting the values of the parameters in the procedure call to textual representations. The behavior of WRITEV is analogous to that of the WRITELN function; the character-string parameter is analogous to a one-line file.

```
WRITEV( str, parameter-list [[, ERROR := error-recovery]] )
```

The parameter str cannot appear within the parameter-list; if you attempt to do this, unexpected results may occur. An error occurs if WRITEV reaches the maximum length of the character string before the values of all the parameters in the procedure call have been written into the string. The error-recovery parameter indicates the action to be taken if an error occurs while the WRITEV procedure is executing. Consider the following:

```
TYPE
    Flower = ( Daisy, Lily, Orchid, Tulip );
VAR
    Real_Var : REAL VALUE 232.705;
    Write_String : VARYING[21] OF CHAR;
{In the executable section:}
WRITEV( Write_String, Daisy, Real_Var:7:3, PRED( Bouquet ) );
{Write_String contains ' DAISY232.705   LILY'}

WRITEV( Write_String, Daisy, Real_Var:7:3, PRED( Bouquet ),
        Bouquet );
{Error: there is no more room in the string parameter}
```

For More Information:

- On VARYING OF CHAR strings (Section 2.6.2)
- On error recovery codes (*VAX Pascal Reference Supplement for VMS Systems*)

8.88 XOR Function

The XOR function returns a value (of the same type as the parameters) of a binary logical exclusive-OR operation on two specified parameters.

XOR(p1, p2)

The p1 and p2 parameters must be of the same type and must be of either the BOOLEAN or SET types.

```
Result := XOR( ['A','B','C'],['B','C','D'] );   {Returns ['A','D']}
```

For More Information:

- On Boolean types (Section 2.1.4)
- On SET types (Section 2.4.3)

8.89 ZERO Function

The ZERO function returns data, whose type depends on the context of the function call, that sets any variable (except a file variable) to its binary zero. If you attempt to use the ZERO function to initialize a file variable, an error occurs. Do not specify a parameter list when you call the ZERO function.

Table 8–3 shows the value that ZERO assigns for each data type.

Table 8–3:   Value of ZERO

Data Type	Value
INTEGER	0
UNSIGNED	0
CHAR	The character NUL

Table 8–3 (Cont.): Value of ZERO

Data Type	Value
BOOLEAN	FALSE
Enumerated	The enumerated element with ORD(element) = 0
Subrange	0^1
REAL	0.0
DOUBLE	0.0
QUADRUPLE	0.0
ARRAY	ZERO applied to each component
RECORD	ZERO applied to each field
VARYING OF CHAR, STRING	The null string
SET	The empty set
Pointer	NIL

[1]Note that an ordinal target with a subrange type can thus be initialized outside of the subrange. The compiler treats this as an error, if used in a compile-time expression.

The ZERO function is used in two ways. It can be used on the right side of an assignment statement that appears in the executable section where the target is either a variable, or, within the body of a function, the function identifier. It can also be used as a compile-time expression to initialize a variable in the TYPE, VAR, CONST, or VALUE sections.

For More Information:

- On data initialization (Chapter 2)
- Using the ZERO function with records (Section 2.4.2.2)

Chapter 9

Input and Output Processing

The VAX Pascal I/O module provides an extensive set of predeclared routines. This chapter discusses the following topics:

- File organizations (Section 9.1)
- File component formats (Section 9.2)
- File access methods (Section 9.3)
- File Locking (Section 9.4)
- TEXT files and formatting (Section 9.5)
- I/O routines (Section 9.6)

For More Information:

For information on environment-specific I/O details, see the *VAX Pascal Reference Supplement for VMS Systems*.

9.1 Files and File Organizations

A **file** is an organized collection of logically related data items. Data items within files are called **file components**. The file organization determines how components are situated in the physical file, what types of access information are present in each component, and how components may be accessed by a program.

To open a file, you can call the OPEN procedure. Also, you usually call one of the EXTEND, RESET, and REWRITE procedures to establish a starting position for reading from or writing to a file. (For some files, you can use procedures to locate a specific component to access.)

VAX Pascal makes distinctions between external and internal files. An **external file** has a name in a directory and exists outside the context of a VAX Pascal program. An **internal file** has no name and is not retained after the program finishes execution.

A file declared in the program heading is external by default. A file declared in a nested block is internal by default. To change the default for internal files, call the OPEN procedure. The file is then considered external and is retained with the specified name after the program has finished execution. If you open an internal file with the EXTEND, RESET, or REWRITE procedure, the file remains an internal file. The EXTEND, RESET, and REWRITE procedures do not allow you to explicitly rename your file.

Both internal and external files have a file organization. A **file organization** defines the physical arrangement of components within the file. The VAX Pascal I/O model includes three file organizations: sequential, relative, and indexed. The following sections describe these organizations.

Default Information:

If you do not specify a file organization at the time of file creation (using the OPEN procedure), VAX Pascal creates a file with sequential organization.

9.1.1 Sequential File Organization

Sequential file organization specifies that file components are stored one after the other, in the order in which they were entered into the file. VAX Pascal supports this organization for files on disk. This is the only organization supported for files on magnetic tape, on terminals, on card readers, and on line printers. Figure 9–1 illustrates this file organization.

Figure 9–1: Sequential File Organization
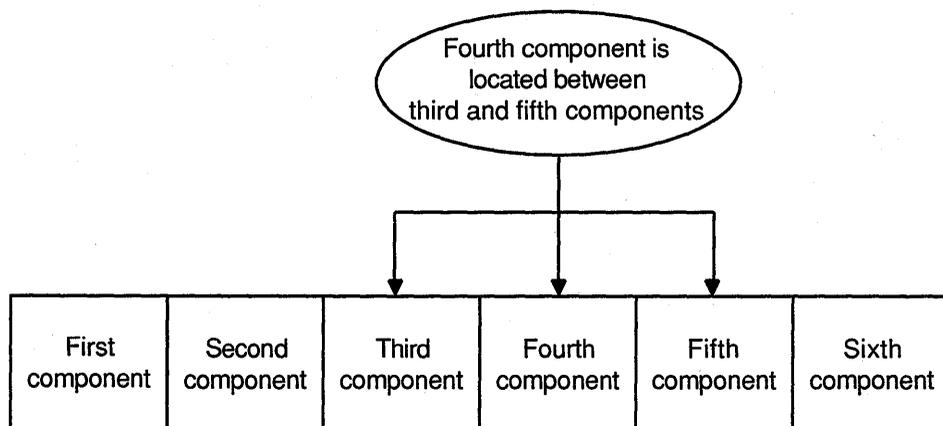
ZK–1332A–GE

You cannot insert components between any two existing components, because no physical space separates them. You can only add records to the end of the file (the most recently added component), truncate the file from a specified component to the end of the file, or rewrite the file.

For More Information:

- On component formats in sequential files (Section 9.2)
- On access methods for sequential files (Section 9.3)

9.1.2 Relative File Organization

Relative file organization consists of a series of fixed-length component positions (called **cells**) numbered consecutively from 1 to n. The numbered, fixed-length cells enable VAX Pascal to calculate the component's physical position in the file. The cell numbers are called **relative component numbers**. VAX Pascal supports this organization on disk files only. Figure 9–2 illustrates this file organization.

Figure 9–2: Relative File Organization

Relative Cells

1	2	3	4	5	6
First component written	Empty cell	Second component written	Third component written	Fourth component written	Fifth component written
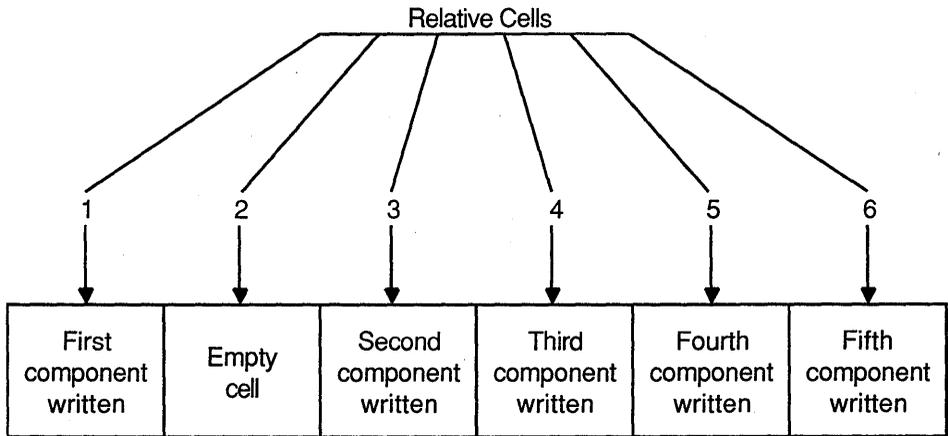
ZK-1333A-GE

Each component in the file may be randomly assigned to a specific cell. You can place components in unused cells and in cells from which components have been deleted. You cannot replace a component in a cell, but you can modify an existing component.

The length of the actual component may vary even though the cell containing the component is of a fixed length. If the component is smaller than the cell, the remaining space in the cell is unused.

For More Information:

* On component formats in relative files (Section 9.2)
* On access methods for relative files (Section 9.3)

9.1.3 Indexed File Organization

Indexed file organization specifies that, in addition to the stored components, there exists at least a primary key and possibly alternate keys (first alternate key, second alternate key, and so forth). VAX Pascal uses the primary key to store components and uses a program-specified key or keys to retrieve data. VAX Pascal supports this organization on disk files only.

To define a key and certain characteristics of keys, use the KEY attribute. You can define up to 254 alternate keys.

An **index** is a structure that provides a component collating sequence for the file, a mechanism for accessing components in different orders depending on the specified index (name, address, telephone number, and so forth).

Figure 9–3 illustrates an indexed file organization that uses only a primary key.

Figure 9–3: Indexed File Organization
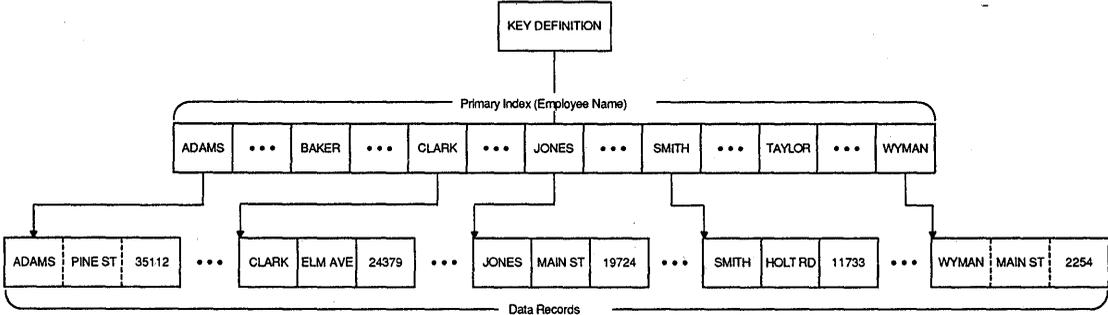
ZK–1335A–GE

Notice that in Figure 9–3, the components are logically stored in an order that is determined by the primary index. (The actual physical location of components is transparent to your program.) Figure 9–4 illustrates the presence of a first alternate index (determined by the presence of first alternate keys in the components) that points to components stored in order by the primary key.

Figure 9–4: A First Alternate Key
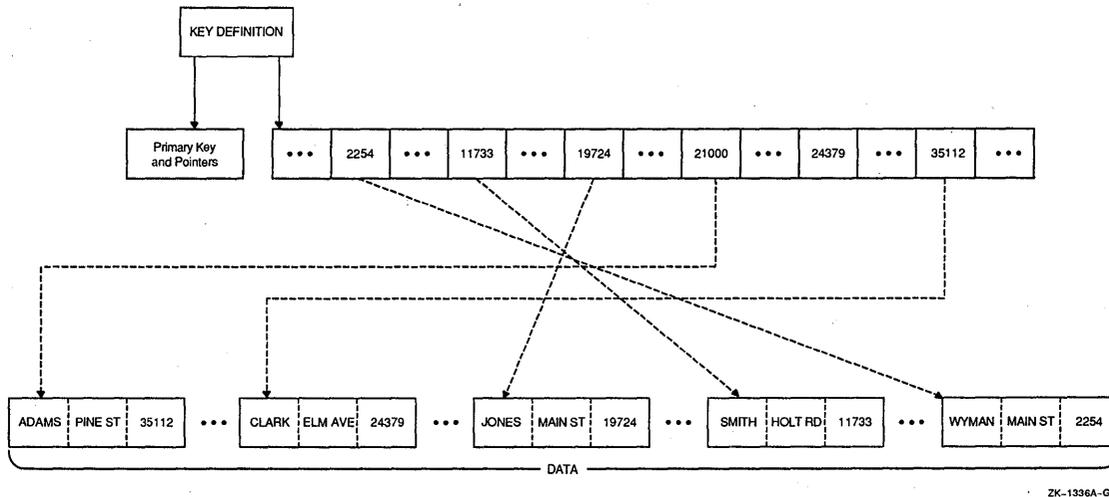
ZK–1336A-G

For More Information:

- On component formats in indexed files (Section 9.2)
- On access methods for indexed files (Section 9.3)
- On the KEY attribute (Section 10.2.21)

9.1.3.1 Keys

In an indexed file, each component includes one or more **key fields** (or simply **keys**) that VAX Pascal uses to build the specified indexes. Each key is indentified by its location within the component, its length, and its data type.

A key may be one of the following data types:

- A single, contiguous character string
- A 2- or 4-byte unsigned binary number
- A 1-, 2-, or 4-byte signed integer

You can use the KEY attribute to specify certain characteristics about the index and about the keys themselves. The following table describes these characteristics:

Keys	Description
Sort order	This characteristic determines how VAX Pascal creates an index, and determines the order in which VAX Pascal accesses components. The order can either be ascending or descending. If you specify ascending order, VAX Pascal considers the component with an equal or greater key value to be the "next" component for access. If you specify descending order, VAX Pascal considers the component with an equal or lesser key value to be the "next" component for access. Using different indexes and both ascending or descending order, you can use different collating sequences for a file's components according to the needs of your application.
Duplicate keys	This characteristic permits you to use the key value in more than one component. However, only the first component having the key value can be accessed randomly; other components having the same key value can only be accessed sequentially.
Changeable keys	This characteristic applies to alternate keys only. When you specify changeable keys, you can change the alternate keys in a component when you update the component. When an alternate key value changes, VAX Pascal automatically adjusts the appropriate index to reflect the new key value.

If you do not allow duplicate keys, VAX Pascal rejects any attempt to place a component into a file if it contains a key value that is a duplicate of an existing component. If you do not explicitly create the file to accept alternate key values, then attempts to change key values generate an error.

For More Information:

- On the KEY attribute (Section 10.2.21)

- On additional key characteristics (*VAX Pascal Reference Supplement for VMS Systems*)

9.2 Component Formats

When you declare a file variable in your program, VAX Pascal automatically creates a file buffer variable of the component type. This variable takes on the value of one file component at a time. You can access only one file component, called the **current component**, at a given time.

To reference the file buffer variable, you write the name of the associated file variable, followed by a circumflex (^). No operations can be performed on the file while a reference to the file buffer variable exists.

Predeclared I/O procedures move the file position. As the file position changes, the variable in the file buffer changes. Figure 9-5 shows how this change occurs.

Figure 9-5: File Buffer Contents

one file component

| 90 | 65 | 70 | 73 | 81 | 89 |
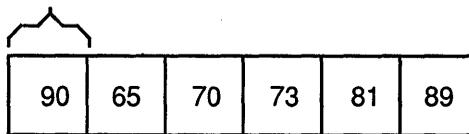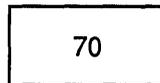
file position

| 70 | File buffer Math_Scores^

ZK-0101-GE

Suppose you declare a file variable Math_Scores of type FILE OF INTEGER. You might call a procedure to move the file position to the first component of this file. At this point, the file buffer variable Math_Scores^ equals the value of the first component (here, 90). If you call other procedures to advance the file position by three components, Math_Scores^ then equals the value of the third component (here, 70).

In each file, all components are of the same file component format. **Component format** defines the size (or maximum size) of each component and any processing information needed in addition to the data portion of the component. The VAX Pascal I/O model supports the following formats for file components:

- Fixed-length format (Section 9.2.1)
- Variable-length format (Section 9.2.2)
- Stream format (Section 9.2.3)

Default Information:

For new TEXT and VARYING OF CHAR files, VAX Pascal creates variable-length components by default. For other types of new files, VAX Pascal creates fixed-length components. If you access an existing file, your specified component type must match the component type specified at the creation of the file; if it does not, you generate an error.

Table 9–1 shows which of the file organizations support which of the component formats.

Table 9–1: File Organization Support for Component Format

Organization	Supported Component Format
Sequential	All component formats
Relative[1]	Fixed length Variable length
Indexed	Fixed length Variable length

[1]Although the relative file organization allows variable-length components, those variable-length components are contained in a fixed-length cell that must be large enough to contain the largest stored component.

For More Information:

For information on TEXT files, see Section 9.5.

9.2.1 Fixed-Length Component Format

Fixed-length components are all the same length. The **fixed-length component format** is supported in all file organizations. VAX Pascal determines the length of a component at the time of file creation. You cannot change the length of the components after you create the file.

9.2.2 Variable-Length Component Format

The **variable-length component format** enables components to be only as long as the data requires. The variable-length format is supported in all file organizations.

When you use OPEN to create a file of variable-length components, you can specify the value (in bytes) of the largest component permitted in the file. Any attempt to store a component containing more bytes than the specified value results in an error.

9.2.3 Stream Component Format

The **stream component format** is a continuous stream of bytes that contains special delimiting characters (called **terminators**) that separate components. In addition to being recognized as delimiters, VAX Pascal considers the terminators to be a valid part of the component data. The stream format is supported only in sequential files on disk.

The following are acceptable types of stream component formats:

Type	Description
STREAM_CR	This type recognizes a carriage return character as the component terminator.
STREAM_LF	This type recognizes a line feed as the component terminator.
STREAM	This type uses a terminator from a limited set of special characters: the carriage return (CR); the carriage-return/line-feed combination (CR/LF); or the form feed (FF).

9.3 Component Access Modes

A **component access mode** is a method by which VAX Pascal retrieves components from a file. Although you cannot change the file organization or component format after file creation, you can change the component access mode each time you access a file. The VAX Pascal I/O model defines two component access modes: sequential and random access. Random access can be further broken down into the categories of random access by number (also called **direct** access) and random access by key value (also called **keyed** access). The sections that follow describe these access methods in further detail.

You specify the access method using the OPEN procedure when you open a file. You cannot change the access method unless you first use the CLOSE routine, and then reopen the file specifying a new access method.

Before attempting to use any of the access methods on a file, VAX Pascal determines the organization of the file. The organization determines how the specified access method works. For instance, sequential access on a sequentially organized file works differently than sequential access on an indexed file.

Default Information:

- The default is the sequential access method.

- You can always process a file using sequential access, even when the currently specified access method is one of the direct access methods.

- By default, VAX Pascal does not designate a component as a starting point for access; you must do this explicitly using one of the RESET, REWRITE, or REWIND procedures, or using an access-specific procedure to locate a specified component.

Table 9–2 shows which file organizations support which component access modes.

Table 9–2: File Organization Support for Component Access Modes

Access Mode	Sequential Organization	Relative Organization	Indexed Organization
Sequential	Yes	Yes	Yes
Random by relative component number (direct access)	Yes[1]	Yes	No
Random by key value (keyed access)	No	No	Yes

[1]This access is permitted with a fixed-length component format on disk only.

For More Information:

- On file organizations (Section 9.1)
- On component formats (Section 9.2)

9.3.1 Sequential Access

Using the **sequential access method**, storage or retrieval begins at a designated position in the file and continues through the file according to the component's position in storage. You can specify two starting points for sequential access: the beginning of the file (using REWRITE, or RESET) or the end of the file (using EXTEND).

The following are the VAX Pascal I/O routines that are used for sequential access:

EOF	REWRITE
EXTEND	STATUS
GET	TRUNCATE
PUT	UFB
READ	UNLOCK
RESET	WRITE

For More Information:

- On file organization (Section 9.1)
- On component format (Section 9.2)

Sequential Access to Sequential Files

To retrieve a component in a sequential file, you must retrieve all components from the time you establish a current position (using either EXTEND, RESET, or REWIND) to the desired component. After an operation to the file, VAX Pascal positions the file pointer to the next file component in anticipation of the next operation on the file.

To access a previous component, you must reopen (implicitly or explicitly) and reread the previous components; or, you can reopen the file, switching to random access mode.

You cannot add components in between any two components. You can add components only to the current end of the file.

Figure 9–6 illustrates sequential access to sequential files.

Figure 9–6:   Sequential Access to a Sequential File
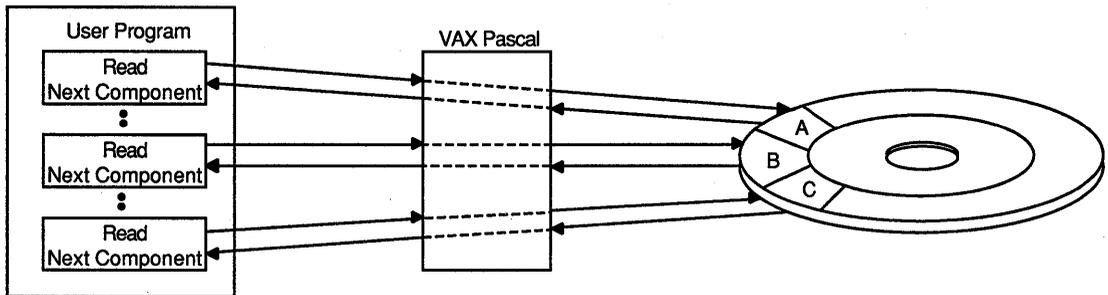
ZK–1338A–GE

9.3.1.2 Sequential Access to Relative Files

VAX Pascal can use sequential access for relative files as long as the components are fixed length. VAX Pascal tries to store or retrieve from the cell whose relative component number is one higher than the most recently accessed cell.

You cannot overwrite a component, but you can modify the contents of the current component. If the cell with the next highest relative component number contains a component and if you are trying to store data in that cell, you generate an error.

Figure 9–7 illustrates the use of sequential access to read from a relative file:

Figure 9–7: Using Sequential Access to Read from a Relative File
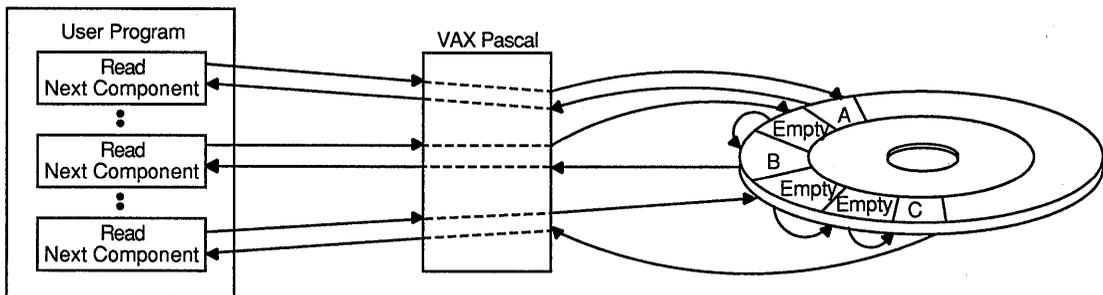
ZK–1339A–GE

Figure 9–8 illustrates the use of sequential access to write to a relative file:

Figure 9–8: Using Sequential Access to Write to a Relative File
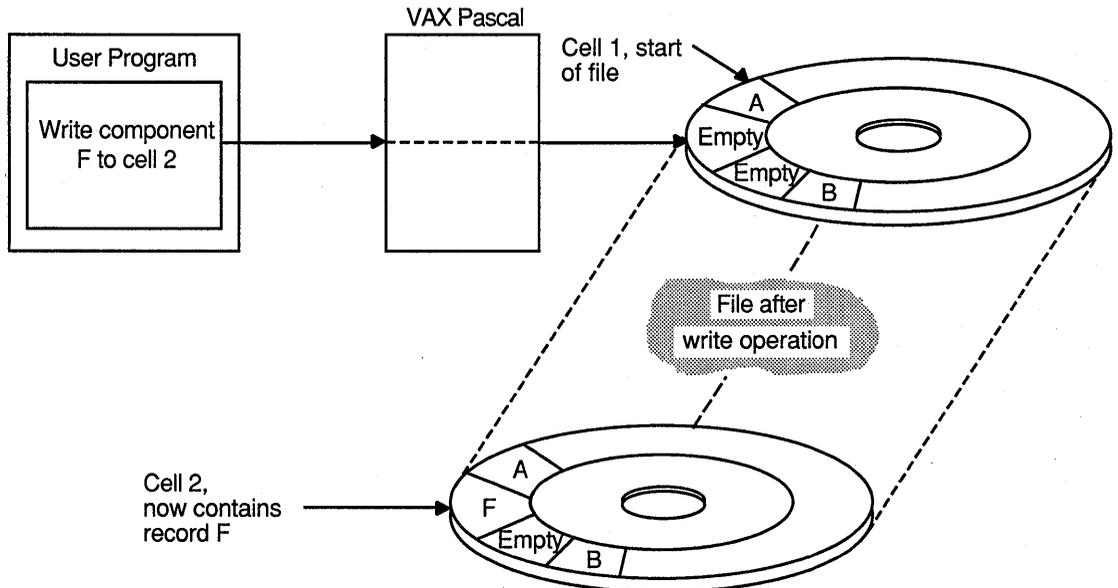
ZK–1340A–GE

In Figure 9–8, VAX Pascal writes the component to the current cell. If the program requests that another component be stored sequentially, then VAX Pascal places that component in cell 3. If the program places another request to store a component sequentially, an error occurs because cell 4 contains component B.

9.3.1.3 Sequential Access to Indexed Files

When sequentially accessing an indexed file, VAX Pascal uses a specified index to determine the order in which to sequentially process the file components. The specified keys are called the **keys of reference**.

If you specify ACCESS_METHOD := SEQUENTIAL when you open an indexed file, you can only access components sequentially according to the primary key. If you specify ACCESS := KEYED when you open an indexed file, you can access components sequentially according to any key.

When sequentially writing components to an indexed file, VAX Pascal stores the component according to the primary key. If your program uses secondary keys, VAX Pascal updates the secondary key pointers to include the newly stored component.

9.3.2 Random Access

Random access allows you to access file components in an order that is not dependent on the file organization or on the order in which the components are stored. Random access is available for all relative and indexed files, and for sequential files composed of fixed-length components (the fixed-length components allow VAX Pascal to "count" component positions in the sequential file without having to worry about variations in the lengths of the components).

VAX Pascal supports the following types of random access:

* Random access by relative component number (direct access)
* Random access by key value (keyed access)

The following are the VAX Pascal I/O routines that are used for random access:

Random access by relative component number:	
DELETE	UFB
EOF	UNLOCK
FIND	UPDATE
LOCATE	

Random access by key:	
EOF	UFB
FINDK	UNLOCK
RESETK	UPDATE

For More Information:

- On file organization (Section 9.1)
- On component format (Section 9.2)

9.3.2.1 Random Access by Relative Component Numbers (Direct Access)

VAX Pascal supports random access by relative component numbers for relative files and for sequential files with fixed-length components on disk. To access the desired component, you need to specify the relative component number of the corresponding cell; relative component numbers are relative to the beginning of the file.

Figure 9–9 illustrates the process of randomly accessing cells in a file.

Figure 9–9: Using Random Access on Sequential[1] and Relative Files
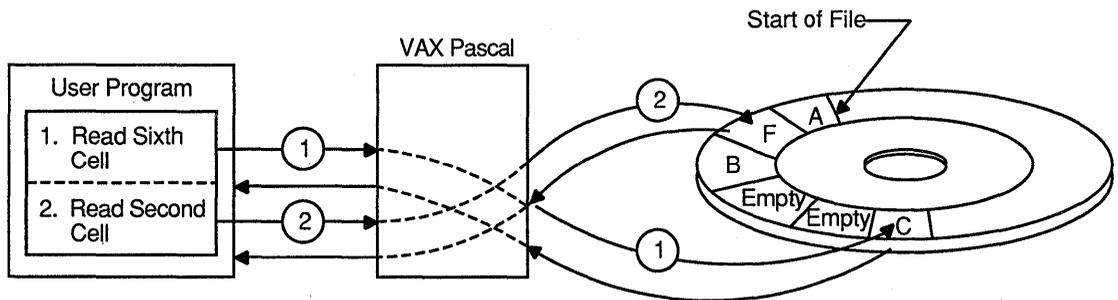
ZK–1354A–GE

9.3.2.2 Random Access to Indexed Files (Keyed Access)

VAX Pascal supports random access to indexed files. To retrieve a component, you must specify an index (primary index, first alternate index, second alternate index, and so forth) and a key value. To store a component, VAX Pascal determines existing keys from the file organization and stores the record (and alternate key information) according to information as it exists in the data portion of the component.

---

[1] Fixed-length components on disk storage only.

Your program can use several methods to randomly access a record by key:

- Exact match of key values.
- Approximate match of key values. When accessing an index in ascending sort order, VAX Pascal returns the component that has the next higher key value (in descending order, the component with next lower key value).
- Generic match of key values. Generic matching is applicable to string data-type keys only (PACKED ARRAY OF CHAR record fields). For a generic match, the program need specify only a match of some specified number of leading characters in the key.
- Combination of approximate and generic match.

9.4 File Locking

Under some circumstances, if a file component is in the process of being read or written to by one program, VAX Pascal **locks** the component, preventing other programs from accessing the component. This prevents programs from accessing outdated or inaccurate data.

If you OPEN a file and specify that the file is not to be shared, or that reading or writing sharing is allowed, VAX Pascal may not lock the record.

Record locking occurs most often when accessing relative and indexed files, but it can happen when accessing sequential files as well. Successful calls to FIND, FINDK, GET, RESET, and RESETK lock the current component. If you want to make a locked file component available to other programs on the system, you can call the UNLOCK procedure.

For More Information:

- On enabling other programs to access new files created with OPEN (Section 9.6.11)
- On unlocking components (Section 9.6.22)

9.5 TEXT Files

Files of type TEXT are sequences of characters with special markers (end-of-line and end-of-file) added to the file. Although each character of a TEXT file is one file component, the end-of-line marker allows you to process the file line-by-line (using READLN, WRITELN, or EOLN), if you choose.

The predeclared file variables INPUT and OUTPUT are files of type
TEXT. They refer to the standard input and output files. (When executing
programs at a terminal, INPUT and OUTPUT default to the terminal you
are using.)

The file type FILE OF CHAR differs from TEXT files in that FILE OF CHAR
allows a single character to be the unit of transfer between a program and
its associated I/O devices and that FILE OF CHAR files do not include
special markers. FILE OF CHAR components are always read with the
READ procedure, and must be read exclusively into variables of type CHAR,
including CHAR components of structured variables. You cannot use the
EOLN, READLN, and WRITELN routines on FILE OF CHAR files.

Default Information:

- A new file of type TEXT or FILE OF VARYING OF CHAR is a sequential
  file with variable-length components.
- All TEXT file routines use the predefined files INPUT and OUTPUT by
  default.
- VAX Pascal performs an implicit call to RESET on the predeclared
  file INPUT and an implicit call to REWRITE on the predeclared file
  OUTPUT.
- The default size for the output buffer is 255 characters for TEXT files.

The following are the VAX Pascal I/O routines that are used only with TEXT
files:

EOLN	READLN
LINELIMIT	WRITELN
PAGE	

9.5.1 Carriage Control

Some devices, such as printers and terminals, are carriage-control devices
and require characters to provide information regarding output. VAX Pascal
supports the following carriage-control options.

OPEN Parameter Option	Description
LIST	Single spacing between components. This is the default carriage-control option for all TEXT files (including OUTPUT) and VARYING OF CHAR files.
CARRIAGE, FORTRAN	The first character of every output line is a carriage-control character.
NONE, NOCARRIAGE	No carriage control. This is the default for all files other than TEXT and VARYING OF CHAR files.

For FORTRAN carriage control, if output is directed to devices that do not use carriage-control characters, the character is written into the file as a component and is read back when the file is opened for input. If output is directed to devices that do use carriage-control, then the OPEN parameter options described previously determine the action taken by VAX Pascal.

Table 9–3 summarizes carriage control characters and their effects. For purposes of carriage control, VAX Pascal ignores any characters other than those listed in the table.

Table 9–3:   Carriage Control Characters

Character	Meaning
' + '	Overprinting: starts output at the beginning of the current line.
' '	Single spacing: starts output at the beginning of the next line.
' 0 '	Double spacing: skips a line before starting output.
' 1 '	Paging: starts output at the top of a new page.
' $ '	Prompting: starts output at the beginning of the next line and suppresses carriage return at the end of the line.
' ' (0)	Prompting with overprinting: suppresses line feed at the beginning of the line and carriage return at the end of the line; note that this character is the ASCII character NUL.

9.5.2   Prompting on a Terminal

Normally, when you call the WRITE procedure to access a TEXT file connected to a terminal, VAX Pascal accumulates the characters in a line buffer until a subsequent WRITELN procedure is executed. In effect, WRITELN generates an end-of-line marker. When you complete a line or close a file, VAX Pascal writes a full line of characters to the specified TEXT file.

VAX Pascal can manipulate partial lines in a TEXT file; however, when characters are being written to a terminal output file opened with the LIST carriage control option (LIST is the default), partial lines are written to the terminal before input is transferred from any terminal to the line buffer of a TEXT file. In this situation, VAX Pascal searches for all TEXT files opened for output on terminals; it then writes to those files any partial lines contained in the files' respective line buffers. These partial lines, called prompts, appear on the screen. You respond to a prompt by typing a line of input data terminated by pressing RETURN.

Consider the following:

```
WRITE( 'Name three presidents:' );
READ( Pres1, Pres2, Pres3);
```

VAX Pascal stores the string 'Name three presidents:' in the output buffer; when executing the READ procedure, VAX Pascal locates the TEXT file opened for output to the appropriate terminal and the partial output buffer is written, causing the string 'Name three presidents:' to appear on the terminal screen. The user can then begin typing on the same line as the prompt, providing the names of three presidents. Note that prompting works only for files associated with interactive terminals. For any other files, VAX Pascal does not write output until you start the new line with a WRITELN procedure.

9.5.3 Delayed Device Access to Text Files

The Pascal standard requires that the file buffer always contain the next file component that will be processed by the program. This definition can cause problems when the input to the program depends on the output most recently generated. To alleviate such problems in the processing of the TEXT files, VAX Pascal uses a technique called **delayed device access**, also known as **lazy lookahead**.

As a result of delayed device access, VAX Pascal does not retrieve an item of data from a physical file device and does not insert it in the file buffer until the program is ready to process it. VAX Pascal fills the file buffer when the program makes the next reference to the file. A reference to the file consists of any use of the file buffer variable, including its implicit use in the GET, READ, and READLN procedures, or any test for the status of the file, namely, the EOF, EOLN, STATUS, and UFB functions.

The RESET procedure, which is required when any TEXT file is opened for input, initiates the process of delayed device access. (Note that RESET is done automatically on the predeclared file INPUT.) RESET expects to fill the file buffer with the first component of the file. However, because of delayed device access, an item of data is not supplied from the input device to fill the file buffer until the next reference to the file.

When writing a program for which the input will be supplied by a TEXT file, you should be aware that delayed device access occurs. Because RESET initiates delayed device access, and because EOF and EOLN cause the file buffer to be filled, you should place the first prompt for input before any tests for EOF or EOLN. The information you enter in response to the prompt supplies data that is retained by the file device until you make another reference to the input file.

Consider the following:

```
VAR
    i : INTEGER;
{In the executable section:}
WRITE( 'Enter an integer or an empty line: ' );
WHILE NOT EOLN DO
    BEGIN
    READLN( i );
    WRITELN( 'The integer was: ', i:1 );
    WRITE( 'Enter an integer or an empty line: ' );
    END;
WRITELN( 'Done' );
```

The first reference to the file INPUT is the EOLN test in the WHILE statement. When the test is performed, VAX Pascal attempts to read a line of input from the TEXT file. Therefore, it is very important to prompt for the integer or empty line before testing for EOLN.

Suppose you respond to the first prompt by supplying an integer as input. Access to the input device is delayed until the EOLN function makes the first reference to the file INPUT. The EOLN function causes a line of text to be read into the internal line buffer. The subsequent READLN procedure reads the input value from the line of text and assigns it to the variable i. The WRITELN procedure writes the input value to the text file OUTPUT. The final statement in the WHILE loop is the request for another input value. The loop terminates when EOLN detects the end-of-line marker.

A sample run of a program containing this loop might be as follows:

```
Enter an integer or an empty line:  10
The integer was:  10
Enter an integer or an empty line:  99
The integer was:  99
Enter an integer or an empty line:  [RETURN]
Done
```

The following program fragment illustrates a method of writing the same loop that does not take into account delayed device access and therefore produces incorrect results:

```
WHILE NOT EOLN DO
    BEGIN
    WRITE( 'Enter an integer or an empty line: ' );
    READLN( i );
    WRITELN( 'The integer was: ', i:1 );
    END;
```

The EOLN test at the beginning of the loop causes the file buffer to be filled. However, because no input has been supplied yet, the prompt does not appear on the screen until you have supplied input to fill the INPUT file buffer.

A sample run of a program containing this loop might be as follows:

```
10
Enter an integer or an empty line: The integer was: 10
99
Enter an integer or an empty line: The integer was: 99
 RETURN
```

The prompt always appears after you type a value for i.

Delayed device access can produce unexpected results if you try to use the STATUS function to test the status of a TEXT file after you have performed a READLN procedure on the file. Remember that a READLN procedure call actually performs a READ procedure on each variable listed as a parameter, then performs a READLN procedure to position the file at the beginning of the next line. Therefore, a call to STATUS after a READLN procedure actually tests whether the file was successfully positioned. To test the status of the file, STATUS causes delayed device access to occur, thereby filling the file buffer with the next component. If you want to test the successful reading of data from the input file, you should read the data with the READ procedure, call the STATUS function, and then perform a READLN procedure to advance the file to the beginning of the next line.

9.5.4 Writing Partial Lines to Terminals

The WRITE procedure buffers output to the terminal until the WRITELN procedure is called. If too many characters are buffered, it can cause the VAX Pascal buffer to overflow. The default size for this buffer is 255

characters for TEXT files. If you want to increase the internal buffer size, you can explicitly open the predeclared file OUTPUT with a larger record length. Consider the following:

```
OPEN( OUTPUT, RECORD_LENGTH := 512 );
```

If you want each record to go directly to the terminal without buffering until the next WRITELN, you can explicitly open the predeclared file variable OUTPUT without carriage control. In this mode, the WRITELN procedure will write the information to the file without adding any carriage control. However, you need to include the carriage return and the line feed characters in the output strings; the WRITELN procedure no longer provides these automatically. Consider the following:

```
CONST
   LF = 10;   {ASCII control characters}
   CR = 13;
{In the executable section:}
OPEN( OUTPUT, CARRIAGE_CONTROL := NONE );
WRITELN( ''(LF)'Output this' );
WRITELN( 'string directly' );
WRITELN( 'to the terminal'(CR) );
```

This is useful when you are writing escape sequences or other graphics characters to terminal devices.

9.6 I/O Routines

VAX Pascal provides predeclared procedures and functions to perform input and output operations on file variables. These routines may operate differently depending on a file's organization and the currently-defined access method.

The I/O routines in the following sections appear in alphabetical order.

At any time during the execution of a process, a file variable is considered to be in one of three modes: **inspection, generation,** or **undefined.** When a file is reading input, it is in inspection mode. When output is being written to a file, the file is in generation mode. A file in an undefined state of processing is in undefined mode. The mode often determines the valid operations for the file.

Table 9–4 shows the mode required before execution of each I/O routine and shows the mode in which the file is left after each routine has executed.

Table 9–4: File Mode During I/O Processing

I/O Routine	Mode Before Execution	Mode After Execution	I/O Routine	Mode Before Execution	Mode After Execution
CLOSE	Any	Undefined	READ	Inspection	Inspection
DELETE	Inspection	Inspection	READLN	Inspection	Inspection
EOF	Inspection or generation	No change	RESET	Any	Inspection
EOLN	Inspection	Inspection	RESETK	Any	Inspection
EXTEND	Any	Generation	REWRITE	Any	Generation
FIND	Any	Inspection if successful; undefined if unsuccessful	STATUS	Any	No change, unless error
FINDK	Any	Inspection if successful; undefined if unsuccessful	TRUNCATE	Inspection	Generation
GET	Inspection	Inspection	UFB	Any	No change
LINELIMIT	Any	No change	UNLOCK	Inspection	Inspection
LOCATE	Any	Generation	UPDATE	Inspection	Inspection
OPEN	Undefined	Undefined	WRITE	Generation, unless keyed access, which may be any mode	Generation
PAGE	Generation	No change	WRITELN	Generation	Generation
PUT	Generation	Generation			

9.6.1 CLOSE Procedure

The CLOSE procedure closes an open file.

```
1. CLOSE (file_variable
         ,[[disposition]]
         ,[[user_action]]
         ,[[ERROR := error_recovery]] )
```

2. CLOSE (FILE_VARIABLE := file_variable
            [[,DISPOSITION := disposition]]
            [[,USER_ACTION := user_action]]
            [[,ERROR := error_recovery]] ...)

file_variable
no default
The name of the file variable associated with the file that VAX Pascal is to close.

disposition
same as for OPEN procedure
A value that determines what VAX Pascal is to do with the file after closing it. The disposition values are the same as those used for the OPEN procedure. The disposition value in the CLOSE procedure supersedes a disposition value specified in the OPEN procedure.

user_action
no default
A routine name that VAX Pascal calls to close the file. You can use a user-action routine to close the file using environment-specific capabilities.

error_recovery
stops execution after first error (default)
The action to be taken if an error occurs during execution of the routine.

Execution of the CLOSE procedure causes the system to close the file and, if the file is internal, to delete it. Each file is automatically closed when control passes from the block in which it is declared.

You cannot close a file that has not been opened (either explicitly by the OPEN procedure, or implicitly by the EXTEND, RESET, or REWRITE procedure). If you attempt to close a file that was never opened, an error occurs.

The file can be in any mode (inspection, generation, or undefined) before the CLOSE procedure is called. Execution of CLOSE sets the mode to undefined.

For More Information:

- On the OPEN procedure and parameters (Section 9.6.11)
- On the error processing parameter (Section 9.6.26.1)

9.6.2 DELETE Procedure

The DELETE procedure deletes the current file component. DELETE can be used only on files with relative or indexed organization that have been opened for direct or keyed access; it cannot be used on files with sequential organization.

DELETE(file_variable[[, ERROR := error_recovery]]);

file_variable
The name of the file variable associated with the file from which a component is to be deleted.

error-recovery
The action to be taken if an error occurs during execution of the routine.

The file must be in inspection mode before DELETE is called; the mode does not change after the procedure's execution.

When the DELETE procedure is called, the current component, as indicated by the file buffer, must already have been locked by a successful FIND, FINDK, GET, RESET, or RESETK procedure before it can be deleted. After deletion, the component is unlocked and the UFB function returns TRUE.

Consider the following:

DELETE(Accounts_Payable);

This procedure call deletes the current component. When the component has been deleted, it is unlocked and UFB(Accounts_Payable) returns TRUE. A run-time error occurs if the current component of Accounts_Payable is not locked.

For More Information:

* On file organizations (Section 9.1)
* On component access (Section 9.3)
* On the UFB function (Section 9.6.21)
* On the error processing parameter (Section 9.6.26.1)

9.6.3 EOF Function

The EOF function indicates whether the file pointer is positioned after the last component in a file by returning a Boolean value.

EOF[[(file_variable)]]

file_variable
The name of the file variable associated with the input file. If you omit the name of the file, the default is INPUT.

The file can be in either inspection or generation mode before EOF is called; however, end-of-file must be defined. The input operations GET, RESET, and FINDK are guaranteed to leave end-of-file defined. The file mode does not change after EOF has been executed.

EOF returns TRUE when the file pointer is positioned after the last component in the file, and returns FALSE up to and including the time when the last component of the input file is read into the file buffer. You must attempt to retrieve another file component after the last to determine whether the file is positioned at end-of-file.

When EOF is tested for a file with relative organization opened for direct access, the result is TRUE if the file is in inspection mode and the last GET or RESET operation positioned the file beyond the last existing component. If the file is in generation or undefined mode, the result of EOF is undefined.

When EOF is tested for a file with indexed organization opened for keyed access, the result is TRUE if the file is in inspection mode and the last FINDK, GET, RESET, or RESETK operation positioned the file beyond the last component with the current key number. Successful attempts at FINDK, GET, RESET, and RESETK cause EOF to be FALSE. If the file is not in inspection mode, EOF is undefined.

If you attempt to read a file after EOF becomes TRUE, an error results.

Consider the following:

```
Coupons := 0;
WHILE NOT EOF DO
   BEGIN
   READLN( Coupon_Amount );
   Coupons := Coupons + Coupon_Amount;
   END;
```

This example calculates the total value of the coupons contained in the file INPUT. The loop is performed while the EOF function returns FALSE.

For More Information:

* On component access (Section 9.3)
* On the error processing parameter (Section 9.6.26.1)
* On retrieval of file components (Section 9.5.3)

9.6.4   EOLN Function

The EOLN function tests for the end-of-line marker within a text file and returns a Boolean value.

EOLN [[(file_variable)]]

file_variable
The name of a file variable associated with a text file. If you omit the name of the file, the default is INPUT.

The file must be in inspection mode and EOF must return FALSE before EOLN is called. EOLN leaves the file in inspection mode.

The Boolean EOLN function returns TRUE when the file pointer is positioned after the last character in a line. When the EOLN function returns TRUE, the file buffer contains a blank character.

The EOLN function returns FALSE when the last component in the line is read into the file buffer. Another character must be read to cause EOLN to return TRUE and to cause the file buffer to be positioned at the end-of-line marker following the last character of the line. If you use the EOLN function on a nontext file, an error occurs.

Consider the following:

```
WHILE NOT EOF( Master_File ) DO
   BEGIN
   WHILE NOT EOLN( Master_File ) DO
      BEGIN
      READ( Master_File, x );
      IF NOT (x IN ['A'..'Z','a'..'z','0'..'9'])
      THEN
         Err := Err + 1;
      END;
   READLN( Master_File );
   END;
```

This example scans the characters on each line of a TEXT file called Master_File and checks for characters that are neither digits nor letters. If a nonnumeric or nonalphabetic character is encountered in the file, the counter Err is incremented by 1. The loop is executed until the last component in the file is read.

For More Information:

For information on TEXT files, see Section 9.5.

9.6.5 EXTEND Procedure

The EXTEND procedure opens an existing file, positions the file buffer after the last component, and prepares it for writing. It is commonly used to append to a file.

EXTEND(file_variable [[, ERROR := error-recovery]]);

file_variable
The name of the file variable associated with the output file.

error-recovery
The action to be taken if an error occurs during execution of the routine.

The file can be in any mode before EXTEND is called to set the mode to generation. If the file is an external file and is not already open, EXTEND opens it using the defaults for the OPEN procedure.

After execution of EXTEND, the file is positioned after the last component, and EOF and UFB return TRUE. If the file does not exist, EXTEND does not create it, but returns an error at run time.

A call to EXTEND on a relative file opened for direct access positions the file after its last existing component.

A call to EXTEND on an indexed file opened for random access by key positions the file after the last component relative to the primary key.

Consider the following:

```
VAR
    f : FILE OF INTEGER;
{In the executable section:}
OPEN( File_Variable := f,
      File_Name      := 'sample.dat',
      History        := OLD,
      Organization   := Relative,
      Access_Method  := Direct; );
EXTEND( f );
F^ := 20;
PUT( f );
```

These statements open an existing relative file named SAMPLE.DAT. The file will be positioned after the last record in the file. Subsequent PUT statements will append new components to the end of the file.

For More Information:

- On component access (Section 9.3)
- On default values for the OPEN procedure (Section 9.6.11)
- On the error processing parameter (Section 9.6.26.1)

9.6.6 FIND Procedure

The FIND procedure positions a file at a specified component. The file must be open for direct access and must contain fixed-length components.

FIND(file_variable, component-number [[, ERROR := error-recovery]]);

file_variable
The name of a file variable associated with a file that is open for direct access.

component-number
A positive integer expression that indicates the component at which the file is to be positioned. If the component number is zero or negative, a run-time error occurs.

error-recovery
The action to be taken if an error occurs during execution of the routine.

The FIND procedure allows direct access to the components of a file. You can use the FIND procedure to move forward or backward in a file.

After execution of the FIND procedure, the file is positioned at the specified component. The file buffer variable assumes the value of the component, and the file mode is set to inspection. If the file has relative organization, the current file component is locked. If there is no file component at the selected position, the file buffer is undefined (UFB becomes TRUE) and the mode becomes undefined. After any call to FIND, the value of EOF is undefined.

You can use the FIND procedure only when reading a file that was opened by the OPEN procedure. If the file is open because of a default open (that is, with EXTEND, RESET, or REWRITE), a call to FIND results in a run-time error because the default access method is sequential.

Consider the following:

```
FIND( Albums, Current + 2 );
```

If the value of Current is 6, this procedure causes the file position to move
to the eighth component; the file buffer variable Albums^ assumes the value
of the component. If no eighth component exists, Albums^ is undefined and
UFB (Albums) returns TRUE.

For More Information:

* On component access (Section 9.3)
* On the UFB function (Section 9.6.21)
* On the error processing parameter (Section 9.6.26.1)

9.6.7 FINDK Procedure

The FINDK procedure searches the index of an indexed file opened for keyed
access and locates a specific component.

```
FINDK( file_variable, key-number, key-value[[, match-type]]
    [[, ERROR := error-recovery]] );
```

file_variable
The name of the file variable associated with the file to be searched.

key-number
A positive integer expression that indicates the key position.

key-value
An expression that indicates the key to be found; it must be assignment
compatible with the key field in the specified key position.

match-type
An identifier that indicates the relationship between the key value in the
FINDK procedure call and the key value of a component.

error-recovery
The action to be taken if an error occurs during execution of the routine.

When you establish key fields with the KEY attribute, you assign each one a
key number from 0 to 254. Key number 0 represents the mandatory primary
key of the file. Separate indexes are built for each key number in the file.

The key value and the match type provide information about the key to be found. The key value must be assignment compatible with the key fields of the key number being searched. The match type must be one of the following identifiers:

- EQL—equal to the key value
- NXT—the next key in the collating sequence after the key value
- NXTEQL—the next or equal key in the collating sequence after the key value

If the FINDK procedure was used on an ascending collating sequence, NXT and NXTEQL would be equivalent to GTR and GEQ. If a descending collating sequence was used, it would be the same as LSS and LEQ. The match type is optional; if omitted, it defaults to EQL.

The FINDK procedure can be called for any indexed file opened for keyed access, regardless of the file's mode. If the component described exists, the file buffer is filled with that component; UFB and EOF both become FALSE. The mode is set to inspection and the component is automatically locked. If no component is found to match the description, UFB becomes TRUE and EOF is undefined. The mode is set to undefined.

Consider the following:

```
FINDK( Book_Index, 1, 35, NXTEQL );
```

Assuming key number 1 is ascending, this procedure searches the index for key number 1 in the file Book_Index until it finds the first component whose key value is greater than or equal to 35. If the component matching the description in the FINDK statement is found, UFB(Book_Index) and EOF(Book_Index) return FALSE, and the component is locked. If the component cannot be found, UFB(Book_Index) returns TRUE, and EOF(Book_Index) is undefined. Book_Index must be an indexed file opened for keyed access.

For More Information:

- On indexed files (Section 9.1.3)
- On random access by key (Section 9.3.2)
- On the UFB function (Section 9.6.21)
- On the error processing parameter (Section 9.6.26.1)

9.6.8 GET Procedure

The GET procedure advances the file position and reads the next component of the file into the file buffer variable. If the file has relative or indexed organization, the component is also locked to prevent access by other processes.

```
GET( file_variable [[, ERROR := error-recovery]] );
```

file_variable
The name of the file variable associated with the input file.

error-recovery
The action to be taken if an error occurs during execution of the routine.

Before the GET procedure is used for the first time to read one or more file components, the file must be in inspection mode and prepared for reading input. Depending on the access method specified when the file was opened, you can prepare the file for input in the following ways:

*   If the file is open for sequential access, call the RESET procedure. RESET sets the mode to inspection, advances the file position to the first component, and assigns the component's value to the file buffer variable.

*   If the file is open for direct access, call either the RESET or the FIND procedure to position the file.

*   If the file is open for keyed access, call the FINDK, RESET, or RESETK procedure to position the file.

As a result of the GET procedure, the file remains in inspection mode, and the file position advances to the next component. If a component is found other than the end-of-file marker, the component is locked, EOF is set to FALSE, the file buffer variable takes on the value of the component, and UFB is set to FALSE. If a component is not found or the end of the file is reached, EOF and UFB are set to TRUE. If the GET procedure fails, UFB is set to TRUE and EOF becomes undefined. The following example shows the use of the GET procedure:

```
RESET( Books );
New_Rec := Books^;
GET( Books );
```

After execution of the RESET procedure, the value of the file buffer variable Books^ is equal to the value of the first component of the file. The assignment statement assigns this value to the variable New_Rec. The GET procedure then assigns the value of the second component to Books^, advancing the file position to the second component. Another GET procedure

advances the file position to the third component. Figure 9–10 illustrates this sequence of events.

Figure 9–10: File Position After GET Procedure
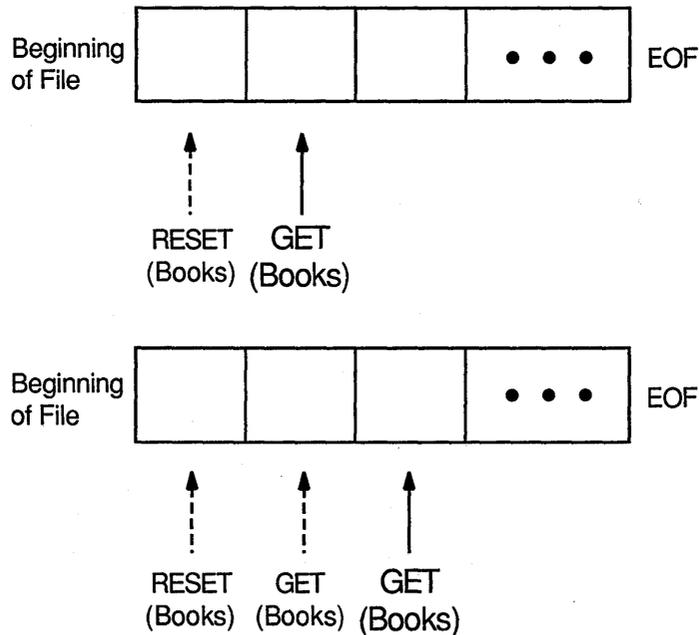
ZK–0103–GE

By using the GET procedure repeatedly, you can read sequentially through a file. When called for a file with relative organization, GET skips any nonexistent components to find the next component.

When you reach the end of the file and EOF returns TRUE, any GET procedure used results in a run-time error.

Consider the following:

```
GET( Phones );
```

This example reads the next component of the file Phones into the file buffer variable Phones^. Prior to executing GET, the value of EOF (Phones) must be FALSE; if it is TRUE, an error occurs.

For More Information:

- On component access (Section 9.3)
- On the UFB function (Section 9.6.21)
- On the error processing parameter (Section 9.6.26.1)

9.6.9 LINELIMIT Procedure

The LINELIMIT procedure stops execution of the program after a specified number of lines has been written into a TEXT file.

LINELIMIT(file_variable, n [[, ERROR := error-recovery]]);

file_variable
The name of the file variable associated with the TEXT file to which this limit applies.

n
A positive integer expression that indicates the number of lines that can be written to the file before execution terminates.

error-recovery
The action to be taken if an error occurs during execution of the routine.

The file can be in any mode before LINELIMIT is called; the file mode does not change after LINELIMIT has been executed.

VAX Pascal first uses environment-specific means to determine if there is a default line limit. If there is no environment-specific default, there is no default line limit. You can use a call to LINELIMIT to override the default.

After the number of lines written into the file has reached the line limit, program execution terminates unless the WRITELN procedure that exceeded the line limit includes the ERROR := CONTINUE parameter.

Consider the following:

LINELIMIT(Debts, 100);

Execution of the program terminates after 100 lines have been written into the text file Debts.

For More Information:

- On TEXT files (Section 9.5)
- On the error processing parameter (Section 9.6.26.1)
- On environment specific line limits (*VAX Pascal Reference Supplement for VMS Systems*)

9.6.10 LOCATE Procedure

The LOCATE procedure positions a random-access file at a particular component so that the next PUT procedure can modify that component.

```
LOCATE( file_variable, component-number [[, ERROR := error-recovery]] );
```

file_variable
The name of the file variable associated with the file to be positioned.

component-number
A positive integer expression that indicates the relative component number of the component to be found.

error-recovery
The action to be taken if an error occurs during execution of the routine.

The file can be in any mode before LOCATE is called. The mode is set to generation after the procedure's execution.

The LOCATE procedure positions the file so that the next PUT procedure writes the contents of the file buffer into the selected component. After LOCATE has been performed, UFB returns TRUE and EOF is undefined.

Consider the following:

```
LOCATE( Accounts_Receivable, 63 );
Accounts_Receivable^ := Next_Account;
PUT( Accounts_Receivable );
```

The LOCATE procedure positions the file Accounts_Receivable before relative component number 63. The call UFB(Accounts_Receivable) now returns TRUE and EOF(Accounts_Receivable) is undefined. The assignment statement loads the file buffer with the contents of file position 63. The PUT operation writes the file buffer into file component number 63. UFB(Accounts_Receivable) remains TRUE.

For More Information:

- On relative files (Section 9.1.2)
- On random access by relative component number (Section 9.3.2)
- On the UFB function (Section 9.6.21)
- On the error processing parameter (Section 9.6.26.1)

9.6.11 OPEN Procedure

The OPEN procedure opens a file and allows you to specify file characteristics.

```
1. OPEN( file_variable
            ,[[file_name]]
            ,[[history]]
            ,[[record_length]]
            ,[[access_method]]
            ,[[record_type]]
            ,[[carriage_control]]
            ,[[organization]]
            ,[[disposition]]
            ,[[file_sharing]]
            ,[[user_action]]
            ,[[default_file_name]]
            ,[[ERROR := error_recovery]] )


2. OPEN( FILE_VARIABLE := file_variable
            [[,FILE_NAME := file_name]]
            [[,HISTORY := history]]
            [[,RECORD_LENGTH := record_length]]
            [[,ACCESS_METHOD := access_method]]
            [[,RECORD_TYPE := record_type]]
            [[,CARRIAGE_CONTROL := carriage_control]]
            [[,ORGANIZATION := organization]]
            [[,DISPOSITION := disposition]]
            [[,SHARING := file_sharing]]
            [[,USER_ACTION := user_action]]
            [[,DEFAULT := default_file_name]]
            [[,ERROR := error_recovery]] ... )
```

file_variable
no default
The name of the file variable associated with the file that VAX Pascal is to open.

file_name
environment specific (default)
A character-string expression containing the external file name. VAX Pascal determines the default file name according to the environment in which you are programming.

history
NEW (default for OPEN/REWRITE openings)
OLD (default for EXTEND/RESET openings)
A value that indicates whether the file exists or if VAX Pascal must create the file. If you specify OLD and if VAX Pascal cannot find the file, an error occurs. If you specify READONLY, you can only read from the file; if you attempt to write to the file, an error occurs. If you specify UNKNOWN, VAX Pascal looks for an existing file but creates a new file if an exisiting file does not exist. If you specify OLD or UNKNOWN and if the attempt to open the file generates a file protection error, VAX Pascal tries again using READONLY.

record_length
255 bytes (default for TEXT and FILE OF VARYING)
ignored (default for other file types)
A positive integer that specifies the maximum size in bytes for a line in a TEXT file or a file of type FILE OF VARYING. ("Record" length is equivalent to "component" length.) The default is 255 bytes. For all other types of files, VAX Pascal ignores this parameter.

If you do not specify a length for an existing file, VAX Pascal uses the length specified at the file's creation.

If you use OPEN to create a sequentially organized file with variable-length components, VAX Pascal records the maximum length of each component in the file only if you specify a value for the record_type field.

access_method
SEQUENTIAL (default)
A value that specifies the component access method to use. The possible values include SEQUENTIAL, DIRECT, and KEYED. The DIRECT access method is equivalent to random access by relative component number. The KEYED access method is equivalent to random access by key.

record_type
VARIABLE (default for new TEXT and VARYING OF CHAR
FIXED (default for other new files)
A value that indicates the component format. ("Record" format and
"component" format are equivalent.) The available values are FIXED
(fixed-length components), VARIABLE (variable-length components),
STREAM (stream component format with either carriage return,
combination carriage return and line feed, or form feed delimiters),
STREAM_CR (stream component format with carriage return delimiters),
and STREAM_LF (stream component format with line feed delimiters).

carriage_control
LIST (default for TEXT and VARYING OF CHAR files)
NONE (default for all other file types)
A value that indicates the carriage control format for the file. The value
LIST indicates single spacing between components. The values CARRIAGE
and FORTRAN are equivalent and indicate that the first character of
every output line is a carriage control character. The values NONE and
NOCARRIAGE indicate that the file has no carriage control.

organization
SEQUENTIAL (default for new files)
A value that specifies the file organization. If you are accessing an existing
file, the specified organization must match the organization of the existing
file; if it does not, an error occurs. The choices for this parameter are
SEQUENTIAL, RELATIVE, and INDEXED.

disposition
SAVE (default for external files)
DELETE (default for internal files)
A value that indicates what VAX Pascal should do with the file after you
close the file. Dispositions are as follows:

Disposition	Discription
SAVE	VAX Pascal retains the file.
DELETE	VAX Pascal deletes the file.
PRINT	VAX Pascal prints the file on a line printer and retains the file.
PRINT_DELETE	VAX Pascal prints the file on a line printer and then deletes the file.

Disposition	Discription
SUBMIT	VAX Pascal submits to a queue or places the print job in a background process and retains the file.
SUBMIT_DELETE	VAX Pascal submits to a queue or places the print job in a background process and deletes the file.

sharing
READONLY (default for HISTORY := READONLY)
NONE (default for other histories)
A value that specifies whether another program can access the file while it is open. A value of READONLY indicates that other programs can read but not write to the file. A value of READWRITE indicates that a program can both read and write to the file while it is open. A value of NONE indicates that a program cannot read or write from the open file.

default
no default
A string expression containing default file specification information. For instance, you can use this value to set a default directory specification.

user_action
no default
A name of a user-written routine that VAX Pascal calls to open the file (instead of allowing VAX Pascal to open the file with the OPEN procedure). You can use a user-action routine to open the file using environment-specific capabilities of the I/O system underlying VAX Pascal.

error_recovery
stops execution after first error (default)
The action to be taken if an error occurs during execution of the routine.

Using the OPEN procedure:

Before the OPEN procedure is called, the file is in undefined mode; its mode does not change after OPEN has been executed.

You cannot use OPEN on a file variable that is already open.

If you use INPUT and OUTPUT, VAX Pascal implicitly opens them just before their first use. VAX Pascal implicitly opens INPUT with a history of READONLY. If you choose, you can explicitly open INPUT and OUTPUT; to do this, call the OPEN procedure at any point in your compilation unit before you use the first I/O routine on that file.

Because the RESET, REWRITE, and EXTEND procedures implicitly open files, you need not always use the OPEN procedure. RESET, REWRITE, and EXTEND impose the same defaults as OPEN, except where noted (in the HISTORY parameter).

You must use the OPEN procedure to do the following:

- Create a TEXT file with fixed-length components
- Create a file with relative or indexed organization
- Open a file for direct or keyed access
- Specify a line length other than the default for a line in a TEXT file

Consider the following:

```
PROGRAM Main( User_Guide );
VAR
    User_Guide : TEXT;
{In the executable section:}
OPEN( User_Guide );
```

When the OPEN procedure is executed, the system first attempts to find an environment-specific translation for User_Guide. If no such translation happens, the system creates the file USER_GUIDE.DAT in the default device and directory on the local computer. If User_Guide had not been specified as an external file in the program header, the OPEN procedure would have created an internal file. By default, the file is created with a record length of 255 bytes and components of variable length. The system then opens the file for sequential access.

Consider the following:

```
OPEN( Journal_Accounts,
      'JOURNAL.DAT',
      HISTORY := UNKNOWN,
      ACCESS_METHOD := KEYED,
      ORGANIZATION := INDEXED );
```

If the file JOURNAL.DAT already exists, this procedure opens it; otherwise, VAX Pascal creates a new file named JOURNAL.DAT with the specified characteristics. If the file does exist, it must have the same characteristics as those in the parameter list of the OPEN procedure. VAX Pascal opens the file with indexed organization for keyed access.

For More Information:

- On file organizations (Section 9.1)
- On component format (Section 9.2)
- On component access (Section 9.3)
- On carriage control (Section 9.5.1)
- On the error processing parameter (Section 9.6.26.1)
- On default file names (*VAX Pascal Reference Supplement for VMS Systems*)

9.6.12 PAGE Procedure

The PAGE procedure skips from the current page to the next page of a TEXT file.

PAGE(file_variable [[, ERROR := error-recovery]]);

file_variable
The name of the file variable associated with a TEXT file.

error-recovery
The action to be taken if an error occurs during execution of the routine.

The file must be in generation mode before the PAGE procedure is called; the mode does not change as a result of the procedure's execution.

Execution of the PAGE procedure clears the record buffer, if it contains data, by performing a WRITELN procedure, and then advances the output to a new page of the specified TEXT file. The next component written to the file begins on the first line of a new page. You can use this procedure only on TEXT files. If you specify a file of any other type, an error occurs.

The value of the page eject component that is output to the file depends on the carriage control format for that file. When CARRIAGE or FORTRAN is enabled, the page eject record is equivalent to the carriage control character '1'. When LIST, NOCARRIAGE, or NONE is enabled, the page eject record is a single form feed character.

Consider the following:

PAGE(User_Guide);

This PAGE procedure causes a page eject record to be written in the text file User_Guide.

For More Information:

- On TEXT files (Section 9.5)
- On the error processing parameter (Section 9.6.26.1)

9.6.13 PUT Procedure

The PUT procedure adds a new component to a file.

PUT(file_variable [[, ERROR := error-recovery]]);

file_variable
The name of the file variable associated with the output file.

error-recovery
The action to be taken if an error occurs during execution of the routine.

Before executing the first PUT procedure on a file opened for sequential access, you must execute an EXTEND, REWRITE or TRUNCATE procedure to set the file to generation mode. EXTEND, REWRITE and TRUNCATE set EOF to TRUE, thus preparing the file for output. (TRUNCATE is legal only on files with sequential organization.) If the file has indexed organization, the components to be written must be ordered by the primary key.

Before executing the first PUT statement on a file opened for direct access, you must execute an EXTEND, REWRITE or LOCATE procedure to position the file.

The PUT procedure writes the value of the file buffer variable at the end of the specified sequential-file or direct-access file. You can use LOCATE to position a direct-access file and then use PUT to write the value of the file buffer variable at that position. After execution of the PUT procedure, the value of the file buffer variable becomes undefined (UFB returns TRUE). EOF remains TRUE and the file remains in generation mode.

You can call the PUT procedure for a keyed-access file, regardless of the file's mode (inspection, generation, or undefined). PUT causes the file buffer variable to be written to the file at the position indicated by the key. If the component has more than one key, the file buffer variable is inserted in each index at the appropriate location. After execution of PUT, a keyed-access file is in generation mode.

Consider the following:

```
PROGRAM Book_File( INPUT, OUTPUT, Books );
TYPE
   My_String = PACKED ARRAY[1..40] OF CHAR;
   Book_Rec  = RECORD
               Author : My_String;
               Title  : My_String;
               END;
VAR
   New_Book : Book_Rec;
   Books : FILE OF Book_Rec;
   n : INTEGER;
{In the executable section:}
REWRITE( Books );
FOR n := 1 TO 10 DO
   BEGIN
   WITH New_Book DO
      BEGIN
      WRITE( 'Title:' );
      READLN( Title );
      WRITE( 'Author:' );
      READLN( Author );
      END;
   Books^ := New_Book;
   PUT( Books );
   END;
CLOSE( Books );
```

This program writes the first 10 components read from the terminal into the file Books. The component data items are typed at the terminal and assigned to the record variable New_Book. They consist of two 40-character strings denoting a book's author and title. The FOR loop accepts 10 values for New_Book, assigning each new record to the file buffer variable Books^. The PUT statement writes the value of Books^ into the file for each input record.

For More Information:

- On component access (Section 9.3)
- On the UFB function (Section 9.6.21)
- On the error processing parameter (Section 9.6.26.1)

9.6.14 READ Procedure

The READ procedure reads one or more file components into a variable.

READ([[file_variable,]] {variable-identifier [[:radix-specifier]]},...
    [[, ERROR := error-recovery]]);

file_variable
The name of the file variable associated with the input file. If you omit the name of the file, the default is INPUT.

variable-identifier
The name of the variable into which a file component will be read; multiple identifiers must be separated with commas.

radix-specifier
One of the format values BIN, OCT, or HEX. These values, when used on a variable identifier, will read the variable in binary, octal, or hexadecimal radix respectively. You can use a radix specifier only when reading from a TEXT file.

error-recovery
The action to be taken if an error occurs during execution of the routine.

The file must be in inspection mode before READ is called. The file remains in inspection mode after execution of a READ procedure.

By definition, the READ procedure for a nontext file performs an assignment statement, a GET procedure, and an UNLOCK procedure for each variable. Consider the following:

```
{This call to READ...}
READ( file_variable, variable-identifier );

{...is equivalent to the following code:}
variable-identifier := file_variable^;
GET( file_variable );
UNLOCK( file_variable );
```

The READ procedure reads from the file until it has found a value for each variable in the list. The first value read is assigned to the first variable in the list, the second value read is assigned to the second variable, and so on. The values and the variables must be of assignment-compatible types. Reading stops if an error occurs.

For a TEXT file, more than one component (character) can be read into a single variable. For example, many characters can be read into a string or converted into a numeric variable. The READ procedure repeats the assignment, GET, and UNLOCK process until it has read a sequence of characters that represent a legal value for the next variable in the parameter list. The procedure continues to read components from the file until it has assigned a value to each variable in the list.

After the last character has been read from a line of a TEXT file, EOLN returns TRUE and the file buffer variable contains a space. Unless you are reading into a character or string variable, a call to READ at this point skips over the end-of-line marker and positions the file at the beginning of the next line. If you are reading into a variable of type CHAR when EOLN returns TRUE, the space is read and assigned to the variable, and the file position advances. If you are reading into a string variable when EOLN becomes TRUE, the file position does not change. In the latter case, you should use the READLN procedure to advance the file position past the end-of-line marker.

Values from a TEXT file can be read into variables of integer, real, Boolean, character, string, and enumerated types. TEXT file values to be read into integer, real, Boolean, and enumerated variables can be preceded in the file by any number of spaces, tabs, and end-of-line markers. Values to be read into character variables, however, must not be separated because they are read and assigned character by character.

In a TEXT file, when VAX Pascal encounters a character that forms an object of a data type that does not match the data type of the parameter, reading stops. Consider the following:

```
VAR
   i : INTEGER;
{In the executable section:}
READ( i );
```

If the object in the input file is 123ABC, the read stops at the character 'A', and i contains the value 123.

When reading constant identifiers of an enumerated type from a TEXT file, VAX Pascal reads all characters in the identifier, but recognizes only the first 31 characters. You need input only enough characters to make the identifier unique among the other constant identifiers of its type. Text input data for enumerated types can consist of both lowercase and uppercase characters.

Boolean input data in TEXT files follow the same rules as other enumerated types. For example, the following character combinations, all of which could appear in a TEXT file, are equivalent: TRUE, True, T, t, tr.

When using a radix specifier, values from a TEXT file can be read into a variable of any type, except a type containing a file component. If the input stream does not provide suffcient data, the high-order bits are set to zero. When reading structured types, the input stream must account for any padding required for alignment.

You can use the READ procedure to read a sequence of characters from a TEXT file into a variable of type PACKED ARRAY OF CHAR. Successive characters from the file are assigned to components of the array, in order, until each component has been assigned a value. If any characters remain on the line after the array is full, the next READ procedure begins with the next character on that line. If the end of the line is encountered before the array is full, spaces are assigned to the remaining components.

You can also read TEXT file characters into a variable of types STRING or VARYING OF CHAR. Characters are assigned to a STRING or VARYING OF CHAR variable in a manner similar to that in which they are assigned to a packed array. However, if the end-of-line marker is encountered before the STRING or VARYING OF CHAR variable has been filled to its maximum length, the STRING or VARYING OF CHAR value is not padded with spaces. Instead, its current length is set equal to the number of characters that have been read into it. If you call the READ procedure with a parameter of type STRING or VARYING OF CHAR, and EOLN returns TRUE, no characters are read into the STRING or VARYING OF CHAR variable; its current length is set to zero.

Every nonempty TEXT file ends with an end-of-line marker and an end-of-file marker. Therefore, EOF never becomes TRUE when you are reading strings with the READ procedure. To test EOF when reading strings, use a READLN procedure to advance the file beyond the end-of-line marker.

Consider the following:

```
READ( Temp, Age, Weight );
```

Assume that Temp, Age, and Weight are real variables, and that the following values have been entered at the terminal:

```
98.6 11 75
```

The variable Temp is assigned the value 98.6, Age is assigned the value 11.0, and Weight is assigned the value 75.0. You need not type all three values on the same line.

Consider the following:

```
TYPE
    A_String = PACKED ARRAY[1..20] OF CHAR;
VAR
    Names      : TEXT;
    Pres, Veep : A_String;
{In the executable section:}
READ( Names, Pres, Veep );
```

This program fragment declares and reads the file Names, which contains the following character strings:

```
John F. Kennedy     Lyndon B. Johnson   Lyndon B. Johnson   <EOLN>
Hubert H. Humphrey  <EOLN>
Richard M. Nixon    Spiro T. Agnew      <EOLN>
```

The first call to the READ procedure sets Pres equal to the 20-character string ' John F. Kennedy   ' and Veep equal to ' Lyndon B. Johnson   '. The second call to the procedure assigns the value ' Lyndon B. Johnson   ' to Pres and, after encountering the end-of-line marker, fills the array Veep with spaces. The file position does not advance to the beginning of the next line until a READLN is performed.

Consider the following:

```
TYPE
    Color = ( Red, Fire_Engine_Green, Blue, Black );
VAR
    Light : Color;
{In the executable section:}
READ( Light );
```

In this example, if the letter R is read, the variable Light is assigned the value Red. However, if the letters Redx are read, an error occurs. If the letters Bl are read, an error also occurs because Bl is not unique. However, the letters Blu are unique and would be interpreted as the constant identifier Blue.

For More Information:

*   On TEXT files (Section 9.5)

*   On the error processing parameter (Section 9.6.26.1)

*   On specifying radixes (Section 9.6.26.3)

9.6.15   READLN Procedure

The READLN procedure reads lines of data from a TEXT file.

READLN [[([[file_variable,]] {variable-identifier [[:radix-specifier]]},...
    [[, ERROR := error-recovery]])]];

file_variable
The name of the file variable associated with the TEXT file to be read. If you omit the name of the file, the default is INPUT.

variable-identifier
The name of the variable into which a value will be read; multiple identifiers must be separated with commas. If you do not specify any variable names, READLN skips a line in the specified file.

radix-specifier
One of the format values BIN, OCT, or HEX. These values, when used on a variable identifier, read the variable in binary, octal, or hexadecimal, respectively. You can use a radix specifier only when reading from a TEXT file.

error-recovery
The action to be taken if an error occurs during execution of the routine.

The file must be in inspection mode before READLN is called; it remains in that mode after the procedure's execution.

The READLN procedure reads values from a TEXT file. After reading values for all the listed variables, the READLN procedure skips over any characters remaining on the current line and positions the file at the beginning of the next line. The values need not all be on a single line; READLN continues until values have been assigned to all the specified variables, even if this process results in the reading of several lines of the input file.

When applied to several variables, READLN performs the following sequence:

```
READ( file_variable, {variable-identifier},... );
READLN( file_variable );
```

EOLN returns TRUE after a READLN procedure only if the new line is empty.

You can use the READLN procedure to read integers, real numbers, Booleans, characters, strings, and constants of enumerated types. The values in the file must be separated as for the READ procedure. The rules governing the reading of values from text files are presented with the READ procedure.

Consider the following:

```
TYPE
    String = PACKED ARRAY[1..20] OF CHAR;
VAR
    Names       : TEXT;
    Pres, Veep : String;
{In the executable section:}
READLN( Names, Pres, Veep );
```

This program fragment declares and reads the file Names, which contains the following characters:

```
John F. Kennedy      Lyndon B. Johnson    Lyndon B. Johnson    <EOLN>
Hubert H. Humphrey   <EOLN>
Richard M. Nixon     Spiro T. Agnew       <EOLN>
<EOLN>
<EOF>
```

The READLN procedure reads the values 'John F. Kennedy   ' for Pres and 'Lyndon B. Johnson   ' for Veep. It then skips to the next line, ignoring the remaining characters on the first line. Subsequent execution of the procedure assigns the value 'Hubert H. Humphrey   ' to Pres and the space detected as the end-of-line marker to Veep. A third call to the procedure reads 'Richard M. Nixon   ' into Pres and 'Spiro T. Agnew   ' into Veep. The procedure then skips past the end-of-line marker to the beginning of the next line. If you call READLN again, EOF becomes TRUE, and EOLN becomes undefined.

For More Information:

- On the READ procedure (Section 9.6.14)

- On the error processing parameter (Section 9.6.26.1)

- On the radix specifiers (Section 9.6.26.3)

9.6.16   RESET Procedure

The RESET procedure readies a file for reading.

RESET(file_variable [[, ERROR := error-recovery]]);

file_variable
The name of the file variable associated with the input file.

error-recovery
The action to be taken if an error occurs during execution of the routine.

The file can be in any mode before you call RESET; a call to RESET sets the file to inspection mode. If the file is an external file and is not already open, RESET opens it using the same defaults as the OPEN procedure. You cannot use RESET to create a file.

After execution of RESET, the file is positioned at the first component, and the file buffer variable contains the value of this component. If the file is not empty, EOF and UFB return FALSE and the first component is locked to prevent access by other processes. If the file is empty, EOF and UFB return

TRUE. If the file does not exist, RESET does not create it, but returns an error at run time.

You should call RESET before reading any file with sequential organization except the predeclared file INPUT. The RESET procedure removes the end-of-file marker from any file connected to a terminal device (including INPUT), thus allowing reading from the file to continue. If you call RESET for the predeclared file OUTPUT, an error occurs.

A call to RESET on a relative file opened for direct access positions the file at its first existing component.

A call to RESET on an indexed file opened for keyed access positions the file at the first component relative to the primary key.

Consider the following:

```
VAR
    f : FILE OF INTEGER;
{In the executable section:}
OPEN( f , 'file.dat', ACCESS_METHOD := DIRECT );
RESET( f );
```

These statements open the file variable f for direct access. After execution of the OPEN and RESET procedures, you can use the FIND procedure for direct access to the components of the file.

```
RESET( Weights );
```

If the file variable Weights is already open, this procedure call prepares it for reading and assigns the value of the first file component to Weights^. If the file is not open, RESET causes VAX Pascal to open the file by default. If Weights is an external file, its file history will be OLD. Otherwise, an error occurs.

For More Information:

* On component access (Section 9.3)
* On the default parameter values for OPEN (Section 9.6.11)
* On the error processing parameter (Section 9.6.26.1)

9.6.17 RESETK Procedure

The RESETK procedure, like the RESET procedure, readies a file for reading.

```
RESETK( file_variable, key-number[[, ERROR := error-recovery]] );
```

file_variable
The name of the file variable associated with the input file.

key-number
A nonnegative integer expression that indicates the key position.

error-recovery
The action to be taken if an error occurs during execution of the routine.

The file can be in any mode before RESETK is called to set the mode to inspection.

RESETK can be applied only to indexed files opened for random access by key. You assign a key number from 0 to 254 to each key field of a file component with the KEY attribute. The file is searched for the component with the lowest value in the specified key number. This component becomes the current component in the file and is locked. The value of the current component is copied into the file buffer; EOF and UFB are set to FALSE. If the component does not exist, EOF and UFB become TRUE. Note that a RESETK procedure on key number 0 is equivalent to a RESET procedure.

Consider the following:

```
RESETK( Book_Index, 0 );
```

This procedure searches the file Book_Index for the component with the lowest value in the primary key. If this component exists, it becomes the current file component and is locked. The function calls UFB(Book_Index) and EOF(Book_Index) return FALSE. If the procedure was unable to find the component, UFB(Book_Index) and EOF(Book_Index) return TRUE.

For More Information:

- On indexed files (Section 9.1.3)
- On random access by key (Section 9.3.2)
- On the UFB function (Section 9.6.21)
- On the error processing parameter (Section 9.6.26.1)

).6.18 REWRITE Procedure

The REWRITE procedure readies a file for output.

```
REWRITE( file_variable [[, ERROR := error-recovery]] );
```

file_variable
The name of the file variable associated with the output file.

error-recovery
The action to be taken if an error occurs during execution of the routine.

The file can be in any mode before REWRITE is called to set the mode to generation. If the file variable has not been opened, REWRITE creates and opens it using the same defaults as the OPEN procedure.

The REWRITE procedure truncates a file to length zero and sets EOF and UFB to TRUE. You can then write new components into the file with the PUT, WRITE, and WRITELN procedures (WRITELN is defined only for text files). After the file is open, successive calls to REWRITE truncate the existing file to a length of zero; they do not create new versions of the file.

To update an existing file with sequential organization, you must either use the EXTEND procedure, use the TRUNCATE procedure, or copy the contents to another file, specifying new values for the components you need to update.

When applied to a file with relative or indexed organization, REWRITE deletes the contents of the file and sets the file position to the beginning of an empty file.

Consider the following:

```
REWRITE( Storms );
```

If the file variable Storms is already open, this REWRITE procedure prepares the file for writing, clears it of old data, and sets the file position to the beginning of the file. If Storms is not open, a new version is created with the same defaults as for the OPEN procedure.

Consider the following:

```
VAR
   Ratings : FILE OF INTEGER;
{In the executable section:}
OPEN( Ratings, 'cars.dat', HISTORY := OLD, RECORD_TYPE := FIXED );
REWRITE( Ratings );
```

The OPEN procedure opens the file variable Ratings, which is associated with the file CARS.DAT. The REWRITE procedure discards the current contents of the file f and sets the file position to the beginning of the file. After execution of this procedure, EOF(Ratings) returns TRUE.

For More Information:

* On component access (Section 9.3)
* On the default parameters for OPEN (Section 9.6.11)
* On the error processing parameter (Section 9.6.26.1)

9.6.19 STATUS Function

The STATUS function indicates the status of a file following the last operation performed on it.

STATUS(file_variable)

file_variable
The name of the file variable associated with the file to be tested.

The file can be in any mode before STATUS is called; unless an error occurs, STATUS does not change the file mode upon execution.

The STATUS function returns one of the following integer codes that indicate the previous operation's effect on the file:

Code	Description
0	Successful operation
-1	End-of-file encountered
Positive integer[1]	Error encountered

[1] The actual number is environment-specific and indicates the exact error that occured.

A test by the STATUS function on a TEXT file causes delayed device access to occur, thus filling the file buffer with the next file component. Therefore, EOF, EOLN, UFB, and STATUS never return an error code following a successful STATUS function call.

Consider the following:

```
RESET( File1, ERROR := CONTINUE );
IF STATUS( File1 ) > 0 THEN WRITELN( 'Cannot access first record' )
ELSE
    IF STATUS( File1 ) < 0 THEN WRITELN( 'File is empty' )
    ELSE READ( File1 );
```

If the RESET procedure encounters either an error condition or an end-of-file, an appropriate error message is displayed. If the STATUS function indicates that the RESET procedure was successful, the first record is read from the file.

For More Information:

- On TEXT files (Section 9.5)
- On the error processing parameter (Section 9.6.26.1)
- On delayed device access (Section 9.5.3)
- On status code translations (*VAX Pascal Reference Supplement for VMS Systems*)

9.6.20 TRUNCATE Procedure

The TRUNCATE procedure indicates that the current file component and all components following it are to be deleted. TRUNCATE can be used only on a file that has sequential organization.

```
TRUNCATE( file_variable [[, ERROR := error-recovery]] );
```

file_variable
The name of the file variable associated with the file to be truncated.

error-recovery
The action to be taken if an error occurs during execution of the routine.

The file must be in inspection mode before TRUNCATE is called. After the procedure has been executed, the mode is set to generation so that you can write to the file.

After the appropriate components have been deleted, the file remains positioned at the new end-of-file, but the file buffer itself is undefined. Thus, EOF and UFB are both set to TRUE.

Consider the following:

```
TRUNCATE( Master_File );
```

This procedure deletes components from Master_File, beginning with the current component and continuing until EOF returns TRUE. When the operation is complete, EOF(Master_File) and UFB(Master_File) are TRUE, and new data can be written at the end of Master_File.

For More Information:

- On sequential files (Section 9.1.1)
- On the error processing parameter (Section 9.6.26.1)

9.6.21 UFB Function

The UFB (Undefined File Buffer) function returns a Boolean value to indicate whether the last file operation gave the file buffer an undefined status.

UFB(file_variable)

file_variable
The name of the file variable associated with the file whose buffer is being tested.

The file can be in any mode before UFB is called; execution of UFB does not change the file mode.

UFB tests the effect of the last I/O operation done to the file. UFB returns FALSE if a successful GET, FIND, FINDK, RESET, or RESETK operation has filled the file buffer. GET, FIND, FINDK, RESET, and RESETK procedure calls that do not fill the file buffer set UFB to TRUE. UFB also returns TRUE after DELETE, EXTEND, LOCATE, PUT, REWRITE, TRUNCATE, and UPDATE procedures have left the contents of the file buffer unknown.

Assigning a new value to the file buffer with an assignment statement does not change the value of UFB. Consider the following:

```
FIND( Supplies, December );
IF NOT UFB( Supplies ) THEN
   Inventory := Inventory - Supplies^;
```

If the variable December has a value of 12, the FIND procedure attempts to find the twelfth component of the file Supplies. If the FIND procedure is successful, Supplies^ assumes the value of this component and UFB(Supplies) is FALSE. If, however, the FIND procedure is unable to find the twelfth component of the file, UFB(Supplies) returns TRUE. In this example, the value of Supplies^ is subtracted from the value of Inventory only if the FIND procedure is successful.

9.6.22 UNLOCK Procedure

The UNLOCK procedure releases the current file component for access by other processes.

UNLOCK(file_variable [[, ERROR := error-recovery]]);

file_variable
The name of the file variable associated with the file whose component is to
be unlocked.

error-recovery
The action to be taken if an error occurs during execution of the routine.

The file must be in inspection mode before UNLOCK is called; it remains in
inspection mode after UNLOCK has executed.

If the component at which the file pointer is positioned has been locked, the
UNLOCK procedure releases it.

Consider the following:

```
UNLOCK( Sales_File );
```

The UNLOCK procedure releases the contents of the current component.

For More Information:

For information on the error processing parameter, see Section 9.6.26.1.

9.6.23 UPDATE Procedure

The UPDATE procedure writes the contents of the file buffer into the current
component.

```
UPDATE( file_variable[[, ERROR := error-recovery]] );
```

file_variable
The name of the file variable associated with the file whose component is to
be updated.

error-recovery
The action to be taken if an error occurs during execution of the routine.

The file must be in inspection mode before UPDATE is called; it remains in
that mode after the procedure's execution.

The UPDATE procedure is legal for files that have been opened for random
access ("direct" or "keyed"). The current component must already have been
locked by a successful FIND, FINDK, GET, RESET, or RESETK procedure
before the contents of the file buffer can be rewritten into it. After the
update has taken place, the component is unlocked and UFB returns TRUE.

Consider the following:

```
UPDATE( October_Sales );
```

This procedure writes the file buffer contents (October_Sales^) back into the current file component October_Sales. The component is then unlocked and UFB(October_Sales) returns TRUE.

For More Information:

- On component access (Section 9.3)
- On the error processing parameter (Section 9.6.26.1)

9.6.24 WRITE Procedure

The WRITE procedure assigns data to an output file.

```
WRITE([[file_variable, ]]{expression},... [[, ERROR := error-recovery]] )
```

file_variable
The name of the file variable associated with the output file. If you omit the name of the file, the default is OUTPUT.

expression
An expression whose value is to be written; multiple output values must be separated with commas. An output value must have the same type as the file components; however, values written to a TEXT file can also be expressions of any ordinal, real, or string type.

error-recovery
The action to be taken if an error occurs during execution of the routine.

The file (unless it is a random-access by key file) must be in generation mode before WRITE is called; it remains in that mode after WRITE has executed.

By definition, a WRITE statement to a nontext file performs an assignment to the file buffer variable and a PUT statement for each output value. For nontext files, the types of the output values must be assignment compatible with the component type of the file. For example:

```
WRITE( file_variable, expression );
```

This procedure call is similar to the following example:

```
file_variable^ := expression;
PUT( file_variable );
```

For TEXT files, the WRITE procedure converts the value of each expression to a sequence of characters. It repeats the assignment and PUT process until all the values have been written to the file.

Consider the following:

```
TYPE
    String = PACKED ARRAY[1..20] OF CHAR;
VAR
    Names :   FILE OF String;
    Pres  :   String;
{In the executable section:}
WRITE (Names, 'Millard Fillmore    ', Pres);
```

This example writes two components in the file Names. The first is the 20-character string constant 'Millard Fillmore   '. The second is the value of the string variable Pres.

For More Information:

- On TEXT files (Section 9.5)
- On component format (Section 9.2)
- On prompting from the terminal (Section 9.5.2)
- On the error processing parameter (Section 9.6.26.1)

9.6.25 WRITELN Procedure

The WRITELN procedure writes a line of data to a text file.

WRITELN [[([[file_variable,]] {expression},... [[, ERROR := error-recovery]])]]

file_variable
The name of the file variable associated with the text file to be written. If you omit the name of the file, the default is OUTPUT.

expression
An expression whose value is to be written; multiple output values must be separated by commas. The expressions can be of any ordinal, real, or string type and are written with a default field width.

error-recovery
The action to be taken if an error occurs during execution of the routine.

The file must be in generation mode before WRITELN is called; it remains in that mode after WRITELN has been executed.

The WRITELN procedure writes the specified values into the TEXT file, inserts an end-of-line marker after the end of the current line, and then positions the file at the beginning of the next line. When applied to several expressions, WRITELN performs the following sequence:

```
WRITE (file_variable, {expression},....);
WRITELN (file_variable);
```

Consider the following:

```
WRITELN( User_Guide, 'This manual describes how to interact');
```

This procedure writes the string to the TEXT file User_Guide, follows it with an end-of-line marker, and skips to the next line.

You can specify a carriage-control character as the first item in an output line. When you use carriage-control characters, make sure that the file is open with either the CARRIAGE or FORTRAN option. Consider the following:

```
WRITELN( Tree, ' ', String1, String2 );
```

The first item in the list is a space character. The space indicates that the values of String1 and String2 are printed on a new line when the file is written to a terminal, line printer, or similar carriage control device.

If you specify a carriage format but use an invalid carriage control character, the first character in the line is ignored. The output appears with the first character truncated. Consider the following:

```
TYPE
    A_String = PACKED ARRAY[1..25] OF CHAR;
VAR
    New_Hires : TEXT;
    n         : INTEGER;
    New_Rec : RECORD
        Id      : INTEGER;
        Name,
        Address : A_String;
        END;
{In the executable section:}
OPEN( New_Hires, 'new_hires.dat', CARRIAGE_CONTROL := FORTRAN );
REWRITE( New_Hires );
WITH New_Rec DO
    BEGIN
    WRITELN( New_Hires, '1New hire #  ', ID:1, ' is ', Name );
    WRITELN( New_Hires, ' ', Name, 'lives at:' );
    WRITELN( New_Hires, ' ' );
    WRITELN( New_Hires, ' ', Address );
    END;
```

In this example, four lines are written to the TEXT file New_Hires. The output starts at the top of a new page, as directed by the carriage-control character '1', and appears in the following format:

```
New hire #  73 is Irving Washington
Irving Washington       lives at:

22 Chestnut St, Seattle
```

For More Information:

* On TEXT files (Section 9.5)

* On carriage-control characters in new files (Section 9.6.11)

* On formatting output (Section 9.6.26.2)

* On the error processing parameter (Section 9.6.26.1)

9.6.26   Error Processing and Formatting Output

This section contains information on error processing and on formatting written output.

9.6.26.1   Error Processing Parameter

For I/O procedures, the last parameter (which is optional) specifies the action to be taken should the procedure fail to execute successfully. You must use nonpositional syntax in order to pass the error recovery parameter to the called procedure. This parameter is called ERROR and can accept two values: CONTINUE and MESSAGE.

If you specify ERROR := CONTINUE, the program continues to execute regardless of any error conditions encountered during execution of the procedure. If you specify this value, you should use the STATUS function to be certain that the I/O routine worked as expected.

If you specify ERROR := MESSAGE and if an error occurs, VAX Pascal generates an appropriate error message and program execution stops. By default, VAX Pascal displays an error message and program execution stops after the first error in an I/O operation.

You cannot use the error recovery parameter with the I/O functions EOF, UFB, and EOLN, nor with any reference to the file buffer.

9.6.26.2 Output with Specified Field Width

The output values of a WRITE, WRITELN, or WRITEV procedure can be
compile-time or run-time expressions, with values of any ordinal, real, or
string type. Each value is written with a default field width, which specifies
the minimum number of characters to be written for the value. Table 9–5
lists the default field widths.

Table 9–5: Default Field Widths

Type of Item Printed	Number of Characters
INTEGER, UNSIGNED	10
CHAR	1
BOOLEAN	6
Enumerated	Size of longest identifier plus 1, up to 32
REAL	12
DOUBLE	20
QUADRUPLE	40
Character string	Length of string

You can override these defaults for a particular value by specifying a field
width in the print list, using the following format:

output:minimum[[:fraction]]

Both minimum and fraction represent integer expressions with positive or
zero values. The minimum indicates the minimum number of characters to
be written for the value. The fraction, which is permitted only for values of
real types, indicates the number of digits to be written to the right of the
decimal point. The format of the field width specification is identical for the
WRITE, WRITELN, and WRITEV procedures.

By default, real numbers are written in exponential format. Regardless of
the real number's type, output procedures always prefix the exponent with
the letter E. Each real number in exponential format is preceded by a blank
or a minus sign, and the value of the rightmost digit is rounded. Consider
the following:

```
WRITELN( Shoe_Size );
```

If the value of Shoe_Size is 12.5, this procedure produces the following
output:

```
1.25000E+01
```

To write the value in decimal format, you must specify a field width, as in this example:

```
WRITELN( Shoe_Size:5:1 );
```

The first integer indicates that a minimum of five characters will be written. The minimum includes the minus sign, if needed, and the decimal point. The second integer specifies one digit to the right of the decimal point. The resulting output is as follows:

```
12.5
```

If the field specified is wider than necessary, the value is written with leading blanks.

If you try to write an integer, unsigned, or real value in a field that is too narrow, the field width is expanded to the minimum necessary to write the value. If you try to write a value of an enumerated type, a Boolean value, a character, or a string value in a field that is too narrow, the value is truncated on the right. The truncated identifier is not checked for uniqueness.

For an expression of an enumerated type, the constant identifier denoting the expression's value is written. Consider the following:

```
VAR
    Color : ( Blue, Yellow, Black, Fire_Engine_Green );
{In the executable section:}
WRITE( 'My favorite color is ', Color:15 );
```

When the value of Color is Yellow, the following is written:

```
My favorite color is          YELLOW
```

When the value of Color is Fire_Engine_Green, the following appears:

```
My favorite color is FIRE_ENGINE_GRE
```

Because the field width specified in this case is not wide enough for all 17 characters in the identifier, the identifier is truncated after the field is filled. Note that constants of enumerated types are written in all uppercase characters.

9.6.26.3   Writing Binary, Decimal, Unsigned Decimal, Hexadecimal, and Octal Values

You can use the predeclared conversion functions BIN, DEC, UDEC, HEX, and OCT in combination with the WRITE, WRITELN, and WRITEV procedures to write binary, decimal, unsigned decimal, hexadecimal, and octal values. Also, the DEC and UDEC functions return values with leading zeros; by default, the I/O routines use leading blanks with decimal numbers, not leading zeros. Consider the following syntax for the WRITE procedure.

```
                ⎛ BIN  ⎞
                ⎜ DEC  ⎟
WRITE( [[file_variable,]] ⎨ UDEC ⎬ ,...)
                ⎜ HEX  ⎟
                ⎝ OCT  ⎠
```

The predeclared conversion functions convert the value of the first
expression in the list to its equivalent as a binary, decimal, unsigned
decimal, hexadecimal, or octal number. The resulting digits are returned
in a VARYING OF CHAR string.

For every expression whose binary, decimal, unsigned decimal, hexadecimal,
or octal value you wish to write, you must call the appropriate conversion
function separately with an actual parameter list. You can call more than
one conversion function in the same output procedure call. Variables of
any type (including pointers) can be written to text files in binary, decimal,
unsigned decimal, hexadecimal, or octal notation.

You can specify field widths with the conversion functions; however, the
results are not likely to be what you expect. For example, if you want
to convert the value of i to its hexadecimal equivalent and you want the
converted value to be written in a field three characters wide, you might
write the following procedure call:

```
WRITELN( HEX( i ):3 );
```

However, because the converted value is longer than the field width
specification, the value is truncated on the right rather than on the left.
Therefore, the output generated by this procedure would be as follows:

```
00
```

So, you should be careful about specifying field widths with BIN, DEC,
UDEC, HEX, and OCT when the converted value could exceed the field
width given.

Consider the following:

```
WRITE( HEX( Payroll, 10 ), HEX( Salary, 12 ) );
```

The values of the variables Payroll and Salary are converted to their
hexadecimal equivalents. Payroll is printed with 10 characters and Salary
is printed with 12 characters. The output values, preceded by two initial
blanks, could look like this:

```
000031F2    000058AB
```

Consider the following:

```
WRITELN( OCT( Social_Security, 14 ), BIN( Survey, 8 ) );
```

The value of the variable Social_Security is converted to its octal equivalent and printed with 14 characters. Then the value of the variable Survey is converted to its binary equivalent and printed with eight characters. A sample line of output, preceded by three blanks, could look like this:

```
0271137762500101110
```

Consider the following:

```
WRITEV( Final_Balance, OCT( Debits, 16 ), OCT( Credits, 16 ) );
```

The values of the variables Debits and Credits are converted to their octal equivalents and written to the string variable Final_Balance with 16 characters each. The output string, preceded by five blanks, could look like this:

```
'     77777770342     00000033766'
```

For More Information:

- On the WRITEV procedure (Section 8.87)
- On the BIN function (Section 8.7)
- On the DEC function (Section 8.21)
- On the UDEC function (Section 8.77)
- On the HEX function (Section 8.36)
- On the OCT function (Section 8.52)

Chapter 10

Attributes

An **attribute** is an identifier that directs the VAX Pascal compiler to change its behavior in some way. This chapter discusses the following information about attributes:

- Attribute syntax (Section 10.1)
- Attributes (Section 10.2)
- Attribute classes (Section 10.3)

When an attribute is not explicitly stated, the compiler follows the default rules to assign properties to program elements. However, using attributes to override the defaults allows additional control over the properties of data items, routines, and compilation units.

For convenience in description, the attributes are grouped in **attribute classes**. All attributes in a given class share common characteristics (sometimes there is only one attribute to a class).

For More Information:

For information on environment-specific issues about attributes, see the *VAX Pascal Reference Supplement for VMS Systems*.

10.1 Attribute Syntax

The following syntax applies to all VAX Pascal attributes:

[{identifier1 [[($\left\{ \begin{array}{l} \text{constant-expression} \\ \text{identifier2} \end{array} \right\}$,...)]] },...]

identifier1
The name of the attribute.

constant-expression
A compile-time integer expression, represented in this chapter by n, that qualifies several of the VAX Pascal attributes.

identifier2
The name of an option available in one of the following instances:

- With the CHECK, OPTIMIZE, or KEY attributes

- With COMMON and PSECT attributes, indicating the name of a storage area

- With the GLOBAL, EXTERNAL, WEAK_GLOBAL, and WEAK_EXTERNAL attributes, indicating an external name

A list of attributes can appear anywhere in the VAR, TYPE, and CONST declaration sections, and anywhere in a program that a type, a type identifier, or the heading of a routine or compilation unit is legal. However, only one attribute from a particular class can appear in a given attribute list. The use of attribute lists is illustrated in examples throughout this chapter. The names of attributes, when used in a suitable context, cannot conflict with other identifiers with the same name in the program.

Syntactically, an attribute list can appear before a VAR, TYPE, and CONST section in the declaration section. In this case, the attributes would apply to all elements in that particular section. However, at this time, VAX Pascal only allows you to use the HIDDEN attribute in this way.

Some attributes require a special form of constant expression called a **name string**. The syntax of a name string differs from that of other strings in VAX Pascal only in that a name string cannot use the extended-string syntax.

Every program element must be associated with one property for each applicable attribute class. The VAX Pascal compiler automatically supplies the defaults for the unspecified classes at the time of the element's declaration. In some classes, as described in the following sections, the default property is not available through an explicit attribute.

Attributes can be associated with data items in the following ways:

- By appearing in a type definition in a TYPE section; the item is later declared to be of that type.

- By appearing in the declaration of an item preceding its type.

• By appearing before the current declaration section.

NOTE

In VAX Pascal, the presence of constant expressions and attribute lists leads to a minor ambiguity in the language syntax. If the compiler finds a left bracket ([) symbol when it expects to find a type or type identifier, it always assumes that the bracket indicates the beginning of an attribute list. The ambiguity arises because the left bracket could also represent the beginning of a set constructor that denotes the low bound of a subrange type. If the latter case is in fact what you intend, enclose the set constructor in parentheses; the compiler will interpret the expression correctly. For example:

```
TYPE  X = (([1] <= [2])..True;
```

When a type definition includes a list of attributes, the type has only those attributes specified. The compiler does not supply the defaults for the unspecified classes until a data item is declared to be of that type. Two rules govern the use of attributes in a TYPE section:

• The attributes of the type can neither conflict with nor duplicate any attributes explicitly stated in the data item's declaration.

• The type cannot be used anywhere that its accompanying attributes are illegal.

The following example shows both legal and illegal use of attributes in type definitions:

```
TYPE
    A = [GLOBAL] INTEGER;
    B = [UNALIGNED] INTEGER;
VAR
    A1 : [GLOBAL] A;          { Illegal; duplicates GLOBAL
                                attribute  of   type   A   }
    A2 : [EXTERNAL] A;        { Illegal; conflicts with
                                GLOBAL attribute of type A }
    B1 : ^B;                  { Illegal; pointer base type
                                cannot   be   UNALIGNED    }
    C  : A;                   { Legal }
```

The first three variable declarations are illegal for the reasons shown in the comments. The declaration of C is legal; C is declared as a global INTEGER because of the characteristics of its type. The compiler supplies defaults for all other classes applicable to the variable C.

Attributes associated with data items usually modify type compatibility rules. These modifications are explained in the sections describing individual attributes.

For More Information:

- On extended-string syntax (Section 2.6)
- On program elements and attribute properties (Section 10.3)
- On type compatibility (Section 2.9)

10.2 Attributes

The following sections describe each attribute in alphabetical order.

10.2.1 ALIGNED

The ALIGNED attribute indicates the object is to be aligned on a specific memory boundary.

ALIGNED [[(n)]]

An aligned object is aligned on the memory boundary indicated by n. The constant expression n indicates that the address of the object must end in at least n zeros. ALIGNED(0) specifies byte alignment, ALIGNED(1) specifies word alignment, ALIGNED(2) specifies longword alignment, ALIGNED(3) specifies quadword alignment, ALIGNED(4) specifies octaword alignment, and ALIGNED(9) specifies page alignment.

Usage and Default Information:

- The default alignment of an object depends on its size.
- The constant expression n must denote an integer. If you omit n, the default is 0, indicating byte alignment.
- ALIGNED(9) is the largest alignment allowed.
- An automatic variable cannot have alignment greater than a longword.
- The minimum alignment for an object of a structured type is the greatest alignment specified for any of its components.
- Alignment attributes are illegal on nonstatic types, components of files, and on VARYING OF CHAR strings.
- The alignment of a formal VAR parameter cannot be greater than the alignment of a corresponding actual parameter, either by default or by means of an alignment attribute. In an array variable passed to a conformant formal parameter, alignment and size attributes are illegal on all dimensions of the actual parameter, except the first, that correspond to the dimensions of the formal parameter.

- The base type of a pointer variable passed to the NEW procedure cannot have alignment greater than a quadword.

- If the base type of a pointer variable has a specified alignment, then the base type of a pointer expression assigned to it must have an alignment equal to that of the variable.

- Pointer types are structurally compatible only if their base types have identical alignment.

The following is an example of the ALIGNED attribute:

```
VAR
    Free_Buffers : [ ALIGNED( 1 ), WORD] -2**15..2**15-1;
{In the executable section:}
IF ADD_INTERLOCKED( -1, Free_Buffers ) <= 0 THEN
    {Statement:}
```

The predeclared function ADD_INTERLOCKED requires that the second parameter passed to it have word alignment and an allocation size of one word. In this example, the variable Free_Buffers is declared with alignment and size attributes to meet these restrictions.

For More Information:

- On automatic and size attribute classes (Section 10.3)

- On static and nonstatic types (Section 2.8)

- On default alignments (*VAX Pascal Reference Supplement for VMS Systems*)

10.2.2 ASYNCHRONOUS

The ASYNCHRONOUS attribute indicates that a routine may be called by an asynchronous event (such as a condition handler). Since such an event can alter the values of variables within the routine upredictably, this attribute forces the routine to reference only local variables or variables declared with the VOLATILE attribute.

Usage and Default Information:

- This attribute can be applied to routines and to routine parameters declared in external routines.

- In the absence of the ASYNCHRONOUS attribute, the compiler assumes that the routine can be activated only by actual calls within the program.

- All predeclared routines are asynchronous by default.

- Any routines called from within the block of an asynchronous routine must be local to the asynchronous routine or must themselves be asynchronous, either by default or by an explicit attribute.

- All nonlocal variables accessed from within the block of an asynchronous routine must be declared VOLATILE or READONLY.

- If a formal routine parameter is asynchronous, all actual parameters passed to it must also be asynchronous.

- An asynchronous routine can be passed as an actual parameter to a formal routine parameter that does not have this attribute.

Consider the following:

```
PROCEDURE Do_Something;
   VAR
      i : [VOLATILE] INTEGER;
      j : INTEGER
[ASYNCHRONOUS] FUNCTION Handler {Two array parameters} : BOOLEAN;
      BEGIN
      i := i + 1;
      {Remaining function body...}
{In the executable section of the procedure:}
ESTABLISH( Handler );
```

This example illustrates the declaration of the asynchronous function Handler. The executable section of Handler cannot access variables declared in the enclosing block of the procedure Do_Something unless those variables are declared VOLATILE. So, Handler can access the variable i, which has the VOLATILE attribute, but cannot access the variable j.

For More Information:

- On the VOLATILE attribute (Section 10.2.41)

- On the READONLY attribute (Section 10.2.33)

- On the ESTABLISH procedure (Section 8.25)

10.2.3 AT

The AT attribute specifies that VAX Pascal allocates no storage for the object (storage has already been allocated) and that it resides at the exact, specified address.

AT(n)

The exact address is specified by the constant expression n. Variables representing machine-dependent entities are frequently given the AT attribute.

Usage and Default Information:

- A variable having the AT, COMMON, or PSECT attribute is implicitly static.

- AT cannot be applied to routines or to compilation units.

- AT cannot be applied to variables of nonstatic types.

For More Information:

- On default allocation for variables declared in the outermost block of a program or in nested blocks (Section 10.2.4)

- On default allocation for variables declared in the outermost block of a module (Section 10.2.35)

- On static and nonstatic types (Section 2.8)

10.2.4 AUTOMATIC

The AUTOMATIC attribute specifies that storage for the variable be allocated each time the program enters the routine in which the variable is declared. The storage is deallocated each time the program exits from that routine. An automatic variable exists as long as the declaring routine remains active.

Usage and Default Information:

- By default, variables declared in nested blocks are automatic.

- By default, variables declared at the outermost level of a program are automatic, though for efficiency they can be made static.

- By default, the control part of the nonstatic types and the pointer part of variables of nonstatic types follow the same rules as regular variables: they are static or automatic depending on the location of the declaration and the usage of the data.

- Global and external variables are implicitly static. Thus, they conflict with the AUTOMATIC attribute.

- Program-level variables with the AUTOMATIC attribute are not recorded in environment files.

- AUTOMATIC cannot be applied to routines and compilation units.

- AUTOMATIC cannot be applied to nonstatic types.

For More Information:

- On an example of the STATIC attribute (Section 10.2.35)
- On the GLOBAL attribute (Section 10.2.15)
- On the EXTERNAL attribute (Section 10.2.13)
- On static and nonstatic types (Section 2.8)
- On default allocation for automatic variables (*VAX Pascal Reference Supplement for VMS Systems*)

10.2.5 BIT

The BIT attribute specifies the amount of storage in bits to be received by the object.

BIT[[(n)]]

The optional constant n indicates the number of bit storage units.

Usage and Default Information:

- The default size of an object depends on its type.
- The constant expression n must denote a positive integer. If you omit n, the default value is 1.
- In VAX Pascal, the following size rules apply:
  - Objects of ordinal types cannot have sizes larger than 32 bits.
  - Objects of REAL, SINGLE, and pointer types must have sizes of exactly 32 bits.
  - Objects of type DOUBLE must have sizes of 64 bits.
  - Objects of type QUADRUPLE must have sizes of 128 bits.
- The amount of storage described must be large enough to contain an object of the specified type; otherwise, a compile-time error occurs.
- Assignment to variables with a size attribute are zero-extended (if necessary) and all bits are written.
- When you fetch from variables with a size attribute, you need only reference sufficient bits to access the legal value of the type. The contents of a variable are undefined if it does not contain a zero-extended legal value of the variable's type.

- A size attribute is illegal on a conformant parameter, on a component of a VARYING string, on an object of a structured type having a file component, or on a nonstatic type. In an array variable passed to a conformant formal parameter, size and alignment attributes are illegal on all dimensions of the actual parameter, except the first, that correspond to the dimensions of the formal parameter.

- Two variables of the same type that have different allocation sizes are assignment compatible, but are not structurally compatible.

For More Information:

- On alignment attributes (Section 10.3)
- On type compatibility (Section 2.9)
- On size attributes and return values of size functions (Section 8.66)
- On default sizes of objects (*VAX Pascal Reference Supplement for VMS Systems*)

10.2.6 BYTE

The BYTE attribute specifies the amount of storage in bytes to be received by the object.

BYTE [[(n)]]

The optional constant n indicates the number of byte storage units.

For More Information:

- On VAX Pascal size rules (Section 10.2.5)
- On default sizes of objects (*VAX Pascal Reference Supplement for VMS Systems*)

10.2.7 CHECK

The CHECK attribute specifies error-checking options that are to be enabled during program execution.

CHECK [[({identifier},...)]]

An identifier specifies an option to be enabled. If you omit the list of options, all available positive options are enabled.

Table 10–1 presents the options that allow you to choose which aspects of a program should be checked. The negations of an option disable checking for that option.

Table 10–1: Summary of Checking Options

Option	Action	Negation
ALL	Enables all forms of checking.	NONE
BOUNDS	Verifies that an index expression is within the bounds of an array's index type and that character-string sizes are compatible with the operations being performed and that schema types are compatible.	NOBOUNDS
CASE_SELECTORS	Verifies that the value of a case selector is contained in the corresponding case label list.	NOCASE_SELECTORS
DECLARATIONS	Verifies that schema definitions yield valid types and that uses of GOTO from one block to an enclosing block are correct.	NODECLARATIONS
OVERFLOW	Verifies that the result of an integer computation does not exceed the machine representation.	NOOVERFLOW
POINTERS	Verifies that the value of a pointer variable is not NIL.	NOPOINTERS
SUBRANGE	Verifies that values assigned to variables of subrange types are within the subrange; verifies that a set expression is assignment compatible with a set variable.	NOSUBRANGE

Usage and Default Information:

- This attribute can be applied to routines and compilation units.
- BOUNDS and DECLARATIONS are the only options enabled by default. The defaults for the other options are NOCASE_SELECTORS, NOOVERFLOW, NOPOINTERS, and NOSUBRANGE. If you specify options for CHECK, VAX Pascal enables only the specified options. Consider the following.

```
[CHECK]              { is equivalent to }   [CHECK( ALL )]
[CHECK( option )]    { is equivalent to }   [CHECK( NONE, option )]
{No attribute}       { is equivalent to }   [CHECK( BOUNDS,
                                                   DECLARATIONS )]
```

Consider the following:

```
PROGRAM Check_Features;

[CHECK( POINTERS, CASE_SELECTORS )] PROCEDURE Linked_List
    (VAR Client : Info_Rec);    {Body of the procedure...}

[CHECK( OVERFLOW )] FUNCTION Integer_Compute
    (VAR Int1, Int2, Int3 : INTEGER) : INTEGER;
    {Body of the function...}

PROCEDURE Bounds_Check (VAR A_String : VARYING[30] OF CHAR;
                        VAR Char_Array : ARRAY[1..25] OF CHAR;
                        VAR Half_Alpha : 'A'..'M'); {Body...}
```

For the routines Linked_List and Integer_Compute, VAX Pascal enables
only the specified options. The procedure Bounds_Check has only the
BOUNDS and DECLARATIONS options enabled by default (unless you use
a compilation switch to override the default).

For More Information:

For information on type compatibility, see Section 2.9.

0.2.8 CLASS_A

The CLASS_A attribute causes a formal parameter to be passed by an
array descriptor that describes contiguous arrays of atomic data types or
contiguous arrays of fixed-length strings. This is the default mechanism
for conformant array parameters. This attribute is illegal on parameters of
schema types.

Consider the following example:

```
PROCEDURE Test2( P3 : [CLASS_S] PACKED ARRAY[L..U : INTEGER] OF CHAR;
                 P4 : [CLASS_A] ARRAY[L2..U2 : INTEGER] OF REAL); EXTERN;
```

This example defines a procedure Test2, which has two parameters. The first
parameter, P3, is passed by descriptor of CLASS_S. The second parameter,
P4, is passed by a CLASS_A descriptor.

For More Information:

• On VAX Pascal parameter defaults (Section 6.3)

• On CLASS_A descriptors (*Introduction to VMS System Routines*)

10.2.9 CLASS_NCA

The CLASS_NCA attribute causes a formal parameter to be passed by a noncontiguous array descriptor. This attribute is illegal on parameters of schema types.

For More Information:

* On VAX Pascal parameter defaults (Section 6.3)
* On CLASS_NCA descriptors (*Introduction to VMS System Routines*)

10.2.10 CLASS_S

The CLASS_S attribute causes a formal parameter to be passed by a single descriptor form that is used for scalar data and fixed-length strings. This attribute allows routines written in VAX Pascal to accept actual parameters from languages, such as VAX FORTRAN, that generate CLASS_S descriptors.

Usage and Default Information:

* In order to pass a CLASS_S string descriptor, you must use a packed conformant array of characters. This overrides the string-descriptor default for conformant formal parameters (CLASS_A).
* This attribute is illegal on parameters of schema types.
* When the packed conformant array is passed by CLASS_S descriptor, the lower bound of the conformant schema is always 1 and the upper bound of the conformant schema is the length of the string being passed.

Consider the following example:

```
PROCEDURE Print_String( String_Parm :
        [CLASS_S] PACKED ARRAY[LOW..HIGH : INTEGER] OF CHAR );
BEGIN
   WRITELN( 'The CLASS_S string is', String_Parm );
   WRITELN( 'The lowerbound is', Low );
   WRITELN( 'The upperbound is', High );
END;
```

The previous example defines the procedure Print_String, which has one parameter. The CLASS_S attribute on the VAX Pascal routine specifies that the calling routine passes the String_Parm parameter by a CLASS_S descriptor.

For More Information:

- On VAX Pascal parameter defaults (Section 6.3)

- On mixed-language programming (*VAX Pascal Reference Supplement for VMS Systems*)

- On CLASS_A and CLASS_S descriptors (*Introduction to VMS System Routines*)

0.2.11 COMMON

The COMMON attribute specifies that storage for a variable be allocated in an overlaid program section called a common block.

COMMON [[(identifier)]]

If you include an identifier in the attribute, it indicates the name of the common block. If you omit the identifier, the name of the variable is used as the name of the common block.

This attribute allows you to share variables with other VAX languages, such as FORTRAN.

Usage and Default Information:

- A variable having the AT, COMMON, or PSECT attribute is implicitly static.

- The COMMON attribute can be applied only to variables.

- Only one variable can be allocated in a particular common block. Therefore, the name of the common block cannot be used as the name of another common block or program section.

- If a VAX Pascal program shares a record variable with a FORTRAN program, the fields must be laid out identically in both common blocks.

- Variables declared with the COMMON attribute are longword aligned by default for compatibility with other VAX languages.

For More Information:

- On default allocation for variables declared in the outermost block of a program or in nested blocks (Section 10.2.4)

- On default allocation for variables declared in the outermost block of a module (Section 10.2.35)

- On environment-specific information on common blocks (*VAX Pascal Reference Supplement for VMS Systems*)

10.2.12 ENVIRONMENT

The ENVIRONMENT attribute can be applied to compilation units and causes the unit's program or module level declarations and definitions to be saved.

ENVIRONMENT [[(name-string)]]

If the name string is omitted, the name of the source file is used as the environment file name.

The declarations and definitions made at the outermost level of the compilation unit (provided they do not have the AUTOMATIC or HIDDEN attribute) are saved in a newly created environment file. If the name string is specified, you must include a legal file specification.

Usage and Default Information:

- The default file type for an environment file is .PEN.
- The ENVIRONMENT attribute may not be specified on a program that declares nonstatic types or variables of nonstatic types at the outermost level.
- The ENVIRONMENT attribute may be specified on a module that declares nonstatic types or variables of nonstatic types at the outermost level.
- Programs and modules may access definitions and declarations in a created environment file by using the INHERIT attribute.

For More Information:

- On name-string syntax (Section 10.1)
- On static and nonstatic types (Section 10.2.35)
- On programs and modules (Section 7.3)
- On examples of separate compilation (*VAX Pascal User Manual*)

10.2.13 EXTERNAL

The EXTERNAL attribute indicates a variable or routine that is assumed to be global in another independently compiled unit.

EXTERNAL [[(identifier)]]

If you specify an identifier with EXTERNAL, VAX Pascal supplies that name, rather than the identifier being declared, to the linker.

Usage and Default Information:

- The names available to the linker for corresponding global and external variables and routines must be identical.

- Global and external variables are implicitly static. Thus, they conflict with the AUTOMATIC attribute.

- Compilation units cannot have the EXTERNAL or WEAK_EXTERNAL attribute.

- By default, global and external routines have the characteristics of unbound routines.

- External routines must be followed by the directive EXTERN, EXTERNAL, or FORTRAN when they are declared.

Consider the following:

```
PROGRAM Freshman_Class;

[GLOBAL( Sort_Students )]
PROCEDURE Class_List( VAR Register_List, Sorted_List : Student_Rec );
{Procedure body...}

{In another compilation unit:}
MODULE Senior_Class;

[EXTERNAL( Sort_Students )]
PROCEDURE Roll_Call( VAR Start_List, End_List : Senior_Rec ); EXTERNAL;
```

This example shows the global declaration of a procedure with the name Sort_Students and an external reference to the same procedure in a different compilation unit.

For More Information:

- On default visibility attribute information (Section 10.2.23)

- On the GLOBAL attribute (Section 10.2.15)

- On the AUTOMATIC attribute (Section 10.2.4)

- On the UNBOUND attribute (Section 10.2.38)

- On compiling and linking (*VAX Pascal Reference Supplement for VMS Systems*)

10.2.14 G_FLOATING

The G_FLOATING attribute specifies that the double-precision variables and expressions in the compilation unit are to be represented in G_floating format.

Usage and Default Information:

- NOG_FLOATING is the default double-precision attribute.

- All independently compiled units that are linked together should use the same double-precision format. If you are passing data between routines written in different compilation units that may be using different precisions, you need to check to make sure that the precisions are the same.

- You can use both types of double-precision in the same compilation unit. However, they may not be used together in the same expression, and you cannot assign a value of one double-precision format into a variable of the other format.

Consider the following:

```
[G_FLOATING, ENVIRONMENT( 'REALDATA.PEN' )] MODULE Real_Data;
    {Module declarations...}
[G_FLOATING, ENVIRONMENT( 'STRINGDATA.PEN' )] MODULE String_Data;
    {Module declarations...}
[G_FLOATING, INHERIT( 'REALDATA.PEN', 'STRINGDATA.PEN' )]
PROGRAM Record_Keeping;
    {Program declarations and body...}
```

This example shows the headings of a program and the two modules whose environments it inherits. All three compilation units must specify the G_FLOATING attribute in order for the G_floating format of representation to be used.

For More Information:

- On the NOG_FLOATING attribute (Section 10.2.25)

- On the precison of G_floating objects (*VAX Pascal Reference Supplement for VMS Systems*)

10.2.15 GLOBAL

The GLOBAL attribute provides a strong definition of a variable or routine so that other independently compiled units can refer to it.

GLOBAL [[(identifier)]]

If you specify an identifier with GLOBAL, VAX Pascal supplies that name, rather than the identifier being declared, to the linker.

Usage and Default Information:

*   You can apply the GLOBAL attribute to variables, routines, and compilation units.

*   Global and external variables are implicitly static. Thus, they conflict with the AUTOMATIC attribute.

*   By default, global and external routines have the characteristics of unbound routines.

*   You cannot apply the GLOBAL attribute to variables of nonstatic types.

For More Information:

*   On default visibility attribute information (Section 10.2.23)

*   On an example of GLOBAL and on the EXTERNAL attribute (Section 10.2.13)

*   On compiling and linking (*VAX Pascal Reference Supplement for VMS Systems*)

10.2.16 HIDDEN

The HIDDEN attribute prevents information concerning a constant definition or a type, variable, procedure, or function declaration from being included in a generated environment file. The HIDDEN attribute can be used only on objects at the outermost level of the compilation unit.

It is possible to prevent all declarations within a declaration section from being included in the environment file by preceding the reserved word CONST, TYPE, or VAR with the HIDDEN attribute.

For More Information:

For information on environment files, see Section 10.2.12.

10.2.17 IDENT

The IDENT attribute can be used to qualify the name of a compilation unit. In the absence of an IDENT attribute, the string ′01′ is supplied to the linker.

IDENT(name-string)

The name-string can contain additional information whose use is implementation specific. The VAX Pascal compiler uses this string to supply identification information to the linker.

Consider the following:

```
[IDENT( '100.5' ),ENVIRONMENT( 'SAMPLE.PEN' )] MODULE SAMPLE;
```

In this example, the IDENT string '100.5' is supplied to the linker.

For More Information:

- On name-string syntax (Section 10.1)

- On compiling and linking (*VAX Pascal Reference Supplement for VMS Systems*)

10.2.18 IMMEDIATE

The IMMEDIATE attribute causes a formal parameter value in a routine declaration to be passed by immediate value.

Usage and Default Information:

- The IMMEDIATE attribute can appear only in external routine declarations.

- The IMMEDIATE attribute is not allowed on formal parameters of schema types.

For More Information:

- On default parameter passing (Section 6.3)

- On an example of IMMEDIATE and on the REFERENCE attribute (Section 10.2.34)

10.2.19 INHERIT

The INHERIT attribute indicates the environment file or files to be inherited by a compilation unit. The environment files specified by the INHERIT attribute must already have been created in compilation units (by either the ENVIRONMENT attribute or a compilation switch).

INHERIT({name-string},...)

The compilation unit inherits one or more environment files named by the file specifications in the name strings. The default file type for an inherited environment file is .PEN.

For More Information:

- On programs and modules (Section 7.3)
- On file specifications and on compilation switches (*VAX Pascal Reference Supplement for VMS Systems*)
- On separate compilation (*VAX Pascal User Manual*)

10.2.20 INITIALIZE

The INITIALIZE attribute can be applied to procedures to indicate that the procedure is to be called before the main program is entered. A compilation unit might include any number of INITIALIZE procedures, all of which are called in an unspecified order before the main program is entered.

Usage and Default Information:

- In the absence of the INITIALIZE attribute, the compiler assumes that a routine can be activated only by actual calls within the program.
- Within modules, you should use the TO BEGIN DO section instead of the INITIALIZE attribute. All TO BEGIN DO clauses are executed before INITIALIZE routines.
- By default, INITIALIZE procedures have the characteristics of unbound routines.
- An INITIALIZE procedure cannot have a formal parameter list.
- An INITIALIZE procedure cannot be external.

Consider the following:

```
PROGRAM Routine_Activate;

[INITIALIZE] PROCEDURE Check_Open; {Procedure body...}

{In the executable section:}
BEGIN    {VAX Pascal activates Check_Open}
   {Body of program...}
```

In this example, the body of the INITIALIZE procedure Check_Open is executed before the main program is activated.

For More Information:

- On procedures (Section 6.1)
- On the UNBOUND attribute (Section 10.2.38)

10.2.21 KEY

The KEY attribute can be applied to record fields to indicate that the field is to be used as a key field when the record is part of an indexed file.

$$\text{KEY } [[(\begin{Bmatrix} \text{n } [[, \{\text{options}\},...]] \\ \{\text{options}\},... \end{Bmatrix})]]$$

The parameter n represents the key number. A key number of 0 indicates that the field is the primary key of the record. All other key numbers indicate alternate keys. The key number must be a constant expression that denotes an integer value in the range from 0 through 254.

In addition, you can specify certain characteristics of the record key by listing the desired options on the KEY attribute.

Table 10–2 lists the possible KEY attribute options.

Table 10–2: KEY Attribute Options

Option	Action	Negation
ASCENDING	Specifies an ascending collating sequence	DESCENDING
CHANGES	Specifies that changes can be performed on the key	NOCHANGES
DUPLICATES	Specifies that duplicates of the key are allowed	NODUPLICATES

Usage and Default Information:

- If you omit the key number, the default value is 0.
- By default, the primary key is ASCENDING, NOCHANGES, and NODUPLICATES. It is possible to override these defaults, with the exception of the NOCHANGES option. It is illegal to specify CHANGES on the primary key.
- The default for an alternate key is ASCENDING, CHANGES, and DUPLICATES.
- When you create a new indexed file with more than one key field, you cannot omit any key numbers in the range from 0 through the highest key number specified.
- The KEY attribute is ignored except when the record is a component of a file.

- A key field can be of any ordinal type or of type PACKED ARRAY OF CHAR. If the key field is of type PACKED ARRAY OF CHAR, its length cannot exceed 255 characters.

- The KEY attribute does not affect type compatibility rules.

- A key field cannot be unaligned.

- A key field of an ordinal type must be allocated in exactly one byte, one word, or one longword.

- An integer key field that is allocated one byte cannot have negative values.

Consider the following:

```
TYPE
   Register = RECORD
        Student_No        : [KEY( 0, DESCENDING )] INTEGER;
        Student_Name      : RECORD
           Last_Name      : PACKED ARRAY[1..20] OF CHAR;
           First_Name     : PACKED ARRAY[1..15] OF CHAR;
           Initial        : CHAR;
           END;
        Course_Load       : INTEGER;
        Grade_Average     : REAL;
        Class             : [KEY( 1 )] PACKED ARRAY[1..9] OF CHAR;
        END;
```

This example defines the identifier Register to denote a record type. The first field, Student_No is the primary key of the record. Notice that it has been defined as a DESCENDING, NOCHANGES and NODUPLICATES key. Register contains another field, Class, which is established as the alternate ASCENDING, CHANGES and DUPLICATES key.

For More Information:

- On indexed files (Section 9.1.3)

- On the UNALIGNED attribute (Section 10.2.37)

10.2.22 LIST

The LIST attribute can be applied to a formal parameter of a routine and indicates that the routine can be called with multiple actual parameters that correspond to the last formal parameter named in the routine heading.

You can also use the ARGUMENT and ARGUMENT_LIST_LENGTH predeclared routines when writing procedures and functions that use the LIST attribute.

Usage and Default Information:

- In the absence of a LIST attribute, an error results if the number of actual parameters exceeds the number of formal parameters.

- The LIST attribute can be applied only to the last formal parameter in a parameter list.

- You can supply zero, one, or more than one actual parameter to correspond to a LIST formal parameter, but you must use positional syntax when supplying them. The number of actual parameters you can supply is limited to 255.

- You can use the LIST attribute on the parameter list of a routine parameter, but you must use positional syntax when specifying them. Using the LIST attribute on routine parameters is allowed only on external routines.

- You can use the LIST attribute on conformant parameters to indicate that an external routine can take an arbitrary number of arrays or VARYING OF CHAR parameters, respectively. Using the LIST attribute on conformant parameters is allowed only on external routines.

- All actual parameters that correspond to a LIST formal parameter must be compatible or congruent with the type of the formal parameter.

- For formal and actual parameter lists of routine parameters to be congruent, the actual routine parameter and the corresponding formal routine parameter must either both have the LIST attribute or both lack the LIST attribute. Consider the following:

```
PROCEDURE Foo( PROCEDURE q( x : [LIST] CHAR ) );
```

This defines the routine Foo with the formal routine parameter q that defines the formal list parameter x. Consider the following:

```
PROCEDURE Bar( x : [LIST] CHAR );
```

This defines Bar to have a formal list parameter x. Consider this call to Foo:

```
Foo( Bar );
```

This calls Foo passing the actual routine parameter Bar. Notice the formal parameters of q and Bar contain the LIST attribute; therefore, this is a legal call.

Consider the following:

```
PROGRAM Arg_Mech;

[EXTERNAL( MTH$JMAX0 )] FUNCTION JMax0
    (Int_List : [LIST] INTEGER) : INTEGER; EXTERNAL;
VAR
    i, j, k, l : INTEGER;
    Int_Array  : ARRAY[1..10] OF INTEGER;

{In the executable section:}
i := JMax0( j, k, l, Int_Array[ j+1 ], Int_Array[ k+2 ],
            Int_Array[ l+3 ] );
```

The program Arg_Mech illustrates the effect of the LIST attribute on the external function MTH$JMAX0. Within the program, this routine is known as the function JMax0. JMax0 is declared with one formal LIST parameter; therefore, the function designator in this example contains excess actual parameter entries. Any number of integer expressions can be passed as actual parameters when JMax0 is called.

For More Information:

- On the ARGUMENT function (Section 8.5)

- On the ARGUMENT_LIST_LENGTH function (Section 8.6)

- On type compatibility (Section 2.9)

10.2.23 LOCAL

The LOCAL attribute indicates that an object is unavailable to other independently compiled units.

Usage and Default Information:

- By default, all variables and routines are local.

- Variables with any visibility attribute other than LOCAL are implicitly static.

- Routines with any visibility attribute other than LOCAL cannot refer to automatic variables declared in enclosing blocks and can call only those routines that are local, predeclared, or unbound. (By default, routines declared at program or module level have the characteristics of unbound routines.)

For More Information:

- On the AUTOMATIC attribute (Section 10.2.4)
- On static and nonstatic types (Section 10.2.35)
- On the UNBOUND attribute (Section 10.2.38)

10.2.24 LONG

The LONG attribute specifies the amount of storage in longwords to be received by the object.

LONG [[(n)]]

The optional constant n indicates the number of longword storage units.

Consider the following:

```
PROGRAM Size;
TYPE
    Status = [LONG] BOOLEAN;
VAR
    Return_Status : Status;

FUNCTION Example( Param1, Param2 : INTEGER ) : Status; EXTERNAL;
    {Function body...}
```

The program Size defines a BOOLEAN type Status and declares a variable Return_Status of this type. So, the result type of the function is declared to have a size of one longword. The machine code that references the result type may not copy the entire longword, as the lower order byte contains sufficient room to represent a Boolean value.

For More Information:

For information on VAX Pascal size rules, see Section 10.2.5.

10.2.25 NOG_FLOATING

The NOG_FLOATING attribute specifies that the double-precision variables and the expressions in the compilation unit are to be represented in D_floating format.

Usage and Default Information:

NOG_FLOATING is the default double-precision attribute.

For More Information:

* On the G_FLOATING attribute (Section 10.2.14)
* On the precison of F_floating objects (*VAX Pascal Reference Supplement for VMS Systems*)

10.2.26 NOOPTIMIZE

The NOOPTIMIZE attribute prohibits the compiler from optimizing code for the compilation unit or routine.

Consider the following:

```
PROGRAM Numbers;

[NOOPTIMIZE] PROCEDURE Process_Negative;    {Procedure body...}
```

This example shows the use of the NOOPTIMIZE attribute to disable optimization of the code for the procedure Process_Negative. VAX Pascal optimizes code for the rest of the compilation unit.

The NOOPTIMIZE attribute guarantees left-to-right evaluation order with full evaluation of both operands of the AND and OR Boolean operators to aid in diagnosing all potential programming errors. If you wish to have short circuit evaluation even with the NOOPTIMIZE attribute, then use the AND_THEN and OR_ELSE Boolean operators.

For More Information:

* On the OPTIMIZE attribute (Section 10.2.28)
* On the AND_THEN and OR_ELSE logical operators (Section 4.2.3)

10.2.27 OCTA

The OCTA attribute specifies the amount of storage in octawords to be received by the object.

OCTA [[(n)]]

The optional constant n indicates the number of octaword storage units.

For More Information:

* On VAX Pascal size rules (Section 10.2.5)
* On default sizes of objects (*VAX Pascal Reference Supplement for VMS Systems*)

10.2.28 OPTIMIZE

The OPTIMIZE attribute specifies optimization options that are to be enabled during compilation of a compilation unit or routine.

OPTIMIZE [[({identifier},...)]]

The options listed with the OPTIMIZE attribute are enabled. The negations of options disable optimization of those options. Table 10–3 presents the program options for VAX Pascal to optimize.

Table 10–3: OPTIMIZE Attribute Options

Option	Action	Negation
ALL	Enables all optimization components	NONE
INLINE	Enables inline expansion of user-defined routines	NOINLINE

Usage and Default Information:

- If you omit the list of options, all positive options are enabled.
- If an explicit OPTIMIZE(INLINE) attribute exists on a routine declaration, the compiler checks for anything that prohibits the routine from being inline expanded, such as being an external routine. However, the compiler does not check the call environment, such as the size of the calling and called routine. Instead, if it is legal to expand the routine, it always expands the code regardless of the call environment. This gives you more control over the decision to inline a routine.
- VAX Pascal does not inline routines that have formal parameters of nonstatic types, or that declare or access objects of nonstatic types.

For More Information:

- On the NOOPTIMIZE attribute (Section 10.2.26)
- On the rules for routine inlining (*VAX Pascal Reference Supplement for VMS Systems*)

10.2.29 OVERLAID

The OVERLAID attribute indicates how storage should be allocated for variables declared within a compilation unit. If you specify OVERLAID on a compilation unit, the variables declared at program or module level (unless they have the STATIC or PSECT attribute) overlay the storage of static variables in all other overlaid compilation units.

Usage and Default Information:

*   By default, variables are not stored in overlaid compilation units. This attribute is intended for use only with programs that use the decommitted separate compilation facility provided by VAX Pascal Version 1.0.

*   This attribute is not allowed on a compilation unit that contains or uses nonstatic types at the outermost level.

For More Information:

*   On static and nonstatic types (Section 2.8)
*   On the PSECT attribute (Section 10.2.31)

10.2.30 POS

The POS attribute forces the field to a specific bit position within the record.

POS(n)

The constant expression n specifies the bit location, relative to the beginning of the record, at which the field begins.

Usage and Default Information:

*   The POS attribute can be applied to a field of a packed or an unpacked record.

*   The constant expression n cannot denote a negative integer.

*   The beginning position of a field must be greater than the ending position of the field preceding it.

*   The POS attribute cannot be used on a field that follows (not necessarily immediately) a field whose type has run-time size and is nonstatic.

*   Inside a record variant, the beginning position of a field must be greater than the ending position of the preceding field within the same variant. The variants themselves may overlap.

- A record variable containing a field of a file type cannot include a POS attribute for any field.

- A field whose allocation size is greater than 32 bits must be positioned on a byte boundary.

- The specified bit position must not conflict with the alignment explicitly required by an alignment attribute.

- Two record types in which corresponding fields are not identically positioned are neither assignment compatible nor structurally compatible.

Consider the following:

```
TYPE
   Control = RECORD
      Flag_1 : [ BIT, POS( 0 ) ] BOOLEAN;
      Flag_2 : [ BIT, POS( 1 ) ] BOOLEAN;
      Count  : [ BYTE, ALIGNED ] 0..100;
      Error  : [ BIT, POS( 31 ) ] BOOLEAN;
      END;
```

This example uses the POS attribute to position the fields of an unpacked record such that Flag_1 occupies bit 0, Flag_2 occupies bit 1, and Error occupies bit 31. Because the Count field has size and alignment attributes, it is allocated one byte of storage and is aligned on the byte boundary following Flag_2; that is, storage for Count occupies bits 8 through 15. Bits 2 through 7 and 16 through 30 are left empty; you cannot refer to them.

For More Information:

- On static and nonstatic types (Section 2.8)

- On type compatibility (Section 2.9)

- On default positioning of record fields (*VAX Pascal Reference Supplement for VMS Systems*)

10.2.31 PSECT

The PSECT attribute is useful for placing static variables and executable blocks in program sections that are shared among executable images.

PSECT(identifier)

The identifier designates the program section in which storage for a variable, routine, or compilation unit is to be allocated. This name can designate a program section that is created by the compiler or by the user. Storage for the object remains allocated as long as the executable image in which the object was declared remains active.

Usage and Default Information:

*   A variable having the AT, COMMON, or PSECT attribute is implicitly static.

*   PSECT is the only allocation attribute that can be applied to routines and compilation units.

For More Information:

*   On default allocation for variables declared in the outermost block of a program or in nested blocks (Section 10.2.4)

*   On default allocation for variables declared in the outermost block of a module (Section 10.2.35)

*   On program sections (*VAX Pascal Reference Supplement for VMS Systems*)

10.2.32 QUAD

The QUAD attribute specifies the amount of storage in quadwords to be received by the object.

QUAD [[(n)]]

The optional constant n indicates the number of quadword storage units.

For More Information:

*   On VAX Pascal size rules (Section 10.2.5)

*   On default sizes of objects (*VAX Pascal Reference Supplement for VMS Systems*)

10.2.33 READONLY

The READONLY attribute specifies that an object can be read by a program, but cannot have values assigned to it.

Usage and Default Information:

*   This attribute can be applied to variables, formal parameters, the base types of pointer variables, and components of structured variables.

*   By default, an object can be both read and written.

*   No value of any type is assignment compatible with a read-only object.

- The presence of a read-only component in an object of a structured type prohibits the object from having values assigned to it.

- A read-only actual VAR parameter can be passed only to a read-only formal VAR parameter.

- A pointer expression whose base type is read-only is assignment compatible only with a pointer variable whose base type is also read-only.

Consider the following example:

```
TYPE
   t = RECORD
       i : INTEGER;
       END;
   P_Read_Only = ^ [READONLY] t;
VAR
   Pro : P_Read_Only;
   Prw : ^ T;

PROCEDURE q( p : P_Read_Only);
   VAR
       x : INTEGER;
   BEGIN
   x := p^.i;
   {More statements...}
   END;
{In the executable section:}
NEW( Pro ):
NEW( Prw );
Q( Pro );
Q( Prw );
Prw^.I := 0;
```

This example shows the declaration of two pointer variables, Pro and Prw, and the calls to NEW that create the dynamic variables Pro^ and Prw^. The type of the formal parameter p requires that a corresponding actual parameter have read access; therefore, both Pro and Prw can legally be passed to Q as actual parameters. Because P is a READONLY parameter, the value of the dynamic variable P^ (which corresponds to either Pro^ or Prw^) can be assigned to a variable, as shown in the assignment statement in the body of Q. However, only Prw^ can have values assigned to it, as shown in the last statement.

For More Information:

- On the NEW procedure (Section 8.50)

- On parameters (Section 6.3)

- On type compatibility (Section 2.9)

10.2.34 REFERENCE

The REFERENCE attribute causes the formal parameter value in a routine to be passed by reference using foreign semantics.

Usage and Default Information:

- The REFERENCE attribute can appear only in external routine declarations.
- The REFERENCE attribute is not allowed on formal parameters of schema types.

Consider the following:

```
PROCEDURE Test1( P1 : [REFERENCE] INTEGER;
                 P2 : [IMMEDIATE] INTEGER ); EXTERNAL;
```

This example defines a procedure, Test1, which has two parameters. The first parameter, P1, is passed by reference. The second parameter, P2, is passed by immediate value.

For More Information:

- On default parameter passing (Section 6.3)
- On the IMMEDIATE attribute (Section 10.2.18)

10.2.35 STATIC

The STATIC attribute causes VAX Pascal to create a static object, which is allocated only once and which exists as long as the executable image in which it is allocated remains active.

Usage and Default Information:

- You can override the default (automatic) for variables declared in nested blocks or in the outermost level of compilation units by specifying the STATIC attribute on the variable.
- By default, variables declared at the outermost level of a module are static.
- Global and external variables are implicitly static. Thus, they conflict with the AUTOMATIC attribute.

- A variable having the AT, COMMON, or PSECT attribute is implicitly static.

- Allocation attributes may not be applied to nonstatic types.

Consider the following example:

```
PROGRAM Print_Random( OUTPUT );

VAR
    i : [AUTOMATIC] INTEGER;

FUNCTION Random : INTEGER;
    VAR
        x : [STATIC] INTEGER VALUE 15;
    BEGIN
    x := (( 9 * x ) + 7 ) MOD 11;
    Random := x;
    END;
{In the executable section:}
FOR i := 1 TO 20 DO
    WRITELN( Random );
END.
```

The program Print_Random includes a function that generates a random integer. Because the variable x is declared STATIC, its value is preserved from one activation of the function to the next. By default, the storage for x would have been deallocated when control returned to the main program. Because x is static, it retains the value it had when Random ended and assumes this value the next time Random is called. In the program Print_Random, the program-level variable i is declared AUTOMATIC.

For More Information:

- On the AUTOMATIC attribute (Section 10.2.4)

- On allocation attributes (Section 10.3)

- On default storage of objects (*VAX Pascal Reference Supplement for VMS Systems*)

10.2.36 TRUNCATE

The TRUNCATE attribute indicates that an actual parameter list for a routine can be truncated at the point that the attribute was specified. (TRUNCATE can be used with the PRESENT function.)

Usage and Default Information:

- This attribute can be specified on a formal parameter in a routine declaration.

- If a parameter with the TRUNCATE attribute is specified either in the actual parameter list or is specified by default, the list is not truncated at that point, and any parameters that follow must be present in the actual parameter list (unless they also have the TRUNCATE attribute, or are present by default).

- If a parameter is physically positioned after the TRUNCATE parameter in the formal parameter list, and is present (or defaulted), then the list is not truncated at the TRUNCATE parameter, and any parameters after the TRUNCATE parameter must be present (or present by default) up to the next parameter that specifies the TRUNCATE attribute.

- You can specify actual parameters either positionally or nonpositionally; it is the order in the formal parameter list that is used to determine where the list has been truncated and which parameters are required.

Consider the following example:

```
PROGRAM Trunc( OUTPUT );
VAR
    w    : CHAR VALUE 'w';
    x    : CHAR VALUE 'x';
    y    : CHAR VALUE 'y';
    z    : CHAR VALUE 'z';
PROCEDURE p( a : [TRUNCATE] CHAR := 'a';
             b :            CHAR := 'b';
             c : [TRUNCATE] CHAR := 'c';
             d :            CHAR := 'd' );
    BEGIN
    IF PRESENT( a ) THEN WRITE( a );
    IF PRESENT( b ) THEN WRITE( b );
    IF PRESENT( c ) THEN WRITE( c );
    IF PRESENT( d ) THEN WRITE( d );
    WRITELN;
    END;
{In the executable section:}
{ CALL            LIST                               RESULT  }
  p;              { NO PARAMETERS--TRUNCATE AT a     ""      }
  p();            { DEFAULT a AND b--TRUNCATE AT c "ab"      }
  p(,);           { DEFAULT a AND b--TRUNCATE AT c "ab"      }
  p(,,);          { DEFAULT a, b, c AND d            "abcd"  }
  p(,,,);         { DEFAULT a, b, c AND d            "abcd"  }
  p( w );         { DEFAULT b--TRUNCATE AT c         "wb"    }
  p( w, x );      { TRUNCATE AT c                    "wx"    }
  p( w, x, y );   { DEFAULT d                        "wxyd"  }
  p( w, x, y, z );    { NO DEFAULTS                  "wxyz"  }
```
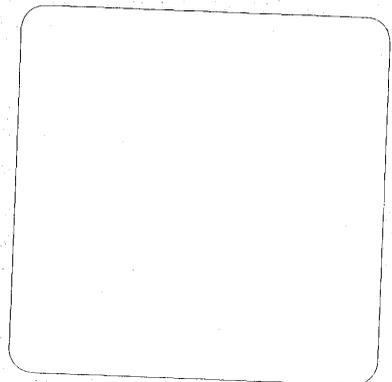
```
p( a := w );    { DEFAULT b--TRUNCATE AT c        "wb"    }
p( b := x );    { DEFAULT a--TRUNCATE AT c        "ax"    }
p( c := y );    { DEFAULT a, b AND d              "abyd" }
p( d := z );    { DEFAULT a, b AND c              "abcz" }
```

In this example, each call to procedure p in the main body of the program
has a comment that shows the expected parameter list behavior and the
expected output. The parameter list is truncated at either parameter a or
parameter c.

If parameters b and d did not have defaults, the call p(w) or p(w, x, y)
would be illegal because the list cannot be truncated at the second or fourth
positions.

For More Information:

- On the PRESENT function (Section 8.58)

- On parameters (Section 6.3)

10.2.37 UNALIGNED

The UNALIGNED attribute specifies that an object can be aligned on any
bit boundary.

Usage and Default Information:

- Alignment attributes are illegal on nonstatic types, components of files,
  and on VARYING OF CHAR strings.

- In VAX Pascal, an unaligned variable cannot have an allocation size
  greater than 32 bits.

- A formal parameter cannot be unaligned. Thus, an unaligned variable
  cannot be passed to a formal variable parameter.

- The base type of a pointer variable passed to the NEW procedure cannot
  have alignment greater than a quadword, nor can it be unaligned.

For More Information:

- On allocation size attributes (Section 10.3)

- On VAX Pascal alignment rules (Section 10.2.1)

10.2.38 UNBOUND

The UNBOUND attribute specifies that a routine does not access automatic variables outside the scope in which it is declared. That is, the bound procedure value of an unbound routine does not include the static scope pointer.

Usage and Default Information:

- This attribute can be applied to routines and formal routine parameters.

- In the absence of an UNBOUND attribute, the compiler assumes that the bound procedure value of a routine includes the static scope pointer.

- By default, all predeclared routines and all routines declared at program or module level have the characteristics of unbound routines. All routines declared in nested blocks are considered bound unless they have an UNBOUND, GLOBAL, WEAK_GLOBAL, or INITIALIZE attribute.

- All routines called from within the block of an unbound routine must be local to the unbound routine, or be unbound, whether by default or by an explicit attribute.

- Nonlocal variables accessed from within the block of an unbound routine cannot have automatic allocation.

- If a formal routine parameter is unbound, all actual routine parameters passed to it must also be unbound.

- You can pass an unbound routine as an actual parameter to a formal routine parameter that is not unbound.

Consider the following:

```
[EXTERNAL] FUNCTION f( [IMMEDIATE, UNBOUND] PROCEDURE Count )
   : BOOLEAN; EXTERNAL;

PROCEDURE a;
   VAR
      i : [STATIC] INTEGER;
      b : BOOLEAN;

   [UNBOUND] PROCEDURE p;
      BEGIN
      i := i + 1;
      {Additional statements...}
      END;
   b := f( p );
   END;
```

This example illustrates the declaration of the unbound procedure p and the unbound formal procedure parameter Count. Note that the executable section of p cannot access variables declared in the enclosing block of procedure a unless those variables are statically allocated. Thus, procedure p can access the variable i, which is declared with the STATIC attribute, but cannot access the variable b, which is automatically allocated. Because the formal parameter Count is unbound, only other unbound routines (such as p) can be passed to function f as actual parameters. Count must be declared UNBOUND because it is passed by immediate value.

For More Information:

- On the AUTOMATIC attribute (Section 10.2.4)
- On parameters (Section 6.3)

10.2.39 UNSAFE

The UNSAFE attribute indicates that an object can accept values of any type without type checking. The exact properties of an unsafe object depend on the object's machine representation.

Usage and Default Information:

- This attribute can be applied to variables, formal parameters, formal discriminants, the base types of pointer variables, components of structured variables, function results, and the types of other data items listed in Table 10–6.

- A conformant VARYING parameter or a formal schema parameter cannot be declared UNSAFE.

- UNSAFE is the only attribute allowed on schema formal discriminants.

- An expression of any type is assignment compatible with an unsafe object. However, neither the expression nor the object can contain a file component. If the machine representations of the expression and the unsafe object differ, the compiler forces them to have the same number of bits by modifying the value of the expression as follows:

  - Assignment to a variable with the UNSAFE attribute causes the value of the right-hand side to be truncated or zero-extended to the bit size of the left-hand variable. Note this may not always be its natural bit size; for example, if the variable you are assigning a value to was declared with an explicit size attribute. If that value is the legal value of the left-hand type, then the assignment occurs; otherwise, the variable is undefined.

- The UNSAFE attribute has no effect on variable fetches.

Consider the following examples:

```
v : [LONG, UNSAFE] ( aa, bb, cc );
```

As an enumeration of less than 256 elements, its natural size is one byte. But because of the LONG attribute, it is allocated a longword in memory. However, all fetches from the variable are BYTE fetches, because the explicit size attribute has no effect on any fetches. Assignments correctly assign the lowest byte of V, but the contents of the extra bits are zero-extended after the assignment.

- A pointer expression is assignment compatible with a pointer variable whose base type is unsafe only if the base types have the same allocation size and if they have compatible alignment, READONLY, VOLATILE, and WRITEONLY attributes.

- An actual parameter variable can be passed to an unsafe formal VAR parameter if the types have the same allocation size and if they have compatible alignment, READONLY, VOLATILE, and WRITEONLY attributes.

- When a formal parameter is an unsafe conformant array, the VAX Pascal compiler must be able to establish bounds for the corresponding actual parameter that exactly describe the amount of storage the parameter occupies. If the conformant array is one-dimensional, the actual parameter need not be an array. The compiler constructs the bounds of the formal array so that the actual parameter and the formal array have the same size.

  For this construction to be possible, the size of the actual parameter must be an exact multiple of the size of the formal array component. The compiler chooses the low bound of the formal parameter's index to be the smallest possible value of the index type. If the formal conformant parameter is a multidimensional array with n dimensions, the actual parameter must be an array having no fewer than n–1 dimensions. The first n–1 dimensions of the two arrays will have identical array bounds. The compiler chooses bounds for the last dimension of the conformant array so that the conformant as a whole describes the exact size of the actual parameter.

VAX Pascal allows you to pass an actual parameter of a schema type to the an unsafe conformant array; however, since VAX Pascal cannot determine the size of the actual parameter until run-time, you must be sure that the actual parameter is an exact multiple of the size of the formal array component.

Consider the following example:

```
PROGRAM Output_Buffer( Data_File );
TYPE
   Natural = 0..MAXINT;
VAR
   Data_File : FILE OF ARRAY[0..511] OF CHAR;
   Int_Array : ARRAY[0..1023] OF INTEGER;
   A_String  : VARYING[2048] OF CHAR;
   Chr_Array : ARRAY[0..4095] OF CHAR;
   Status    : BOOLEAN;

FUNCTION Put_Buf( VAR Buffer :
                   [UNSAFE] ARRAY[ a..b : Natural ] OF CHAR )
         : BOOLEAN;
   VAR
      Cur : [STATIC] INTEGER VALUE 0;
      i   : INTEGER;
   BEGIN
   FOR i := a TO b DO
      BEGIN
      Data_File^[Cur] := Buffer[i];
      Cur := Cur + 1;
      IF Cur > 511 THEN
         BEGIN
         PUT( Data_File);
         Cur := 0;
         END;
      END;
   Put_Buf := (Cur = 0);
   END;
{In the executable section:}
Status := Put_Buf( Int_Array );
Status := Put_Buf( A_String );
Status := Put_Buf( Chr_Array );
```

The function Put_Buf assigns successive components of the conformant array parameter to the file buffer variable of Data_File. If Data_File^ is filled, the function returns TRUE; otherwise, it returns FALSE.

The program issues three calls to Put_Buf. In the first and second calls, the actual parameters are not of the same type as the formal parameter Buffer. However, because Buffer has the UNSAFE attribute, it accepts an actual parameter of any type and treats it as though it were an array of characters. The third call to Put_Buf passes an actual parameter of the same type as the formal parameter.

For More Information:

- On type compatibility (Section 2.9)
- On machine representation of data (*VAX Pascal Reference Supplement for VMS Systems*)

10.2.40 VALUE

The VALUE attribute causes the variable to be a reference to an external constant or to be the defining point of a global constant.

Usage and Default Information:

- The VALUE attribute can be used only on a variable that has the EXTERNAL or GLOBAL attribute.
- A value variable with global visibility must be initialized in the VAR, TYPE, or VALUE declaration sections.
- The VALUE attribute cannot be applied to variables larger than 32 bits.
- The VALUE attribute is legal only on ordinal or real types.
- The VALUE attribute causes the READONLY attribute to be placed on the variable.

Consider the following:

```
PROGRAM Value_Test( OUTPUT );
VAR
    CLI$_PRESENT : [VALUE, EXTERNAL] INTEGER;
    My_Global : [VALUE, GLOBAL] INTEGER VALUE 1985;
{In the executable section:}
WRITELN( 'The value is', CLI$_PRESENT );
```

The linker resolves the reference to CLI$_PRESENT, and the example writes the decimal value to OUTPUT. This example also defines a global symbol with the name My_Global and with a value of 1985.

For More Information:

- On the EXTERNAL attribute (Section 10.2.13)
- On the GLOBAL attribute (Section 10.2.15)
- On the READONLY attribute (Section 10.2.33)

10.2.41 VOLATILE

The VOLATILE attribute indicates to the compiler that the value of an object is subject to change at unusual points in program execution. Normally, during execution, an object's value generally changes only under the following circumstances:

- When another value is assigned to it
- When it is passed as a writeable VAR parameter
- When it is read into by a READ, READLN, or READV procedure
- When it is used as the control variable of a FOR loop

In addition, the compiler expects to evaluate the object only when it appears in an expression.

The value of a volatile object may change as the result of an action not directly specified in the program. Thus, the compiler assumes that the value of a volatile object can be changed or evaluated at any time during program execution. Consequently, a volatile object does not participate in any optimization based on assumptions about its value.

The behavior of many device registers, and modifications by asynchronous processes and exception handlers, are two examples that demonstrate volatile behavior.

Usage and Default Information:

- This attribute can be applied to variables, formal parameters, the base types of pointer variables, components of structured variables, and function results.
- By default, objects are not volatile.
- An object of a structured type that has a volatile component is volatile as a whole. However, the presence of a volatile component does not make other components of the same variable volatile.
- The presence of the VOLATILE attribute guarantees that operations are performed on scalar objects in a single machine instruction. Because operations on structured objects may require more than one instruction, the use of the VOLATILE attribute on an object of a structured type may not produce the expected results.
- A volatile variable is structurally compatible only with a formal variable parameter that is volatile.

- A pointer expression whose base type is volatile is assignment compatible only with a pointer variable whose base type is volatile.

- Two pointer types are structurally compatible only if their base types have identical volatility.

Consider the following:

```
VAR
    x : CHAR;
    a : [VOLATILE] RECORD
        CASE BOOLEAN OF
            FALSE : ( i : INTEGER );
            TRUE  : ( c : CHAR );
        END;
{In the executable section:}
a.c := 'A';        {TRUE becomes the current variant}
a.i := 66;         {Assignment makes FALSE the current variant}
x   := a.c;        {TRUE is again the current variant;
                       X is assigned the value 'B', which
                       has an ordinal value of 66}
```

As the comments in this example show, a reference to one field identifier causes the corresponding variant to become the current variant. In addition, each reference immediately causes the other variant to become undefined. So, when the assignment a.i := 66 is made, the reference to a.i causes FALSE to become the current variant and a.c to become undefined. As a result of the statement x := a.c, the value last assigned to the variant is assigned to x. Ordinarily the compiler could assume that a.c had retained the value 'A', because no further assignments had been made directly to a.c. However, the value of a.c changed unexpectedly through the assignment to a.i. Therefore, unless the record a is declared VOLATILE, the result of the assignment x := a.c would be undefined because the compiler's legitimate assumptions had been incorrect.

Consider the following:

```
PROGRAM Volatility( OUTPUT );
VAR
    Pint : ^[VOLATILE] INTEGER;
    i    : INTEGER;
    j    : [VOLATILE] INTEGER;
    a    : ARRAY[0..10] OF INTEGER;
{In the executable section:}
NEW( Pint );
i     := 0;
j     := 0;
Pint^ := 0;

{Compiler may assume i = 0, makes no assumptions about j}
WRITELN( i, j, Pint^, a[i] );    {Values are 0, 0, 0, a[0] }
Pint  := ADDRESS( j );           {Pint^ now = j}
Pint^ := 1;                      {Therefore j now = 1}
```

```
{Compiler may assume i = 0, makes no assumptions about j}
WRITELN( i, j, Pint^, a[i] );    {Values are 0, 1, 1, a[0]}
Pint := ADDRESS( i );            {Causes a warning message
                                  since i is not VOLATILE}
Pint^ := 2;

{Compiler may assume i = 0 and a[I] = a[0],
May make no assumptions about j}
WRITELN( i, j, Pint^, a[i] );  {Actual values are 2, 1, 2, a[2]}
```

This example assigns values to the variables i and j and to the newly
created variable Pint^. The comments illustrate the difference between
the assumptions the compiler can legally make about the values of the
variables and the values actually contained in the variables. The compiler's
assumption about the value of i was incorrect because the value of i changed
unexpectedly. The ADDRESS(i) call caused Pint to point to i (that is, Pint^
and i became the same variable). When Pint^ was assigned the value 2, the
variable i also received the value 2. Since i had been initialized to 0 and
was not directly referred to in the rest of the program, the compiler assumed
that a reference to i at this point would be equivalent to a reference to
0. Likewise, the compiler also assumed that a reference to a[i] would be
equivalent to a reference to a[0]. In fact, however, when execution ceases,
the value of i is 2 and the value of a[i] is the value of a[2].

Depending on the optimizations the compiler made based on the value of i,
any operations performed after the unanticipated assignment to i could yield
unexpected results. Because j was declared VOLATILE, the compiler did not
optimize code based on the value of j. Therefore, any reference to j yields the
expected results.

The ADDRESS(i) call in this program causes a warning message. The VAX
Pascal compiler assumes that pointer variables point only to variables in
heap-allocated storage and not to statically allocated, nonvolatile variables
such as i. So, ADDRESS(i) in this case differs from the expected usage.

For More Information:

- On use of VOLATILE with the ASYNCHRONOUS attribute
  (Section 10.2.2)

- On the VOLATILE attribute or on exception handlers (*VAX Pascal
  Reference Supplement for VMS Systems*)

0.2.42 WEAK_EXTERNAL

The WEAK_EXTERNAL attribute specifies that a variable or routine is not critical to the linking operation. To resolve a weak reference, the linker searches only the named input modules.

WEAK_EXTERNAL [[(identifier)]]

You can specify an identifier with this attribute to indicate the name by which the corresponding object is known to the linker. Compilation units cannot have the EXTERNAL or WEAK_EXTERNAL attribute.

For More Information:

* On the EXTERNAL attribute (Section 10.2.13)

* On linking (*VAX Pascal Reference Supplement for VMS Systems*)

0.2.43 WEAK_GLOBAL

The WEAK_GLOBAL attribute specifies that an object is linked only when it is specifically included in the linking operation. To resolve a weak reference, the linker searches only the named input modules.

WEAK_GLOBAL [[(identifier)]]

You can specify an identifier to indicate the name by which the corresponding object is known to the linker.

For More Information:

* On the GLOBAL attribute (Section 10.2.15)

* On linking (*VAX Pascal Reference Supplement for VMS Systems*)

0.2.44 WORD

The WORD attribute specifies the amount of storage in words to be received by the object.

WORD [[(n)]]

The optional constant n indicates the number of word storage units.

For More Information:

- On VAX Pascal size rules (Section 10.2.5)
- On default sizes according to type (*VAX Pascal Reference Supplement for VMS Systems*)

10.2.45 WRITEONLY

The WRITEONLY attribute specifies that an object can have values assigned to it but cannot be read by a program.

Usage and Default Information:

- This attribute can be applied to variables, formal parameters, the base types of pointer variables, and components of structured variables.
- By default, objects can be both read and written.
- A write-only object cannot be used in expressions.
- A write-only component in an object of a structured type prohibits the object from being read.
- A write-only actual variable parameter can be passed only to a formal variable parameter that is write-only.
- A pointer expression whose base type is write-only is assignment compatible only with a pointer variable whose base type is write-only.

Consider the following:

```
PROGRAM SAMPLE;
TYPE
   W_Only = [WRITEONLY] INTEGER;
VAR
   Writ_Int : W_Only;
   Norm_Int : INTEGER;

PROCEDURE Try_Access( VAR Write_Param : W_Only );  EXTERNAL;

{In the executable section:}
Writ_Int := SQR( Norm_Int );
Try_Access( Writ_Int );
```

This example shows legal statements involving write-only variables. The write-only variable Writ_Int is assigned the result of the square root operation, and is then passed as an actual parameter to a write-only formal parameter.

For More Information:

For information on the READONLY attribute, see Section 10.2.33.

10.3 Attribute Classes

An attribute class can consist of a single attribute or of several attributes
with a common characteristic. Table 10–4 lists the classes and their
attributes.

Table 10–4: Attribute Classes

Class	Attributes	Description of Attributes
Alignment	ALIGNED, UNALIGNED	Indicate whether the object should be aligned on a specific address boundary in memory.
Allocation	AT, AUTOMATIC, COMMON, STATIC, PSECT	Indicate the form of storage that the object should occupy.
Asynchronous	ASYNCHRONOUS	Indicates that the routine may be called by an asynchronous event, such as a condition handler.
Check	CHECK	Indicates error-checking options to be enabled or disabled.
Double precision	G_FLOATING, NOG_FLOATING	Indicate the type of precision to use for objects of type DOUBLE.
Environment	ENVIRONMENT	Indicates that VAX Pascal creates an environment file, which allows compilation units to share data definitions and declarations.
Hidden	HIDDEN	Indicates exclusion of a declaration or definition from a created environment file.
Ident	IDENT	Indicates the identification of a compilation unit to be passed to the linker.
Inherit	INHERIT	Indicates that the compilation unit can use the definitions and declarations specified in the inherited environment file.

Table 10–4 (Cont.): Attribute Classes

Class	Attributes	Description of Attributes
Initialize	INITIALIZE	Indicates that the procedure is to be called before execution of the main program.
Key	KEY	Indicates key information for a record field that is used when accessing data in an indexed file.
List	LIST	Indicates that the routine can be called with actual parameter lists of various lengths.
Optimization	OPTIMIZE, NOOPTIMIZE	Indicate whether VAX Pascal should optimize code.
Overlaid	OVERLAID	Indicates how storage should be allocated for variables.
Parameter passing	CLASS_A, CLASS_NCA, CLASS_S, IMMEDIATE, REFERENCE	Indicate the passing mechanism to be used for a parameter.
Pos	POS	Indicates that a record field should be forced to a specific bit position.
Read-only	READONLY	Indicates that the object can be read but cannot be written to.
Size	BIT, BYTE, WORD, LONG, QUAD, OCTA	Indicate the amount of storage to be reserved for the object.
Truncate	TRUNCATE	Indicates that the actual parameter list can be truncated at the position of this attribute in the formal parameter list.
Unbound	UNBOUND	Indicates that the routine does not access automatic variables outside its scope.
Unsafe	UNSAFE	Indicates that an object can accept values of any type without type checking.

Table 10–4 (Cont.): Attribute Classes

Class	Attributes	Description of Attributes
Value	VALUE	Indicates that the variable is a reference to an external constant or is the defining point of a global constant.
Visibility	LOCAL, EXTERNAL, GLOBAL, WEAK_EXTERNAL, WEAK_GLOBAL	Indicate the ability of an object to be shared by compilation units.
Volatile	VOLATILE	Indicates that the value of an object may change at unusual points in program execution.
Write-only	WRITEONLY	Indicates that the object can be written to but cannot be read.

Some attributes are allowed to appear on routine declarations, routine parameters, and compilation units. Table 10–5 lists these attribute classes.

Table 10–5: Attributes on Routines and Compilation Units

Class	Program Element		
	Routine Parameter	Routine	Compilation Unit
Allocation	No	Yes[1]	Yes[1]
Asynchronous	Yes	Yes	No
Check	No	Yes	Yes
Double-precision	No	No	Yes
Environment	No	No	Yes
Ident	No	No	Yes
Inherit	No	No	Yes
Initialize	No	Yes	No
List	Yes[2]	No	No

[1] PSECT is the only allocation attribute allowed.
[2] Allowed only on EXTERNAL routine definitions.

Table 10–5 (Cont.): Attributes on Routines and Compilation Units

Class	Program Element		
	Routine Parameter	Routine	Compilation Unit
Optimization	No	Yes	Yes
Overlaid	No	No	Yes
Truncate	Yes	No	No
Unbound	Yes	Yes	No
Visibility	No	Yes	Yes[3]

[3]EXTERNAL and WEAK_EXTERNAL are not allowed.

Attribute classes are allowed on various data items. Table 10–6 lists the classes that can be applied to various data items.

Table 10–6: Attributes on Data Items

Class	Data Item					
	Variable	Formal Parameter	Pointer Base Type	Component[1]	Function Result	Various Items[2]
Alignment	Yes[3]	Yes[4]	Yes[4]	Yes[5]	Yes	No
Allocation	Yes[6]	No	No	No	No	No
Hidden	Yes	No	Yes	No	No	No
Key	No	No	No	Yes[7]	No	No
List	No	Yes[8]	No	No	No	No

[1]Component of a record, array, VARYING OF CHAR string, or file (includes conformant parameters).
[2]Index of an array, tag field of a variant record (when no tag identifier is present), base type of a set, formal discriminant.
[3]Variables of nonstatic types must be byte aligned.
[4]UNALIGNED not allowed.
[5]Not allowed on components of files or VARYING OF CHAR strings.
[6]Not allowed on variables of nonstatic types.
[7]Allowed only on record fields (including the tag field of a variant record).
[8]Procedure parameters and conformant parameters are allowed only on EXTERNAL routines.

Table 10–6 (Cont.): Attributes on Data Items

Class	Data Item					
	Variable	**Formal Parameter**	**Pointer Base Type**	**Component[1]**	**Function Result**	**Various Items[2]**
Parameter-passing	No	Yes[9]	No	No	No	No
Pos	No	No	No	Yes[7]	No	No
Read-only	Yes	Yes	Yes	Yes	No	No
Size	Yes[6]	Yes [10]	Yes	Yes [11]	Yes	No
Truncate	No	Yes	No	No	No	No
Unsafe[6]	Yes	Yes[9]	Yes	Yes	Yes	Yes
Value	Yes [12]	No	No	No	No	No
Visibility[6]	Yes	No	No	No	No	No
Volatile	Yes	Yes	Yes	Yes	Yes	No
Write-only	Yes	Yes	Yes	Yes	No	No

[1] Component of a record, array, VARYING OF CHAR string, or file (includes conformant parameters).

[2] Index of an array, tag field of a variant record (when no tag identifier is present), base type of a set, formal discriminant.

[6] Not allowed on variables of nonstatic types.

[7] Allowed only on record fields (including the tag field of a variant record).

[9] Not allowed on conformant VARYING parameters; not allowed on schematic parameters.

[10] Not allowed on conformant parameters; not allowed on schematic parameters.

[11] Not allowed on components of files or VARYING OF CHAR strings, or on structured types with file components.

[12] Not allowed on variables larger than 32 bits or structured variables.

<div align="right">

Chapter 11

Directives

</div>

This chapter provides information on the following:

- The %INCLUDE directive (Section 11.1)
- The %DICTIONARY directive (Section 11.2)
- The %TITLE and %SUBTITLE directives (Section 11.3)

11.1 %INCLUDE Directive

The %INCLUDE directive inserts the contents of a file at the location of the directive in the code, and has the following form:

%INCLUDE 'file-spec [[/[[NO]]LIST]]'

file-spec
environment specific (default)
The name of the file to be included.

/[[NO]]LIST
/LIST (default)
The /LIST qualifier indicates that the included file should be printed in the listing of the program if a listing is being generated. If not specified, the default is determined by the use of compilation switches. Use of this parameter overrides compilation switches.

This directive can appear anywhere that a comment is legal.

In the following example, the %INCLUDE directive specifies the file CONDEF.PAS, which contains constant definitions:

Main Pascal Program:

```
PROGRAM Student_Courses ( INPUT, OUTPUT, Sched );
CONST
    %INCLUDE 'CONDEF.PAS/LIST'
TYPE
    Schedules = RECORD
        Year    : ( Fr, So, Jr, Sr );
        Name    : PACKED ARRAY[1..30] OF CHAR;
        Parents : PACKED ARRAY[1..40] OF CHAR;
        College : ( Arts, Engineering, Architecture,
                    Agriculture, Hotel );
        END;
```

File CONDEF.PAS:

```
Max_Class = 300;
N_Profs   = 140;
Frosh     = 3000;
```

The main program Student_Courses is compiled as though it were written as follows:

```
PROGRAM Student_Courses ( INPUT, OUTPUT, Sched );
CONST
    Max_Class = 300;
    N_Profs   = 140;
    Frosh     = 3000;

    Schedules = RECORD
        Year    : ( Fr, So, Jr, Sr );
        Name    : PACKED ARRAY[1..30] OF CHAR;
        Parents : PACKED ARRAY[1..40] OF CHAR;
        College : ( Arts, Engineering, Architecture,
                    Agriculture, Hotel );
        END;
```

You can use the %INCLUDE directive in another included file; however, two files cannot attempt to include each other.

A file included at the outermost level of a program is said to be included at the first level. A file included by a first-level file is said to be included at the second level, and so on. In general, a program may not include any files beyond the fifth level; it may not include any files beyond the fourth level if you have included a %DICTIONARY directive in the fourth level. Nesting levels may be further restricted by the number of files you are allowed to have open at one time. Figure 11–1 illustrates the legal levels of included files.

Figure 11–1: %INCLUDE File Levels

```
PROGRAM P
%INCLUDE 'A.PAS'    { level 1}

        A.PAS
        { TYPE definitions }
        %INCLUDE 'B.PAS'    { level 2 }

            B.PAS
            { VAR declarations }

    %INCLUDE 'C.PAS'    { level 2 }

        C.PAS
        { CONST definitions }
        %INCLUDE 'D.PAS'    { level 3 }

            D.PAS
            { VAR declarations }
            %INCLUDE 'E.PAS'    { level 4 }

                E.PAS
                { FUNCTION declaration }
                %INCLUDE 'F.PAS'    { level 5 }

                    F.PAS
                    { may not have any
                      included files }
```
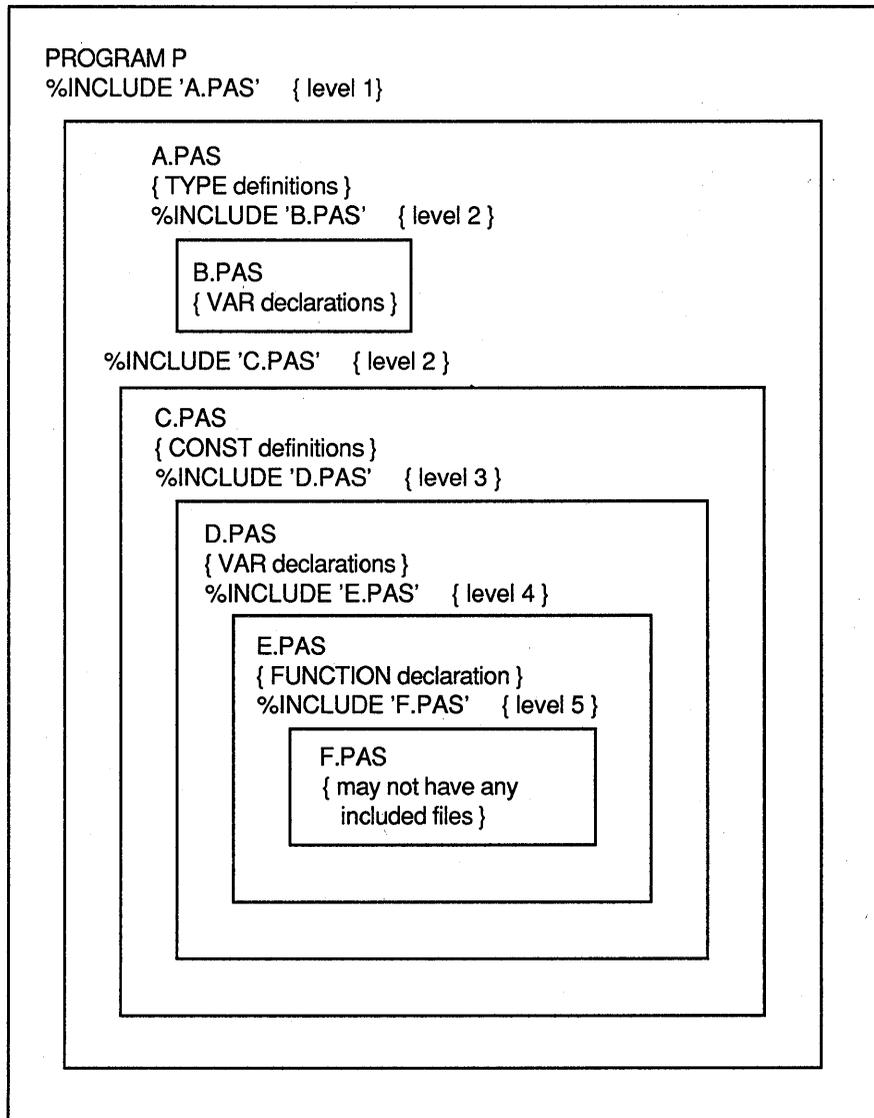
ZK–0285–GE

For More Information:

- On the Common Data Dictionary (CDD) (Section 11.2)
- On default file specifications and on including text libraries (*VAX Pascal Reference Supplement for VMS Systems*)

11.2 %DICTIONARY Directive

The %DICTIONARY directive allows access to data definitions stored in the VAX Common Data Dictionary (CDD), which is a product that must be purchased separately and may not be available on your environment; the directive has the following form:

%DICTIONARY 'cdd-path-name [[/[[NO]]LIST]] '

cdd-path-name
A character string that represents the full or relative path name of a CDD record description to be extracted. The resulting path name must conform to the rules for forming VAX CDD path names.

A full path name is one that begins with CDD$TOP and specifies the names of all its descendants; it is a complete path to the record definition. Descendant names are separated from each other by a period.

A relative path name begins with any generation other than CDD$TOP, and specifies the names of the descendants after that point. You can create a relative path by establishing a default directory with a logical name.

/[[NO]]LIST
/LIST (default)
Indicates that the included declarations should be printed in the listing of the program if a listing is being generated. If not specified, the default is determined by compilation switches. Use of this parameter overrides compilation switches.

For More Information:

For information on using the Common Data Dictionary with VAX Pascal, see the *VAX Pascal Reference Supplement for VMS Systems*.

11.3 %TITLE and %SUBTITLE Directives

The %TITLE and %SUBTITLE directives allow you to specify a compile-time string expression for the listing title and subtitle lines; they have the following form:

```
%TITLE 'character string'
%SUBTITLE 'character string'
```

The compiler listing header includes the %TITLE and %SUBTITLE strings in the title and subtitle sections. If you do not specify these directives, VAX Pascal fills the %TITLE field with blanks and the first %SUBTITLE field with 'source listing'. If a specified character string is too long to fit in the predefined title and subtitle sections, the string will be truncated on the right without warning.

If a %TITLE directive appears on the first line of a page, it sets the title area for the current page and any following pages until the compiler encounters another %TITLE directive. If the %TITLE directive does not appear on the first line of a page, then the title area is not set until the next page.

The %SUBTITLE directive affects only the subtitle area in the source listing section. If a %SUBTITLE directive appears on the first or second line of a page, then the subtitle area is set for the current page. If the %SUBTITLE directive does not appear in the first two lines of a page, then the subtitle area is not set until the next page.

If either of these directives is used and if a listing is being generated, VAX Pascal generates a table of contents page by default. It appears first in the listing, preceding the source listing section. To disable the table of contents option, you must use a compilation switch.

For More Information:

For information on creating listings and on using compilation switches, see the *VAX Pascal Reference Supplement for VMS Systems*.

Appendix A

ASCII Character Set

Table A–1 summarizes the ASCII character set. Each element of the character set is a constant of the predefined VAX Pascal type CHAR. An ASCII decimal number in Table A–1 is the same as the ordinal value (as returned by the VAX Pascal ORD function) of the associated character in the type CHAR.

Note that VAX Pascal uses an extended implementation of the ASCII character set. The extended characters, which do not appear in Table A–1, have the following decimal values:

ASCII	Meaning
128–160	Extended control characters
161–254	Extended graphics characters
255	Eight binary one values

Table A–1: The ASCII Character Set

ASCII Decimal Number	Character	Meaning
0	NUL	Null
1	SOH	Start of heading
2	STX	Start of text
3	ETX	End of text

Table A–1 (Cont.): The ASCII Character Set

ASCII Decimal Number	Character	Meaning
4	EOT	End of transmission
5	ENQ	Enquiry
6	ACK	Acknowledgement
7	BEL	Bell
8	BS	Backspace
9	HT	Horizontal tab
10	LF	Line feed
11	VT	Vertical tab
12	FF	Form feed
13	CR	Carriage return
14	SO	Shift out
15	SI	Shift in
16	DLE	Data link escape
17	DC1	Device control 1
18	DC2	Device control 2
19	DC3	Device control 3
20	DC4	Device control 4
21	NAK	Negative acknowledgement
22	SYN	Synchronous idle
23	ETB	End of transmission block
24	CAN	Cancel
25	EM	End of medium
26	SUB	Substitute
27	ESC	Escape
28	FS	File separator
29	GS	Group separator
30	RS	Record separator

Table A–1 (Cont.): The ASCII Character Set

ASCII Decimal Number	Character	Meaning
31	US	Unit separator
32	SP	Space or blank
33	!	Exclamation point
34	"	Quotation mark
35	#	Number sign
36	$	Dollar sign
37	%	Percent sign
38	&	Ampersand
39	'	Apostrophe
40	(	Left parenthesis
41	)	Right parenthesis
42	*	Asterisk
43	+	Plus sign
44	,	Comma
45	-	Minus sign or hyphen
46	.	Period or decimal point
47	/	Slash
48	0	Zero
49	1	One
50	2	Two
51	3	Three
52	4	Four
53	5	Five
54	6	Six
55	7	Seven
56	8	Eight
57	9	Nine

Table A–1 (Cont.): The ASCII Character Set

ASCII Decimal Number	Character	Meaning
58	:	Colon
59	;	Semicolon
60	<	Left angle bracket
61	=	Equal sign
62	>	Right angle bracket
63	?	Question mark
64	@	At sign
65	A	Uppercase A
66	B	Uppercase B
67	C	Uppercase C
68	D	Uppercase D
69	E	Uppercase E
70	F	Uppercase F
71	G	Uppercase G
72	H	Uppercase H
73	I	Uppercase I
74	J	Uppercase J
75	K	Uppercase K
76	L	Uppercase L
77	M	Uppercase M
78	N	Uppercase N
79	O	Uppercase O
80	P	Uppercase P
81	Q	Uppercase Q
82	R	Uppercase R
83	S	Uppercase S
84	T	Uppercase T

Table A–1 (Cont.): The ASCII Character Set

ASCII Decimal Number	Character	Meaning
85	U	Uppercase U
86	V	Uppercase V
87	W	Uppercase W
88	X	Uppercase X
89	Y	Uppercase Y
90	Z	Uppercase Z
91	[	Left square bracket
92	\	Back slash
93	]	Right square bracket
94	^ or ↑	Circumflex or up arrow
95	← or _	Back arrow or underscore
96	`	Grave accent
97	a	Lowercase a
98	b	Lowercase b
99	c	Lowercase c
100	d	Lowercase d
101	e	Lowercase e
102	f	Lowercase f
103	g	Lowercase g
104	h	Lowercase h
105	i	Lowercase i
106	j	Lowercase j
107	k	Lowercase k
108	l	Lowercase l
109	m	Lowercase m
110	n	Lowercase n
111	o	Lowercase o

Table A–1 (Cont.):   The ASCII Character Set

ASCII Decimal Number	Character	Meaning
112	p	Lowercase p
113	q	Lowercase q
114	r	Lowercase r
115	s	Lowercase s
116	t	Lowercase t
117	u	Lowercase u
118	v	Lowercase v
119	w	Lowercase w
120	x	Lowercase x
121	y	Lowercase y
122	z	Lowercase z
123	{	Left brace
124	|	Vertical line
125	}	Right brace
126	~	Tilde
127	DEL	Delete

Appendix B

Language Syntax Summary

The diagrams in this section illustrate the syntax of the following items:

- Actual parameter list
- Array constructor (standard and nonstandard)
- Attribute list
- Binary digits
- Block
- Compilation unit
- Conformant parameter
- Decimal digits
- Declaration part
- Expression
- Factor
- Field list
- Formal parameter list
- Formal parameter section
- Function heading
- Hexadecimal digits
- Identifier
- Initial value
- Mechanism specifier
- Numeric constant
- Octal digits
- Primary

- Procedure heading
- Real constant
- Record constructor (standard and nonstandard)
- Routine declaration
- Set constructor
- Simple expression
- Simple statement
- Simple type
- Statement
- String constant
- Structured statement
- Term
- Type
- Variable

An example of how to interpret a diagram follows:

ZK–0129–GE

The diagram illustrates that the first character of an identifier can be either an underscore (_), a dollar sign ($), or a letter. The next character is chosen from the section labeled A and can be a digit, a letter, a dollar sign, or an underscore. Section A is repeated until the identifier is defined. Note that rounded symbols (circles or ovals) denote elements that must appear exactly as shown; rectangular symbols denote elements for which there is a separate diagram.

NOTE

A few of the rectangular elements contained in the diagrams do not have corresponding diagrams that expand on their meaning. Further information on these elements can be found in the syntax examples in the chapters of this manual.

actual parameter list
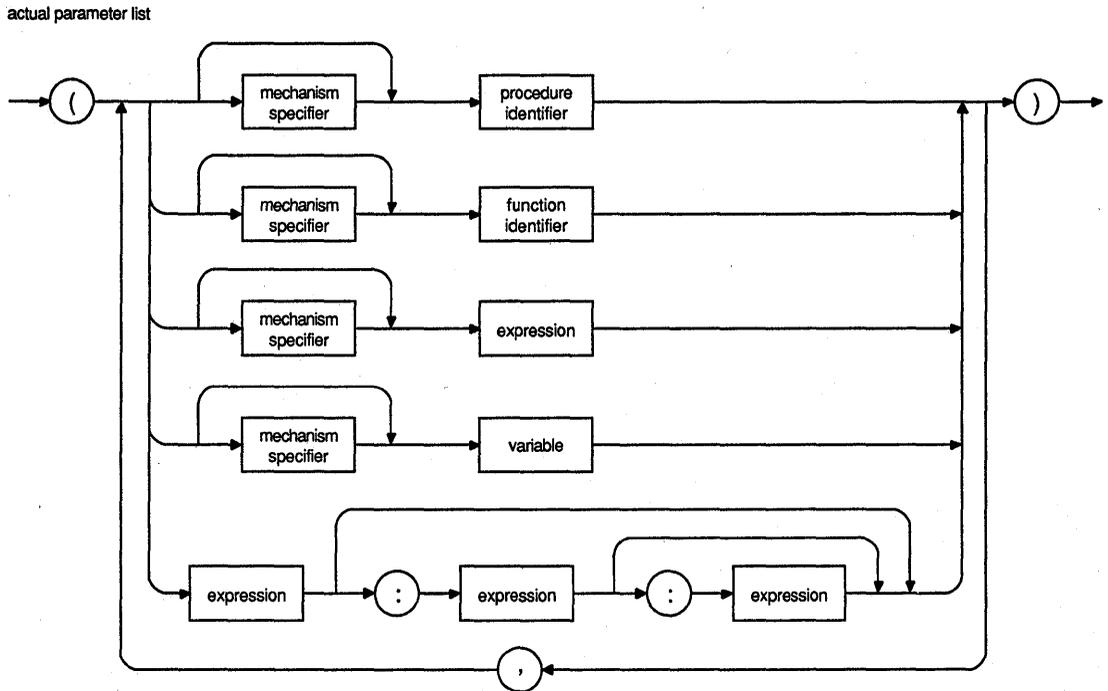
ZK–0571–GE

array constructor
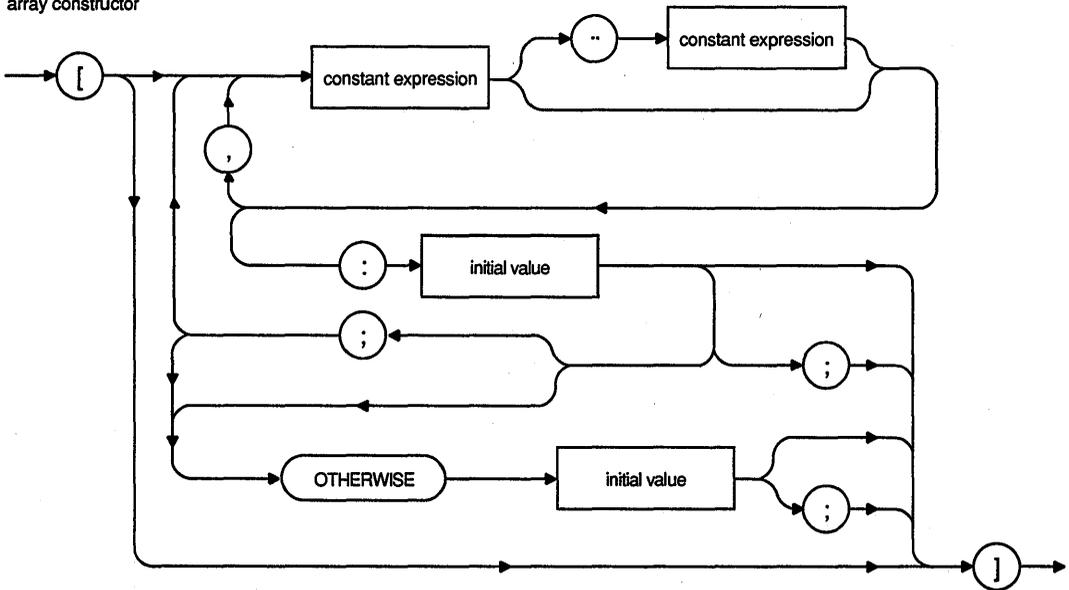
ZK–1397A–GE

nonstandard array constructor

ZK–0570–GE

attribute list

ZK–0131–GE

binary digits
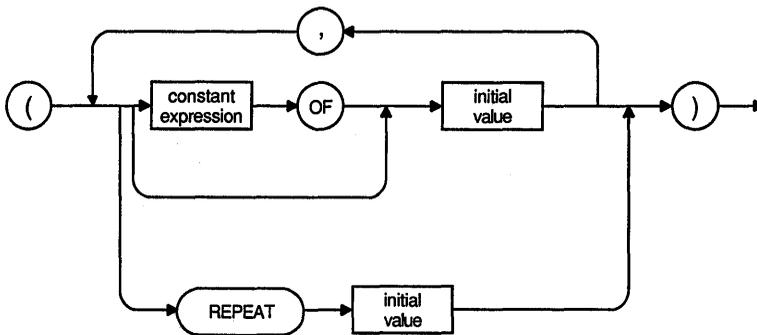
ZK–0132–GE

block
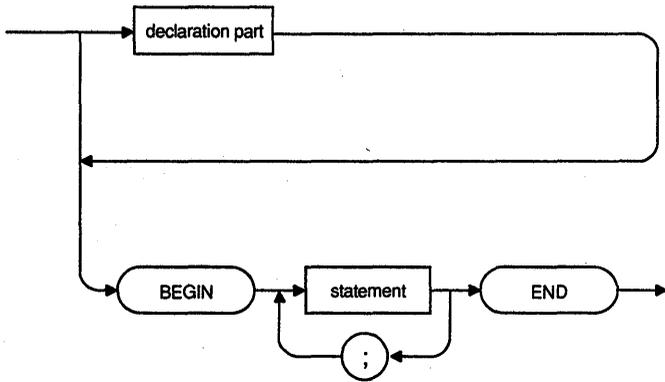
ZK–0107–GE

compilation unit
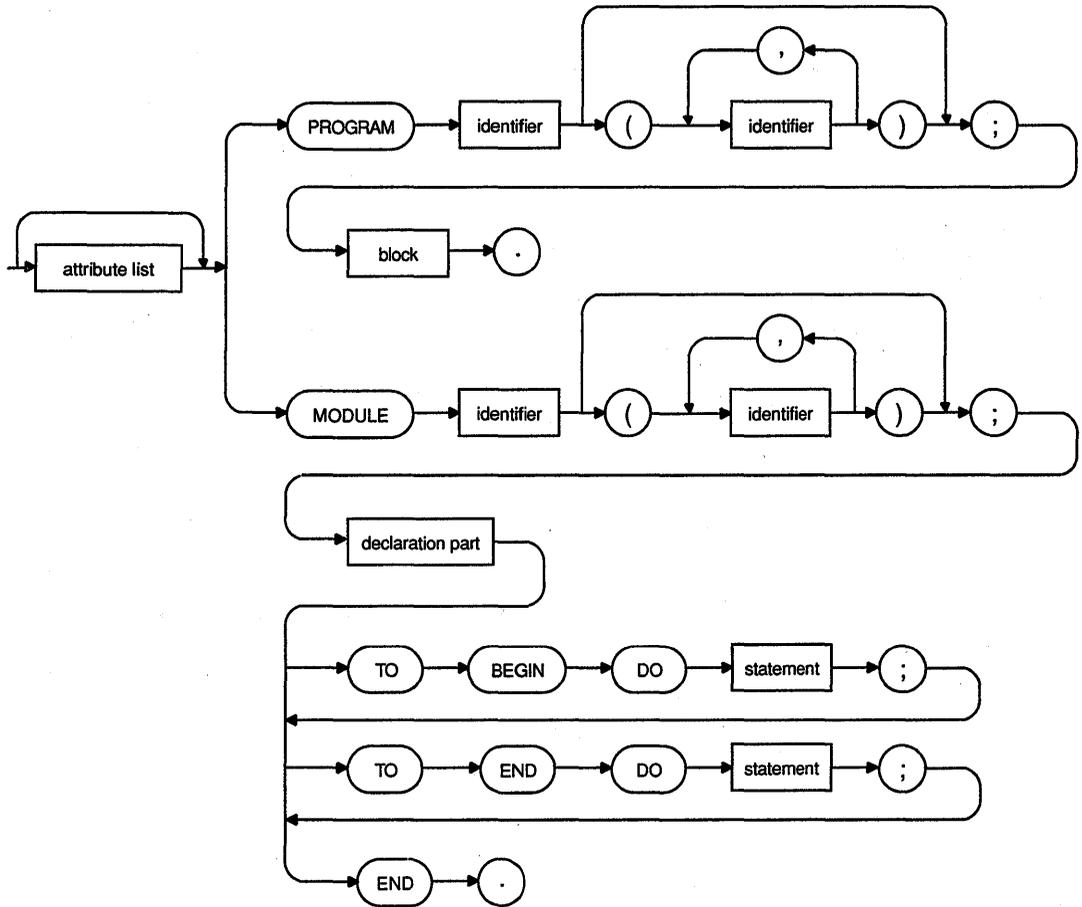
ZK–0111–GE

conformant parameter
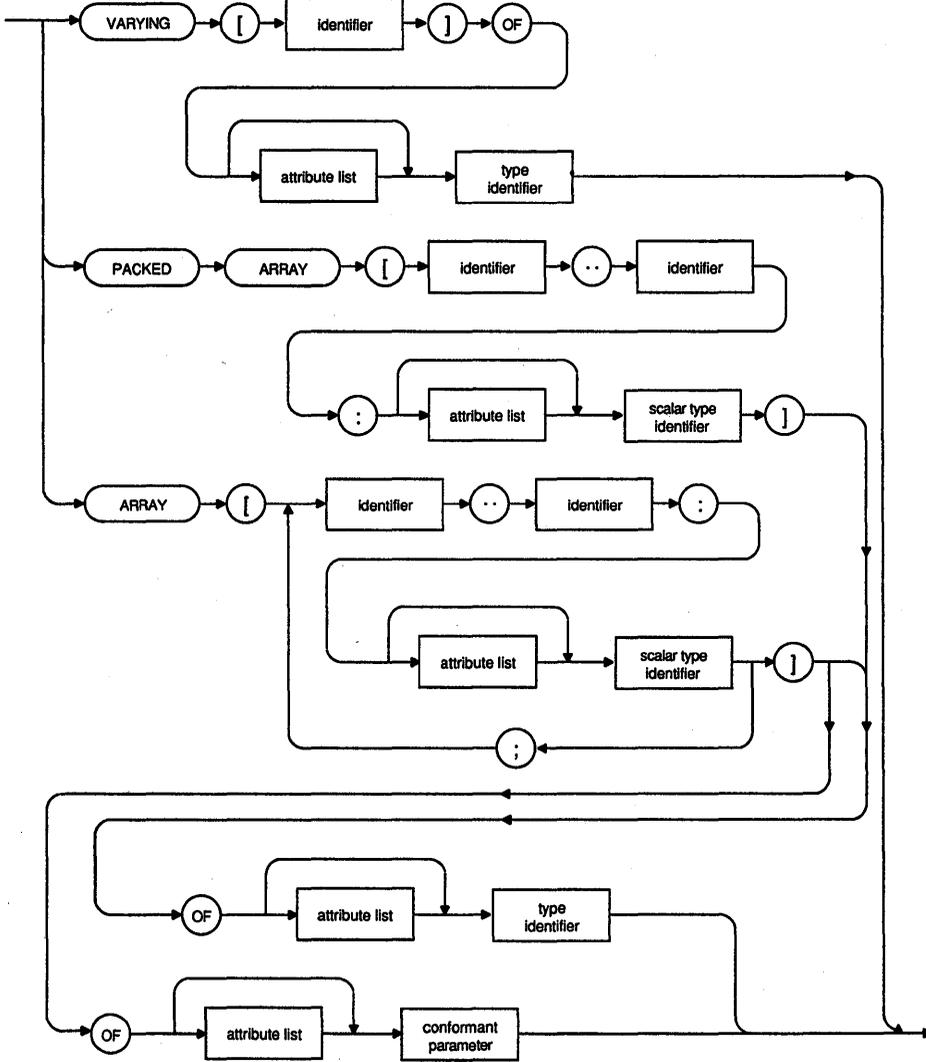
ZK-0133-GE

decimal digits

ZK-0566-GE

declaration part

LABEL → decimal digits / identifier

CONST → identifier = constant expression

TYPE → identifier (identifier : ordinal-type identifier) = type VALUE initial value

VAR → identifier : type := / VALUE initial value

VALUE → identifier := initial value

routine declaration
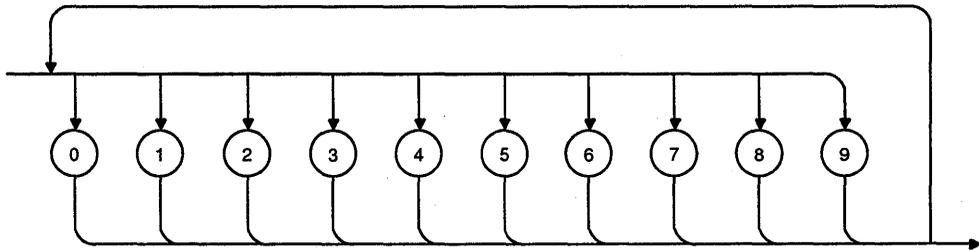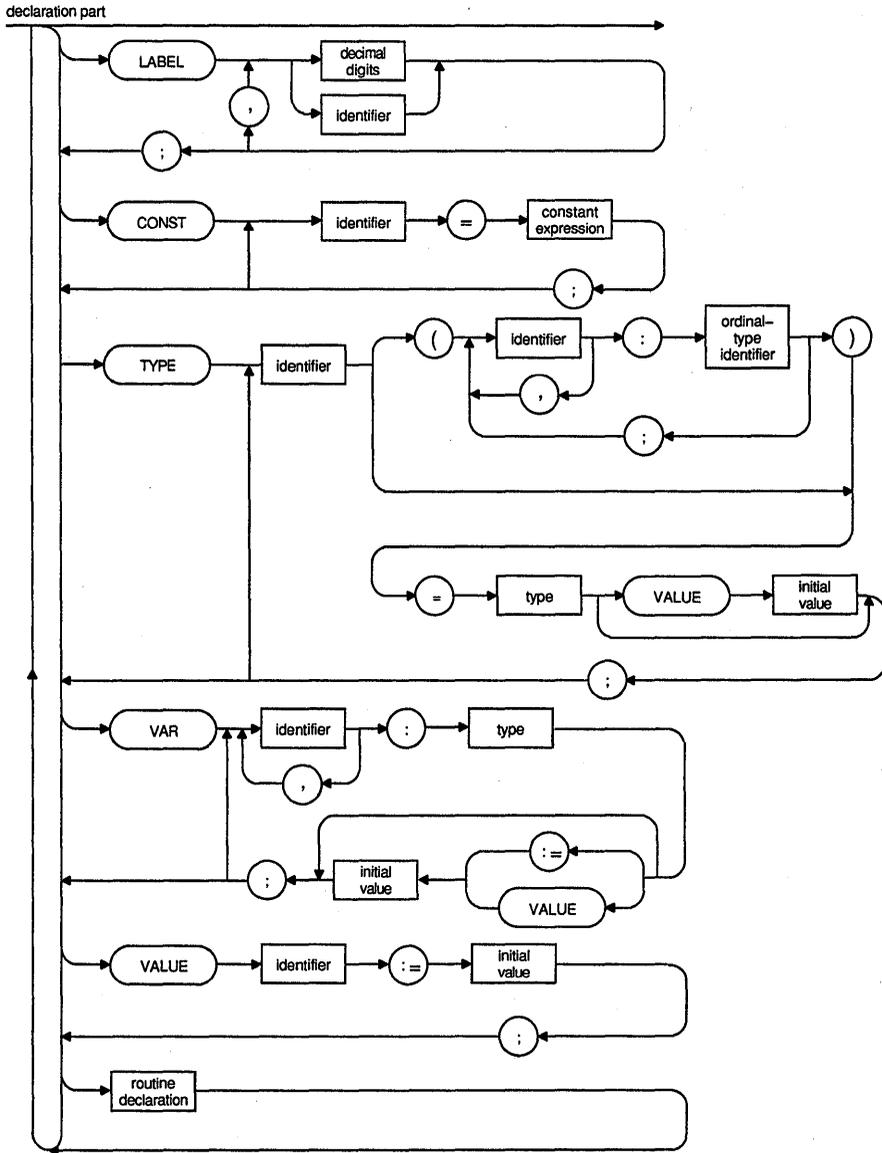
ZK–0109–GE

expression

ZK–0119–GE

factor

variable

constant
identifier

string
constant

real
constant

numeric
constant

function
identifier

actual
parameter
list

[expression]

,

. field identifier

^

(expression)

:: type
identifier

NOT factor

array type
identifier

array
constructor

record type
identifier

record
constructor

set type
identifier

set
constructor
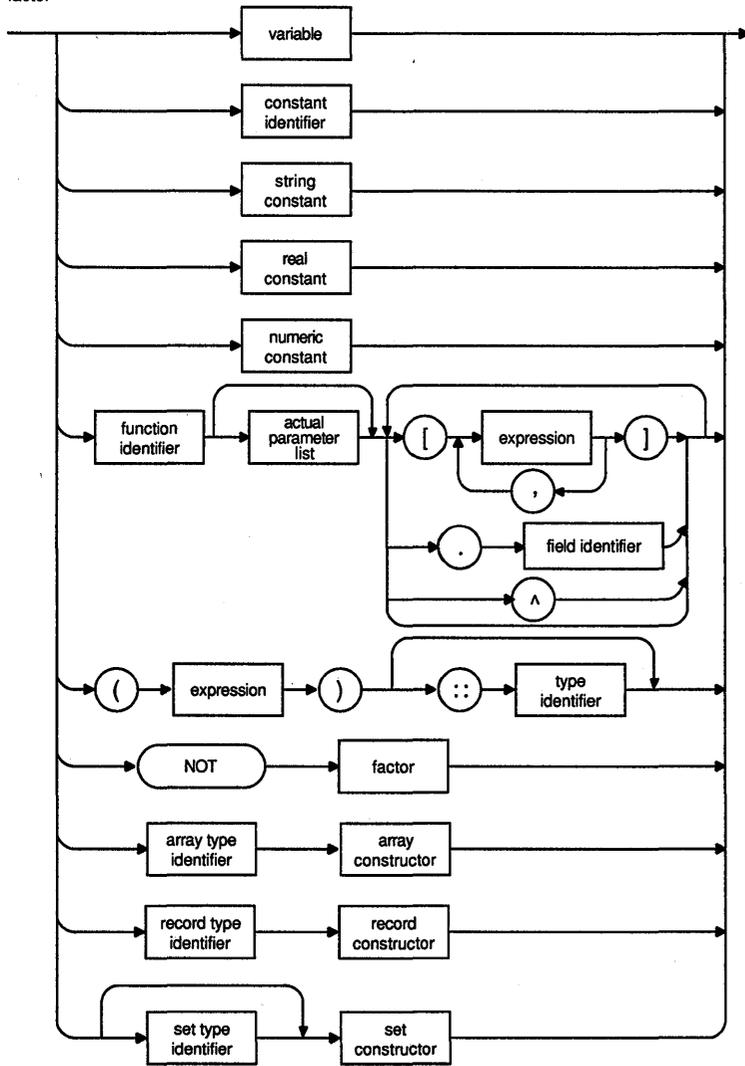
ZK–0115–GE

field list
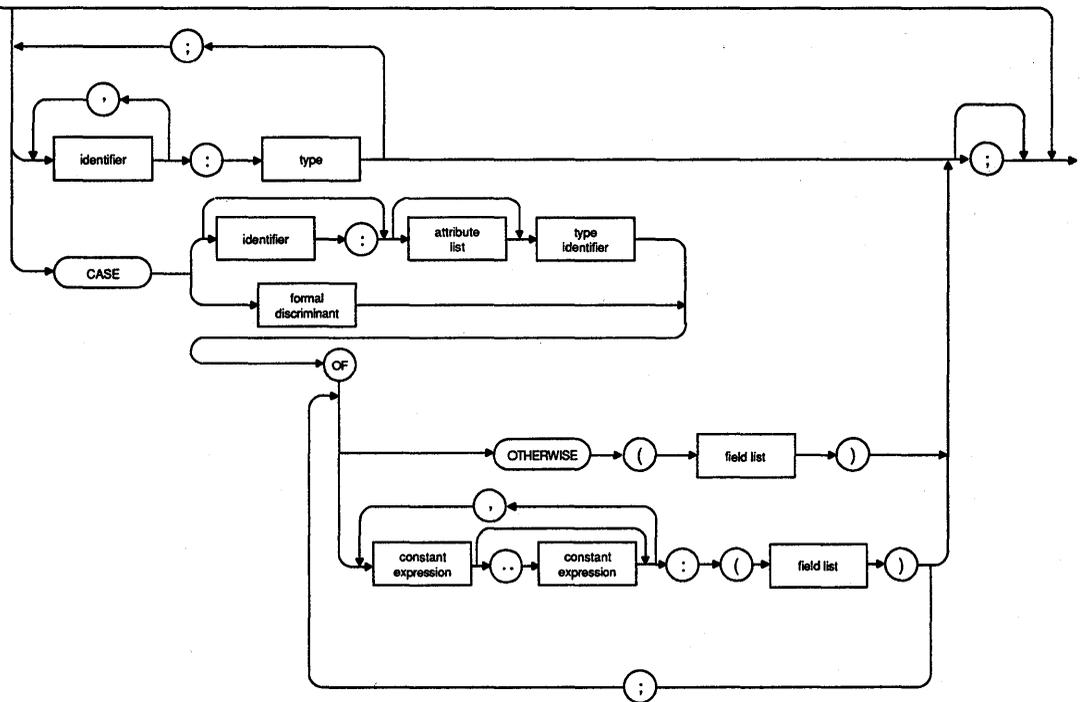
ZK–0134–GE

formal parameter list
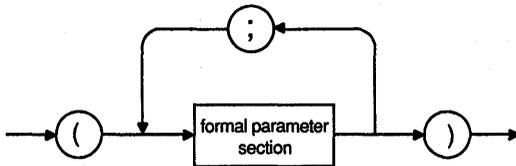
ZK–0568–GE

formal parameter section
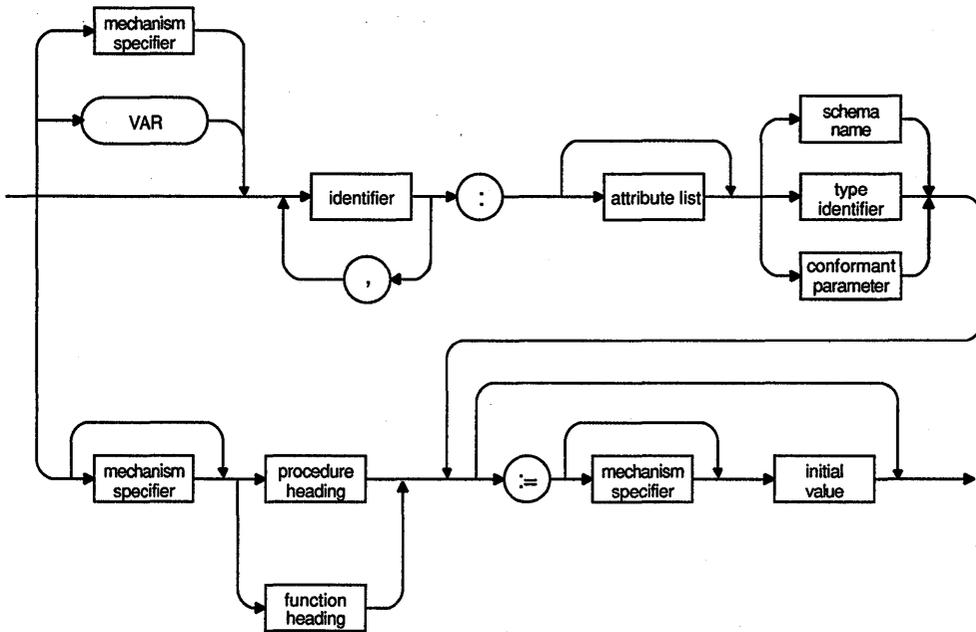
ZK-0136-GE

function heading

ZK-1035-GE

B–14 Language Syntax Summary

hexadecimal digits

ZK-0121-GE

identifier

ZK-0112-GE

initial value

```
                    ┌──────────────┐
              ┌────►│ nonstandard  ├────┐
              │     │    array     │    │
              │     │ constructor  │    │
              │     └──────────────┘    │
              │     ┌──────────────┐    │
              ├────►│ nonstandard  ├────┤
              │     │   record     │    │
              │     │ constructor  │    │
              │     └──────────────┘    │
              │     ┌──────────────┐    │
              ├────►│   constant   ├────┤
              │     │  expression  │    │
              │     └──────────────┘    │
              │     ┌──────────────┐    │
    ──────────┼────►│    array     ├────┼──X──►
              │     │ constructor  │    │
              │     └──────────────┘    │
              │     ┌──────────────┐    │
              ├────►│    record    ├────┤
              │     │ constructor  │    │
              │     └──────────────┘    │
              │     ┌──────────────┐    │
              ├────►│     set      ├────┤
              │     │ constructor  │    │
              │     └──────────────┘    │
              │       ╭──────────╮      │
              └──────►│   ZERO    ├──────┘
                      ╰──────────╯
```
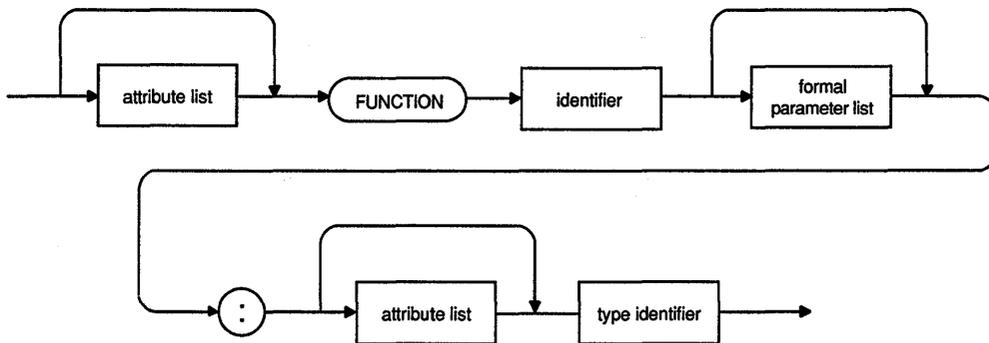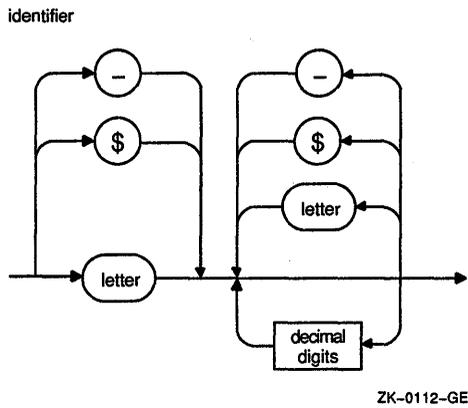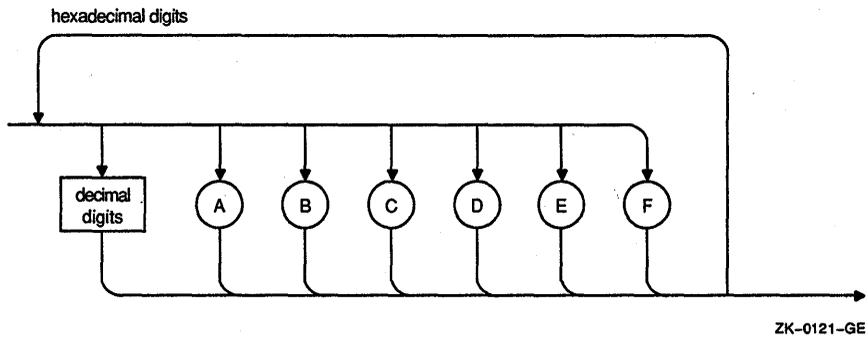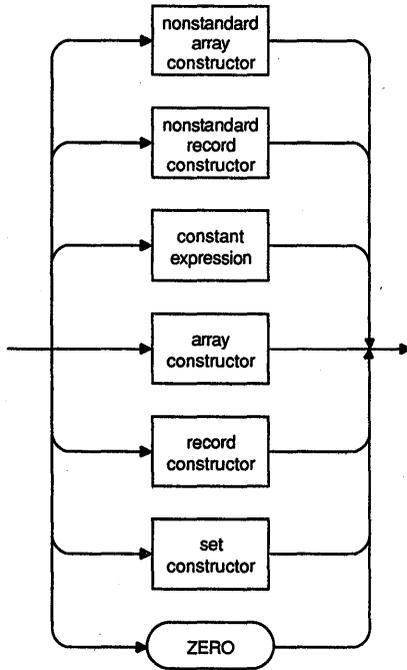
ZK–0565–GE

mechanism specifier

```
                   ╭──────────╮
        ┌─────────►│   %REF    ├─────────┐
        │          ╰──────────╯          │
        │          ╭──────────╮          ▲
        ├─────────►│  %IMMED   ├──────────┤
        │          ╰──────────╯          │
    ────┤          ╭──────────╮          │
        ├─────────►│  %DESCR   ├──────────┤
        │          ╰──────────╯          │
        │          ╭──────────╮          │
        └─────────►│ %STDESCR  ├──────────┘
                   ╰──────────╯
```

ZK–0564–GE

B–16 Language Syntax Summary
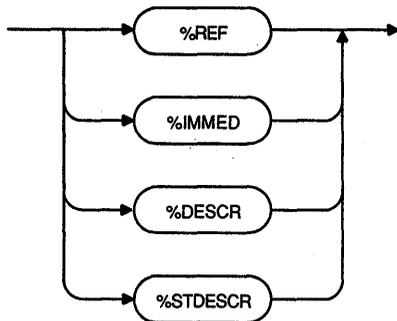
numeric constant

ZK-0114-GE

octal digits
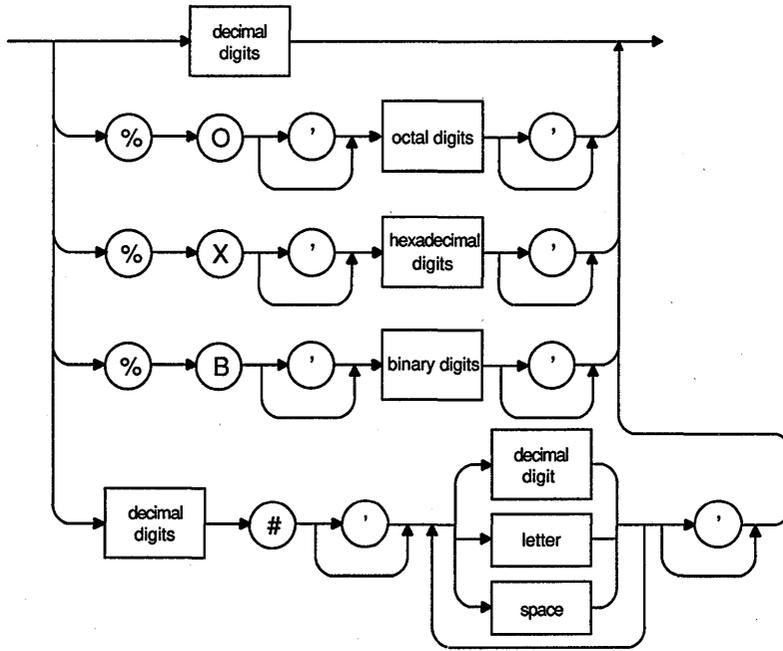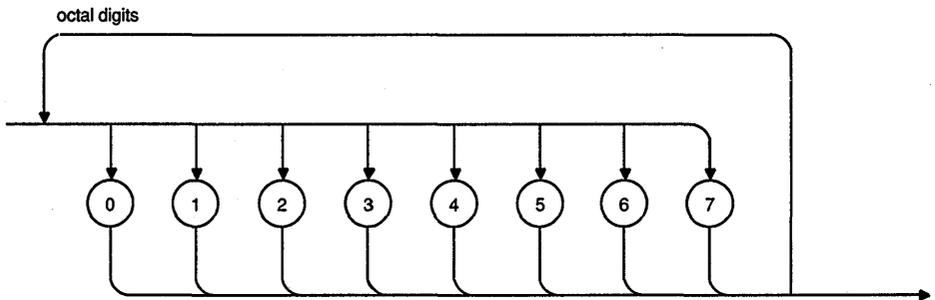
ZK-0120-GE

primary
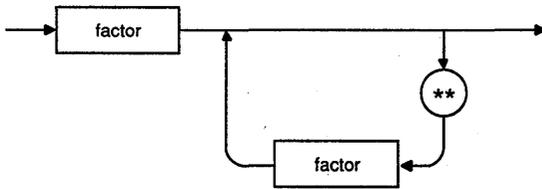
ZK–0116–GE

procedure heading
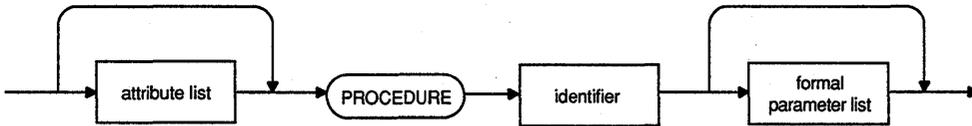
ZK–1034–GE

real constant
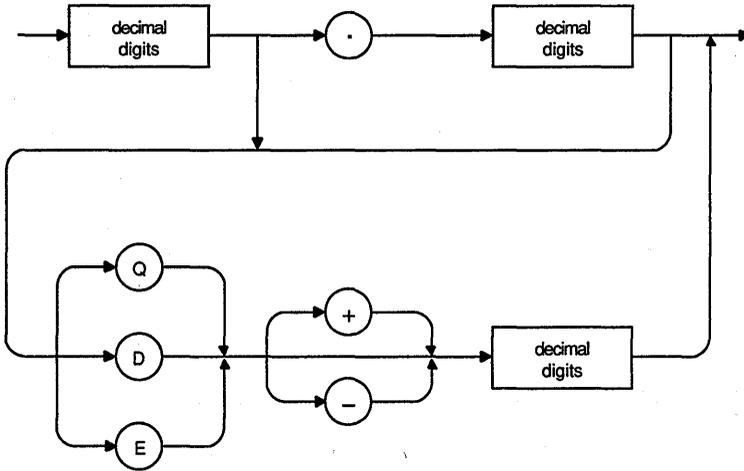
ZK–0113–GE

record constructor
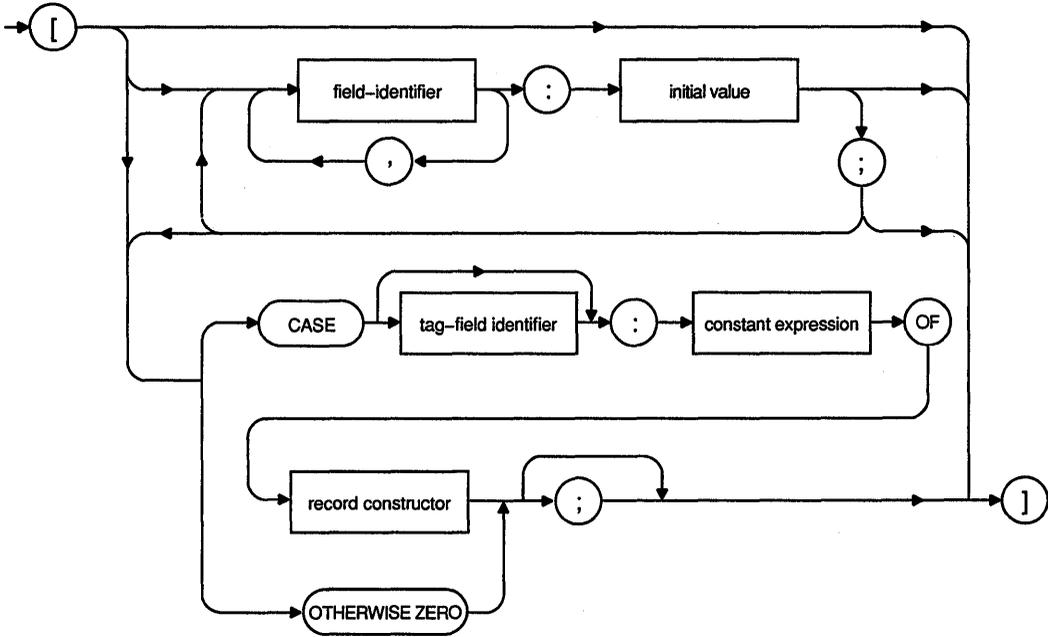
ZK–1398A–GE

nonstandard record constructor
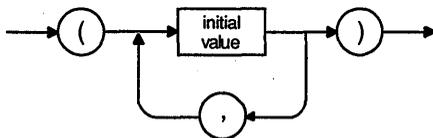
ZK–0567–GE

routine declaration
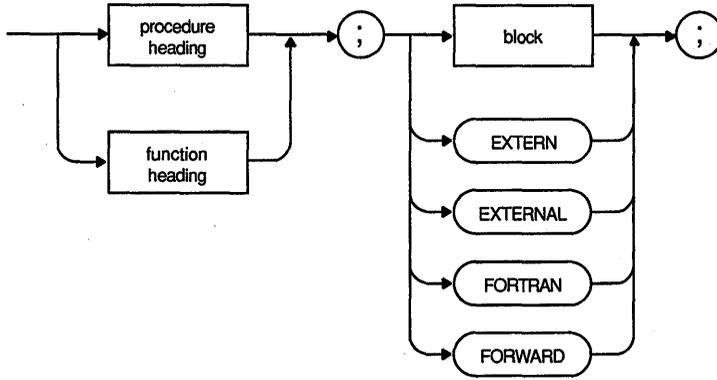
ZK–0130–GE

set constructor
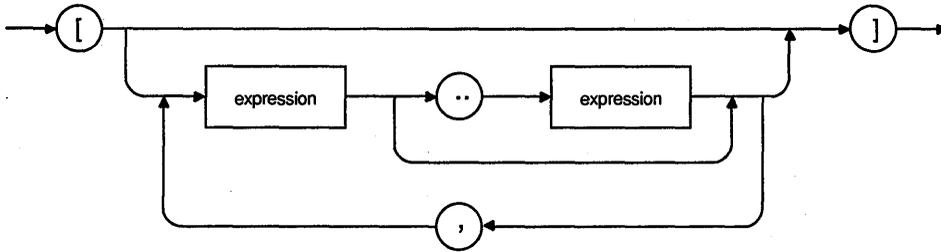
ZK–0569–GE

simple expression
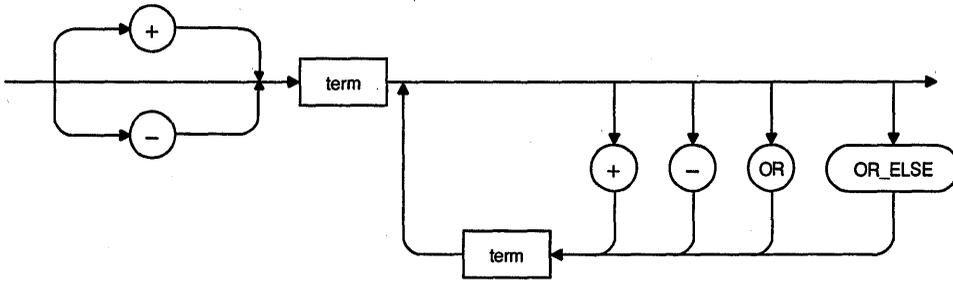
ZK–0118–GE

simple statement
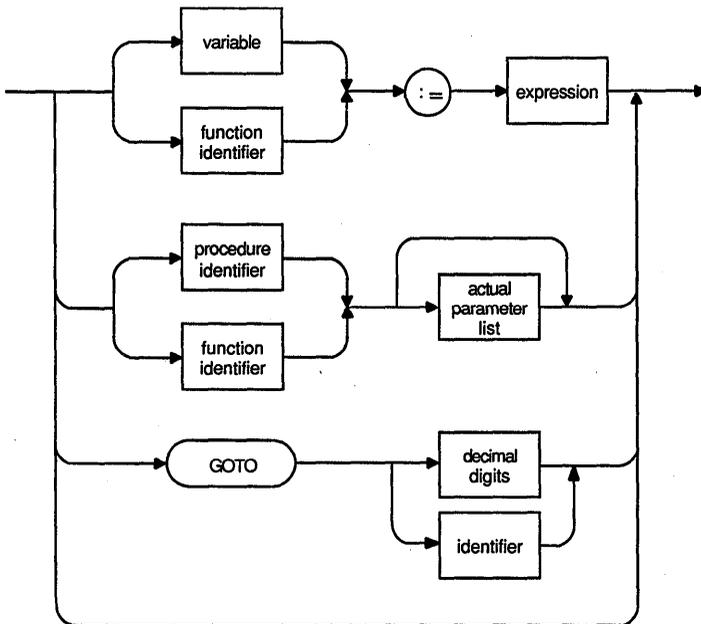
ZK–0108–GE

simple type
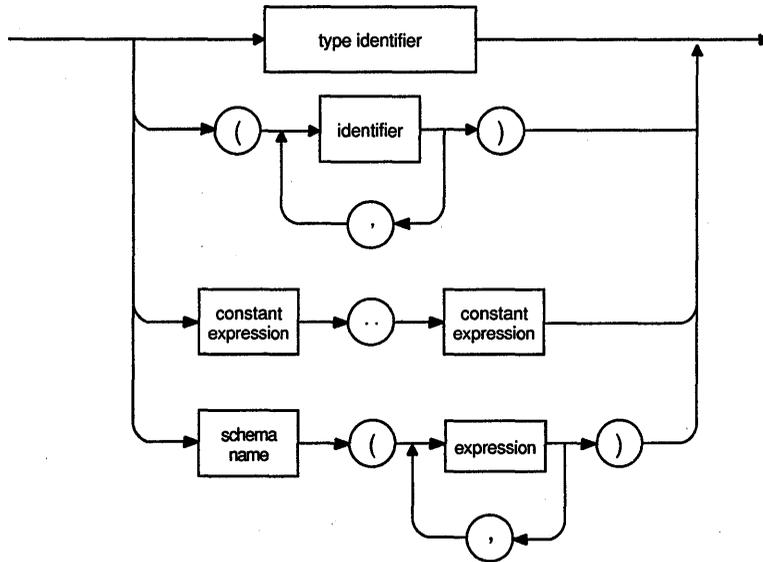
ZK–0124–GE

statement
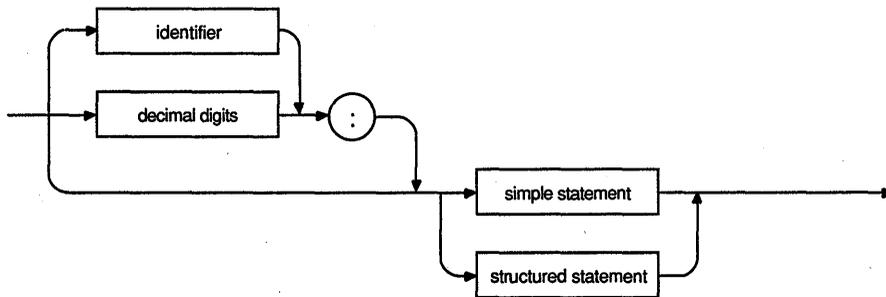
ZK–0137–GE

string constant
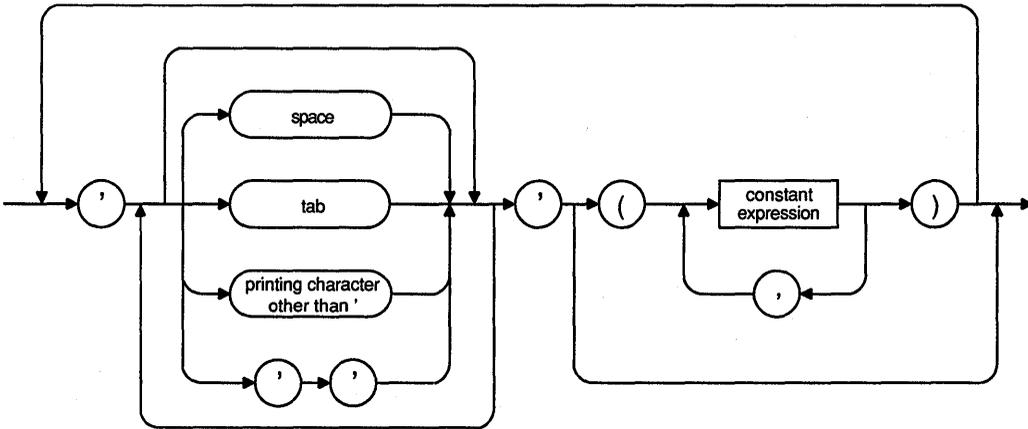
ZK–0572–GE

structured statement

ZK–0110–1–GE

B–24 Language Syntax Summary
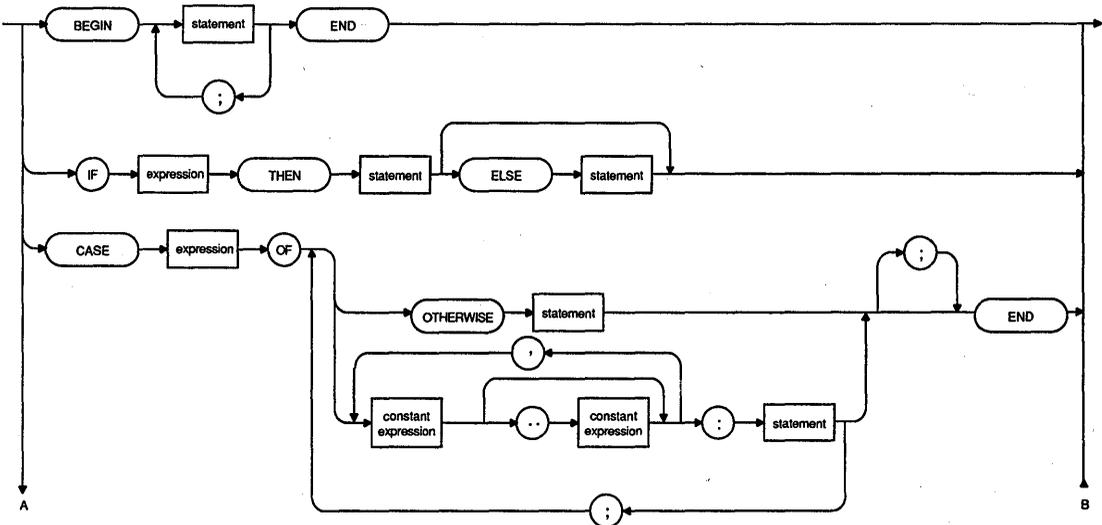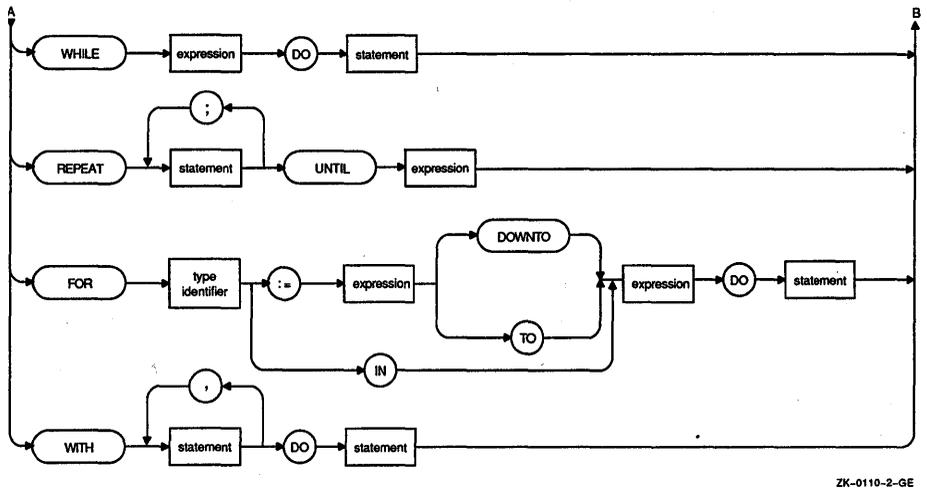
ZK-0110-2-GE

term

ZK-0117-GE

type
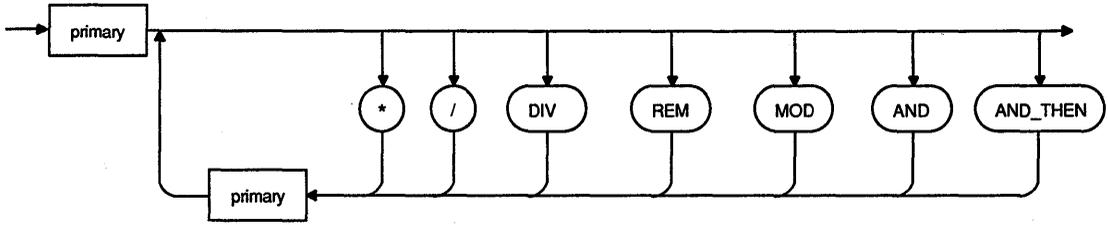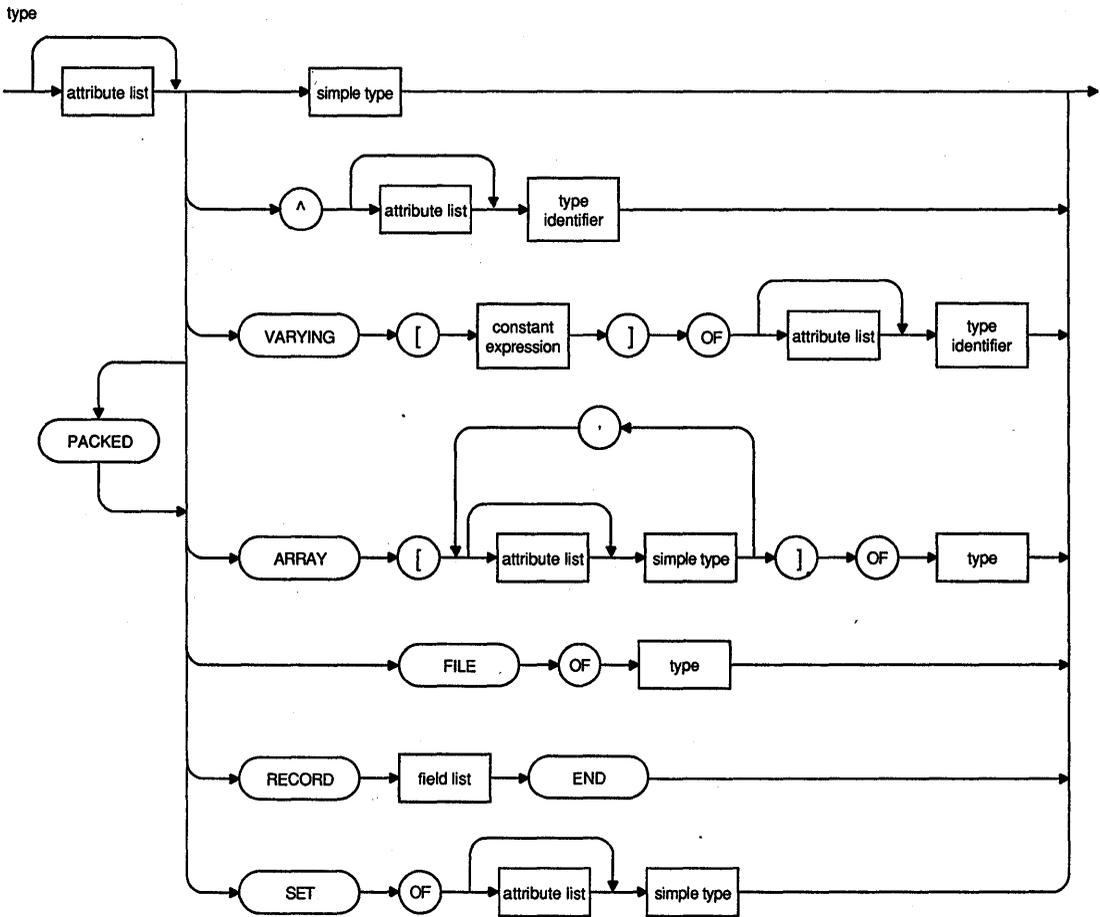
ZK–0125–GE

variable
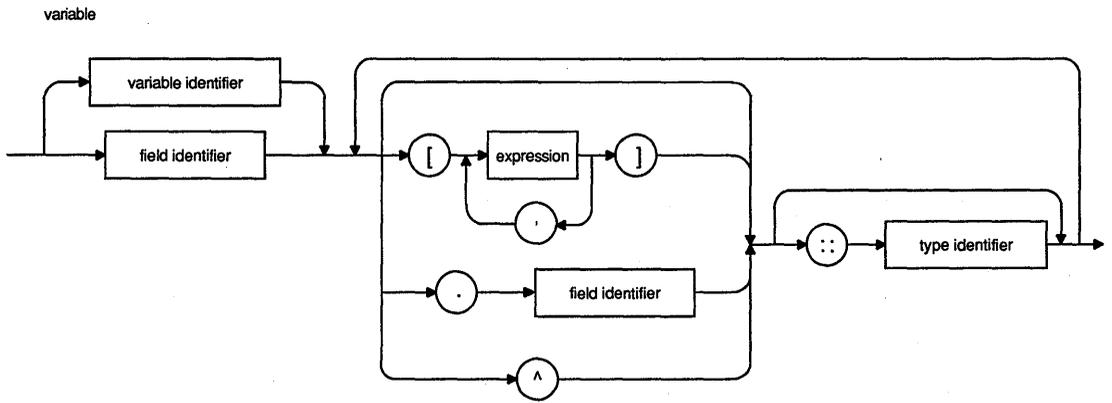
ZK–0127–GE

Compatibility of VAX Pascal Versions

This appendix provides the following information:

* Differences between Version 1.0 and subsequent versions (Section C.1)
* Differences between this version and past versions back to Version 2.0 (Section C.2)

This appendix describes the differences between VAX Pascal Version 1.0 and all subsequent higher versions. In this appendix, the term Version 2+ will refer to Version 2.0, Version 3.0, and Version 4.0.

NOTE

This appendix is intended for users migrating from Version 1.0 to Version 2+. If you are migrating from Version 2.0 to a higher version, you can disregard this appendix, as there have been no incompatible changes made between Version 2.0 and Version 4.0.

C.1 Differences Between Version 1.0 and Subsequent Versions

The differences between Version 1.0 and Version 2+ fall into three categories:

* Features that have been decommitted. The previous versions of these features are still supported in Version 2+ to allow you to run existing programs; however, it is recommended that you modify your programs to reflect the new changes.
* Features that are controlled by the /OLD_VERSION compile-time qualifier.

- Minor changes that are not likely to affect the vast majority of existing VAX Pascal programs.

If you modify a program that executed successfully under Version 1.0 of VAX Pascal, you should not make changes that conflict with the Version 2+ standard. If conflicts exist and you compile the program with Version 2+, one of two problems may result:

- You may get warning messages at compile time.
- The program may compile successfully but may not run.

If you must use language features that conflict with Version 2+, you can use the /OLD_VERSION qualifier at compile time to produce the desired results. The /OLD_VERSION qualifier and the conflicts that it resolves are described in Section C.1.2.

C.1.1 Decommitted Features

The following decommitted features are described in this section:

- Syntax of dynamic array parameters
- Predeclared functions LOWER and UPPER
- Printing of hexadecimal and octal values with the WRITE procedure
- Syntax of the OPEN procedure
- Specification of compiler qualifiers in the source code

C.1.1.1 Dynamic Array Parameters

Some programming applications require general routines that can process arrays with different bounds. Version 1.0 of VAX Pascal allows you to declare routines with dynamic array parameters. You can call the routine with arrays of different sizes, as long as their bounds are within those specified by the formal parameter.

For example, you can write a procedure that sums the components of a one-dimensional array. Each time you use the procedure, you might want to pass arrays with different bounds. Instead of declaring multiple procedures using arrays of each possible size, you could use a dynamic array parameter. The procedure will treat the formal parameter as though its bounds were those of the actual parameter.

In routines that contain dynamic array parameters, you use the predeclared functions LOWER and UPPER to return the lower and upper bounds of the actual array parameter (see Section C.1.1.2). An array parameter has the following form:

array-identifier,... : PACKED ARRAY [{index-type-identifier},...] OF type-identifier

Note that you must use a type identifier to specify the range of the indexes. You cannot use a subrange. The type identifier can be any of the predefined ordinal types (for example, INTEGER).

The components and indexes of the actual and formal dynamic array parameters must be of compatible types. The rules for dynamic array compatibility are identical to those for compatibility between other arrays, with one exception: the range of the index types of the actual array parameter must be within the range specified for the formal parameter.

The following differences exist between Version 1.0 and Version 2+ syntax:

- In Version 2+, dynamic arrays are known as conformant arrays, and the syntax that describes them is called a conformant array schema.

- The conformant array schema for Version 2+ requires that the upper and lower bounds of the conformant array parameter be declared with identifiers in the formal parameter list. You can then use these identifiers within the routine block to refer to the upper and lower bounds of the parameter.

- Version 2+ allows the type identifier of a conformant array parameter to be another conformant array schema.

C.1.1.2 LOWER and UPPER Functions

Version 1.0 of VAX Pascal includes the predeclared functions LOWER and UPPER, which you can use to determine the upper and lower bounds of dynamic array parameters (see Section C.1.1.1). Because the syntax of conformant array parameters has changed, these functions are no longer necessary. They are supported, however, for programs that use the old syntax. These functions have the following form:

LOWER(a [[, n]]) UPPER(a [[, n]])

The parameter a denotes an array variable; the optional parameter n is an integer constant that denotes a dimension of a. If you omit a value for the parameter n, it defaults to 1.0. The LOWER function returns the lower bound of the nth dimension of a; the UPPER function returns the upper bound of the nth dimension of a.

C.1.1.3 Printing Hexadecimal and Octal Values

Version 2+ provides the predeclared functions HEX and OCT, which return the hexadecimal and octal equivalents of the input value. You can use these functions in conjunction with the WRITE, WRITELN, and WRITEV procedures to print values in hexadecimal and octal notation.

The following formats of the WRITE procedure are used to print hexadecimal and octal values in Version 1.0:

```
WRITE (expression:field-width HEX,...)
WRITE (expression:field-width OCT,...)
```

expression
The value to be written. Arbitrary items (including pointers) can be written in hexadecimal or octal notation to text files.

field-width
A positive integer expression indicating the length of the print field.

For hexadecimal values, if the field width specified is less than eight characters, and the output value is greater than the field width, the value being printed is truncated on the left. If the field width is greater than eight characters, and the output value is less than the field width, the field is padded with blanks on the left.

For octal values, if the field width specified is less than 11 characters, and the output value is greater than the field width, the value being printed is truncated on the left. If the field width is greater than 11 characters, and the output value is less than the field width, the field is padded with blanks on the left.

Example 1
```
WRITE (Payroll:10 HEX);
```

The value of the variable Payroll is printed in a field of 10 hexadecimal characters.

Example 2
```
WRITE (Social_Security:14 OCT);
```

The value of the variable Social_Security is printed in a field of 14 octal characters.

C.1.1.4 The OPEN Procedure

The OPEN procedure opens a file and allows you to specify file parameters. Version 2+ includes new parameters and additional parameter values, and has changed some defaults. Table C–1 lists the file parameters available under Version 1.0, their possible values, and their defaults.

Table C–1: Summary of Version 1.0 OPEN Parameters

Parameter	Parameter Values	Default
History	OLD or NEW	NEW (OLD, if the file is opened with RESET).
Record length	Any positive integer	133 bytes.
Access method	DIRECT or SEQUENTIAL	SEQUENTIAL.
Record type	FIXED or VARIABLE	VARIABLE for new files; for old files, the record type is established at the file creation.
Carriage control	LIST, CARRIAGE, FORTRAN, NOCARRIAGE, or NONE	LIST for all text files; NOCARRIAGE for all other files. Old files use their existing carriage control parameter.

The following differences exist between the Version 1.0 and Version 2+ OPEN syntax.

- In Version 1.0, the file name is specified as a string constant (VMS file specification) or a logical name. In Version 2+, a string expression containing a file specification can be used as the file name.

- In Version 2+, the parameter values READONLY and UNKNOWN have been added to the history parameter.

- In Version 2+, the parameter value KEYED has been added to the access method parameter.

- In Version 2+, the default record type is VARIABLE for new text files and files of type FILE OF VARYING; for all other new files, the default is FIXED. The default for old files remains the same.

- In Version 2+, the default carriage control is LIST for text files and files of type FILE OF VARYING. The default for all other file types and for old files remains the same.

- Version 2+ includes six new parameters for the OPEN procedure: organization, disposition, sharing, user action, default, and error recovery.

Note that although direct access to text files is prohibited in both Version 1.0 and Version 2+, the point at which the error occurs differs. In Version 1.0, an OPEN procedure is allowed to specify direct access for a text file; the error occurs when a FIND procedure attempts to access the file. In Version 2+, an OPEN procedure that specifies direct access to a text file causes an error to be generated.

C.1.1.5 Specifying Qualifiers in the Source Code

In Version 1.0 of VAX Pascal, you can specify compiler qualifiers within comments in the source code. VAX Pascal Version 2+ does not support this feature. It is recommended that you specify these qualifiers with the Pascal command when you compile the program. Alternatively, you can use attributes in your program to perform some of the same operations that are performed by compiler qualifiers.

In Version 1.0, the following is true:

- The CHECK qualifier (abbreviated C) generates code to perform run-time checks.

- The CROSS_REFERENCE qualifier (X) produces a cross-reference listing of identifiers.

- The DEBUG qualifier (D) generates records for the VMS Debugger.

- The LIST qualifier (L) produces a source listing file.

- The MACHINE_CODE qualifier (M) includes machine code in the source listing file.

- The STANDARD qualifier (S) prints informational messages indicating the use of VAX Pascal extensions.

- The WARNINGS qualifier (W) prints diagnostics for warning-level errors.

The following syntax indicates how to specify qualifiers:

(*${qualifier $\left\{ \begin{array}{c} + \\ - \end{array} \right\}$},...[[; comment]] *)

qualifier
A qualifier name or a 1-character abbreviation.

comment
The text of a comment, which is optional.

+
-
Enables (+) or disables (−) the qualifier.

The first character after the comment delimiter must be a dollar sign ($), which cannot be preceded by a space.

You can specify any number of qualifiers in a single comment. You can also include text in the comment after the qualifiers. The text must be separated from the list of qualifiers by a semicolon (;).

You can use qualifiers in the source code to enable and disable options during compilation. For example, to generate check code for only one procedure in a program, insert a comment that enables the CHECK qualifier before the procedure declaration. After the end of the procedure declaration, include a comment that disables the qualifier. For example:

```
(*$C+; enable CHECK for TEST1 only *)

PROCEDURE TEST1;
    .
    .
    .
END;
(*$C-; disable CHECK option *)
```

You can specify qualifiers in both the source code and the PASCAL command line. Command line qualifiers override source code qualifiers. If, for example, the source code specifies DEBUG+, but you type PASCAL/NODEBUG, the DEBUG option will not be in effect.

C.1.2 /OLD_VERSION Qualifier

The VAX Pascal standard in Version 1.0 conflicts in some respects with that of Version 2+, which is based on Level 0 of the Pascal standard. The /OLD_VERSION qualifier on the Pascal command cause the compiler to default to the VAX Pascal Version 1.0 standard when conflicts arise. By default, /OLD_VERSION is disabled so that the compilation conforms to the Pascal standard.

Because the Version 2+ compiler performs many optimizations on the source code, you should also enable the /NOOPTIMIZE qualifier during the recompilation of old programs. The /NOOPTIMIZE qualifier prevents the compiler from making optimizations that might cause an old program to behave unexpectedly when it is recompiled.

The following sections describe the conflicts between Version 1.0 and Version 2+ and explain how they are resolved by the /OLD_VERSION qualifier.

C.1.2.1 Comment Delimiters

Unlike Version 1.0, Version 2+ considers the opening comment delimiters, (* and {, equivalent; likewise, the closing delimiters, *) and }, are considered equivalent. Therefore, a comment begun with (* can be terminated with }, and a comment begun with { can be terminated by *).

Recompilation of the program with the /OLD_VERSION qualifier will cause the Version 1.0 restriction to be enforced so that you cannot combine comment delimiters in this way.

C.1.2.2 %INCLUDE Files

In Version 1.0 of VAX Pascal, the default file type of a %INCLUDE file is .DAT. However, in Version 2+, the default file type is .PAS.

You must use the /OLD_VERSION qualifier to recompile a program that includes one or more files that have a file type of .DAT.

C.1.2.3 Multidimensional Packed Arrays

Version 1.0 of the VAX Pascal compiler interprets the shorthand form of the array type definition to be equivalent to the longer definition, as follows:

```
PACKED ARRAY[x,y,z]
ARRAY[x] OF ARRAY[y] OF PACKED ARRAY[z]
```

That is, only the last dimension of the array is packed. In Version 2+, however, the shorthand definition above is equivalent to this longer definition:

```
PACKED ARRAY[x] OF PACKED ARRAY[y] OF PACKED ARRAY[z]
```

In the Version 2+ interpretation, all dimensions of the array are packed.

You must use the /OLD_VERSION qualifier to recompile a program that includes a multidimensional packed array of which you want only the last dimension to be packed.

C.1.2.4   Storage of Components

In Version 1.0 of VAX Pascal, a component of the subrange type 0..0 in a packed record or array is allocated one bit of storage. In Version 2+, however, a component of this type is not allocated any storage.

You must use the /OLD_VERSION qualifier to recompile a program in which one bit of storage is required to be allocated for such a component.

C.1.2.5   Storage of Sets

In Version 1.0 of VAX Pascal, an unpacked set type is allocated 256 bits. In Version 2+, the allocation size of an unpacked set depends on the set's base type and on whether the unpacked set is allocated in a packed or an unpacked context.

You must use the /OLD_VERSION qualifier to recompile a program in which an unpacked set requires an allocation size of 256 bits.

C.1.2.6   TEXT Files and FILE OF CHAR

Version 1.0 of VAX Pascal considers the predefined file types TEXT and FILE OF CHAR to be equivalent. In Version 2+, however, files of type TEXT are composed of complete lines of characters terminated by an end-of-line marker, while files of type FILE OF CHAR are composed of individual characters.

You must use the /OLD_VERSION qualifier to recompile a program that requires a TEXT file and a FILE OF CHAR to be treated identically.

C.1.2.7 MOD Operator

The MOD operator, as defined by Version 1.0 of VAX Pascal, returns the remainder that results from the DIV operation. In Version 2+, the MOD operator is equivalent to the mathematical modulus operation. Therefore, Version 2+ allows you to perform the operation I MOD J only when J is a positive number; the MOD function always returns a value from 0 to J − 1. To compute the remainder from the DIV operation, Version 2+ provides the REM operator.

You must use the /OLD_VERSION qualifier to recompile a program in which you use the MOD operator to compute the remainder.

C.1.2.8 String Variable Parameters to the READ Procedure

In Version 1.0 of VAX Pascal, if a READ procedure encounters an end-of-line marker as the first character to be read into a string variable, it ignores the marker and advances the file position to the beginning of the next line of input. In Version 2+, a READ procedure never skips an end-of-line marker that is the first character to be read into a string variable. If a READ procedure encounters an initial end-of-line, the file remains positioned at the end of the line; you must call a READLN procedure to advance the file position to the next line.

You must recompile with the /OLD_VERSION qualifier to cause a READ procedure to skip one end-of-line marker that it encounters as the first character to be read into a string variable.

C.1.2.9 Field Widths

In Version 1.0 of VAX Pascal, a value of type REAL or SINGLE is written with a default field width of 16 characters; a value of type DOUBLE, with 24. In Version 2+, the default field width for a value of type REAL or SINGLE is 12 characters; for a value of type DOUBLE, 20.

In addition, Version 1.0 of VAX Pascal always expands the field width of a real number written in decimal format (when necessary) so that the real number is preceded by a leading blank. No leading blank is inserted in Version 2+.

You must use the /OLD_VERSION qualifier to recompile a program in which you want to use the default field width specifications of Version 1.0.

C.1.2.10 Global Identifiers

In Version 1.0 of VAX Pascal, the names of program-level procedures and functions are considered global identifiers. However, in Version 2+, such names are not considered global unless they have the GLOBAL attribute.

You must use the /OLD_VERSION qualifier to recompile a program in which the names of program-level routines are to be made global by default.

C.1.2.11 Allocation in Program Sections

Unlike Version 1.0, Version 2+ of VAX Pascal does not allocate storage for static, program-level variables in an overlaid program section.

To cause the Version 2+ compiler to treat static, program-level variables and routine identifiers in the same manner as in Version 1.0, you must use the /OLD_VERSION qualifier. You can also apply the OVERLAID attribute to a compilation unit to cause the storage for its static, program-level variables to be allocated in an overlaid program section. Enabling /OLD_VERSION has the same effect as applying the OVERLAID attribute to all compilation units in a program.

C.1.3 Minor Language Changes

Some minor language changes that were made in Version 1.0 cannot be controlled by the /OLD_VERSION qualifier. Such changes, however, are not likely to have adverse affects on most existing VAX Pascal programs. These changes are as follows:

- To flag language extensions when the /STANDARD qualifier is enabled, Version 2+ uses the Pascal standard proposed by the International Organization for Standardization (ISO). The Version 1.0 language is defined by the *Pascal User Manual and Report* by Jensen and Wirth.

- In Version 2+, the /STANDARD qualifier is disabled by default. The Version 2+ compiler does not automatically flag extensions to the Pascal language definition contained in the ISO standard. /STANDARD is enabled in Version 1.0.

- Version 2+ ignores all comments whose first character (inside the opening delimiter) is a dollar sign ($). Note that this behavior prohibits the specification of compile-time qualifiers in the source code, which is legal in Version 1.0 (see Section C.1.1.5).

- In Version 2+, the /CHECK qualifier is enabled by default to check the bounds of array and character-string assignments. You can change the default if you wish, and you can also specify the checking of other aspects of your program. /CHECK is disabled by default in Version 1.0 and does not allow you to specify checking options.

- In Version 2+, a change in the value of the control variable inside the body of a FOR statement does not affect the number of times the loop body is executed. (This behavior is the reverse of Version 1.0.)

- Version 2+ considers the value of EOLN to be FALSE when the value of EOF is TRUE. (In Version 1.0, EOLN returns the value of TRUE at the end-of-file.) This change is necessary to make VAX Pascal conform to the ISO standard, which forbids a program from testing for EOLN at the end-of-file.

- A negative field-width value in a WRITE or WRITELN procedure call generates an error in Version 2+. In Version 1.0, a negative field-width value defaults to 0.

- When writing double-precision values, Version 2+ output procedures indicate the exponent by the letter E. (Version 1.0 uses the letter D on output values.) The input procedures in both Version 1.0 and Version 2+ accept either D or E as the exponent letter of double-precision values.

- Under VMS Version 4.6 or higher and Version 2+, the default length of a record in a text file is 255 bytes. Under a VMS version prior to VMS 4.6 and Version 2+, the default length of a record is 133 bytes. Because of an error in Version 1.0, the default length is actually 254, contrary to the description in the documentation.

- Sets in Version 2+ have allocation sizes different from those in Version 1.0.

- LIB$ESTABLISH, the VMS Run-Time Library procedure that sets up condition handlers, cannot be used in Version 2+. Instead, use the new predeclared procedures ESTABLISH and REVERT.

- Run-time condition values have new values in Version 2+. These values are contained in the file SYS$LIBRARY:PASDEF.PAS. To make them available in your program, include the file in a CONST section.

- In Version 2+, when a nonlocal GOTO statement transfers control from a routine to a labeled statement in an enclosing block, any condition handlers established by intervening routines are called first with the condition SS$_UNWIND. In Version 1.0, a nonlocal GOTO statement transfers control directly to the labeled statement; no condition handlers are executed for intervening routines.

- In Version 2+, you cannot use the predeclared functions SNGL and ORD as function parameters using the Version 1.0 syntax for function parameter declarations. You must rewrite the formal parameter declarations to use the newer syntax.

C.2 Differences Between the Current Version and Past Versions

With this version of VAX Pascal, programs that ran on previous versions (back to Version 2.0) run without errors on this version. Even though this version introduces new standard features of the language, VAX Pascal retains the extensions that were available with previous versions.

When converting old programs for use with this version, remember that VAX Pascal accepts environment files created by VAX Pascal Versions 2.0 through 3.n. However, Versions 2.0 through 3.n of the compiler do not accept environment files created by this version of the compiler.

For More Information:

- On environment files (Section 7.3)

- On compilation switches (*VAX Pascal Reference Supplement for VMS Systems*)

Summary of VAX Pascal Extensions

If you need to write portable code, you should not use the language features
that are VAX Pascal extensions. The following sections provide information
on VAX Pascal extensions:

- Extensions to the unextended Pascal standards (Section D.1)
- Extensions to the Extended Pascal standard (Section D.2)

For More Information:

For information on Pascal standards, see Section 1.1.

D.1 VAX Pascal Extensions to Unextended Pascal

Table D–1 summarizes the language features provided in VAX Pascal that
are not part of the unextended Pascal language definitions.

Table D–1: VAX Pascal Extensions to Unextended Pascal

Category	Extension
Lexical and syntactical extensions	Reserved words: MODULE, OTHERWISE, REM, VALUE, VARYING, %DESCR, %STDESCR, %IMMED, %REF, %INCLUDE, %TITLE, %SUBTITLE, and %DICTIONARY
	Exponentiation operator (**)
	REM operator

Table D–1 (Cont.): VAX Pascal Extensions to Unextended Pascal

Category	Extension
	AND_THEN and OR_ELSE operators
	Type cast operator (::) for variables and expressions
	%radix-specifier and extended-digit (#) form for binary, hexadecimal, and octal notation for integers
	Double- and quadruple-precision real numbers
	Dollar sign ($) and underscore (_) characters in identifiers
	Identifiers that can begin with any character other than a letter and whose first 31 characters must be unique
	Extended syntax for inclusion of nonprinting characters in character strings
	Compile-time constant expressions allowed anywhere a constant is allowed
	Constructors of structured types used anywhere in place of a constant of the structured type
	Attributes used with data items, routines, and compilation units
	Relaxed rules for assignment compatibility
	Structural compatibility enforced between actual and formal parameters
Predefined types	UNSIGNED, SINGLE, DOUBLE (D_floating and G_floating), QUADRUPLE, STRING, TIMESTAMP
	VARYING OF CHAR structured type and concatenation operator for all strings
Predeclared procedures	CLOSE, CREATE_DIRECTORY, DATE, DELETE, DELETE_FILE, ESTABLISH, EXTEND, FIND, FINDK, HALT, LINELIMIT, LOCATE, OPEN, READV, RENAME_FILE, RESETK, REVERT, TIME, TRUNCATE, UNLOCK, UPDATE, WRITEV, GETTIMESTAMP
Predeclared functions	Transfer functions: DBLE, INT, QUAD, SNGL, TRUNC, UINT, UROUND, UTRUNC
	Dynamic allocation function: ADDRESS, IADDRESS

Table D–1 (Cont.): VAX Pascal Extensions to Unextended Pascal

Category	Extension
	Character-string functions: BIN, DEC, HEX, INDEX, LENGTH, OCT, PAD, STATUSV, SUBSTR, UDEC, GT, GE, LT, LG, EQ, NE
	Parameter functions: ARGUMENT, ARGUMENT_LIST_LENGTH, PRESENT
	Privileged routines: MTPR, MFPR
	Arithmetic functions: UAND, UNOT, UOR, UXOR, XOR, MIN, MAX
	Allocation size functions: SIZE, NEXT, BITSIZE, BITNEXT
	Low-level interlocked functions: ADD_INTERLOCKED, CLEAR_INTERLOCKED, FIND_FIRST_BIT_CLEAR, FIND_FIRST_BIT_SET, FIND_MEMBER, FIND_NONMEMBER, SET_INTERLOCKED
	I/O functions: STATUS, UFB
	Field position functions: BIT_OFFSET, BYTE_OFFSET
	Additional functions: CARD, CLOCK, EXPO, UNDEFINED, ZERO, DATE, TIME, UPPER, LOWER
READ, READLN, WRITE, WRITELN extensions	Parameters of character-string and enumerated types for READ and READLN
	Parameters of enumerated types for WRITE and WRITELN
	Prompting at the terminal with a WRITE/READ or WRITE/READLN sequence
	Optional carriage-control specification for text files with WRITE and WRITELN
	Optional radix specification for READ and READLN
Extended I/O capabilities	Direct access and relative file organization
	Keyed access and indexed file organization
Declarations	Declaration and definition sections that can appear more than once and in any order
	Initialization of variables, types, and record fields in VAR declaration sections at program or module level

Table D–1 (Cont.): VAX Pascal Extensions to Unextended Pascal

Category	Extension
	Schema types
	VALUE initialization section
	OTHERWISE clause in variant records
	Ranges in variant label lists
Statements	OTHERWISE clause in CASE statement
	Ranges in CASE label lists
	FOR statement with SET iterations
Procedures and functions	Functions that return values of structured types (other than file types)
	Functions called as procedures
	External procedure and function declarations
	Default values for formal parameters
	Nonpositional parameter passing
	Extended mechanism specifiers and parameter passing attributes for passing parameters to external procedures and functions: %IMMED, %REF, %DESCR, %STDESCR, IMMEDIATE, REFERENCE, CLASS_S, CLASS_A, CLASS_NCA
Compilation	MODULE capability for combining declarations and definitions to be compiled independently from the main program
	ENVIRONMENT and INHERIT attributes to control independent compilation
	Module initialization and finalization

D.2 VAX Pascal Extensions to Extended Pascal

Table D–2 summarizes the language features provided in VAX Pascal that are not part of the Extended Pascal language definitions.

Table D–2: VAX Pascal Extensions to Extended Pascal

Category	Extension
Lexical and syntactical extensions	Reserved words: REM, VARYING, %DESCR, %STDESCR, %IMMED, %REF, %INCLUDE, %TITLE, %SUBTITLE, and %DICTIONARY
	REM operator
	Type cast operator (::) for variables and expressions
	"%radix-specifier number" form for binary, hexadecimal, and octal notation for integers
	Double- and quadruple-precision real numbers
	Dollar sign ($) characters in identifiers
	Identifiers that can begin with any character other than a letter and whose first 31 characters must be unique
	Extended syntax for inclusion of nonprinting characters in character strings
	Parenthetical form ((constructor)) for constructors of structured types, used anywhere in place of a constant of the structured type
	Attributes
	Relaxed rules for assignment compatibility
	Structural compatibility enforced between actual and formal parameters
Predefined types	UNSIGNED, SINGLE, DOUBLE (D_floating and G_floating), QUADRUPLE
	VARYING OF CHAR structured type and concatenation operator for all strings
Predeclared procedures	CLOSE, CREATE_DIRECTORY, DATE, DELETE, DELETE_FILE, ESTABLISH, FIND, FINDK, LINELIMIT, LOCATE, OPEN, READV, RENAME_FILE, RESETK, REVERT, TIME, TRUNCATE, UNLOCK, UPDATE, WRITEV
Predeclared functions	Transfer functions: DBLE, INT, QUAD, SNGL, TRUNC, UINT, UROUND, UTRUNC
	Dynamic allocation function: ADDRESS, IADDRESS

Table D–2 (Cont.): VAX Pascal Extensions to Extended Pascal

Category	Extension
	Character-string functions: BIN, DEC, HEX, OCT, PAD, STATUSV, UDEC
	Parameter functions: ARGUMENT, ARGUMENT_LIST_LENGTH, PRESENT
	Arithmetic functions: UAND, UNOT, UOR, UXOR, XOR, MIN, MAX
	Allocation size functions: SIZE, NEXT, BITSIZE, BITNEXT
	Low-level interlocked functions: ADD_INTERLOCKED, CLEAR_INTERLOCKED, FIND_FIRST_BIT_CLEAR, FIND_FIRST_BIT_SET, FIND_MEMBER, FIND_NONMEMBER, SET_INTERLOCKED
	Privileged routines: MTPR, MFPR
	I/O functions: STATUS, UFB
	Field position functions: BIT_OFFSET, BYTE_OFFSET
	Additional predeclared functions: CLOCK, EXPO, UNDEFINED, ZERO, UPPER, LOWER
READ, READLN, WRITE, WRITELN extensions	Parameters of enumerated types for READ and READLN
	Parameters of enumerated types for WRITE and WRITELN
	Prompting at the terminal with a WRITE/READ or WRITE/READLN sequence
	Optional carriage-control specification for text files with WRITE and WRITELN
	Optional radix specification for READ and READLN
Extended I/O capabilities	Direct access and relative file organization
	Keyed access and indexed file organization
Declarations	VALUE initialization section

Table D–2 (Cont.): VAX Pascal Extensions to Extended Pascal

Category	Extension
Procedures and functions	Functions called as procedures
	External procedure and function declarations
	Default values for formal parameters
	Nonpositional parameter passing
	Extended mechanism specifiers and parameter passing attributes for passing parameters to external procedures and functions: %IMMED, %REF, %DESCR, %STDESCR, IMMEDIATE, REFERENCE, CLASS_S, CLASS_A, CLASS_NCA
Compilation	MODULE syntax differs from the syntax provided by Extended Pascal.
	ENVIRONMENT and INHERIT attributes to control independent compilation

Appendix E

Description of Implementation Features

The standards for Pascal allow some features of the language to be defined by a particular implementation or to be dependent on an implementation. This appendix describes the VAX Pascal treatment of the following:

* Implementation-defined features (Section E.1)
* Implementation-dependent features (Section E.2)

For More Information:

For information on Pascal standards, see Section 1.1.

E.1 Implementation-Defined Features

The value of each character allowed in a character string

>   **Treatment:** See Appendix A.

The range of real number values represented by the type REAL

>   **Treatment:** See the *VAX Pascal Reference Supplement for VMS Systems.*

The characters represented by the type CHAR and their ordinal values

>   **Treatment:** See Appendix A.

The point at which the REWRITE, PUT, RESET, and GET procedures are performed on a file

>   **Treatment:** Performed immediately unless the file is a terminal file, in which case delayed device access occurs (see Section 9.5.3).

The value of MAXINT

> **Treatment:** 2,147,483,647.

The accuracy to which the results of real-number operations are calculated

> **Treatment:** See the *VAX MACRO and Instruction Set Volume* and *VMS Run-Time Library Routines Volume*.

Default field widths

> **Treatment:**

> | | |
> |---|---|
> | Values of type INTEGER | 10 |
> | Values of type REAL | 12 |
> | Values of type BOOLEAN | 6 |

The number of digits used to represent the exponent of a floating-point number

> **Treatment:** See the *VAX Pascal Reference Supplement for VMS Systems*.

The value of the exponent character

> **Treatment:** ′E′.

The case (upper or lower) in which the Boolean values TRUE and FALSE are printed as output

> **Treatment:** Uppercase; that is, TRUE and FALSE.

The effect of the PAGE procedure

> **Treatment:** PAGE writes a line containing only the form-feed character (ASCII value 12).

E.2 Implementation-Dependent Features

The order of evaluation of the following items:

- Index values of an array variable
- Expressions in a set constructor
- Operands in a dyadic operation

> **Treatment:** Random order.

Order of evaluating, and accessing of actual parameters in a function designator and a procedure call

> **Treatment:** Random order.

Order of accessing the variable and evaluating the expression in an assignment statement

> **Treatment:** Random order.

The effect of reading a text file for which the PAGE procedure was called

> **Treatment:** Reads a line containing only the form-feed character (ASCII value 12).

Error Detection

This appendix describes how the VAX Pascal compiler and run-time system detect violations of the Pascal language standards. Errors detected·at run time cause a program to terminate and return appropriate error messages. Errors described here as not detected cause a program to produce unexpected results.

For More Information:

* On standards (Section 1.1)
* On VAX Pascal error messages (*VAX Pascal Reference Supplement for VMS Systems*)

F.1 Error-Detection Information

The type of an index value is not assignment compatible with the index type of an array.

> **Explanation:** Detected at run time if bounds checking was enabled during compilation.

The current variant changes while a reference to it exists.

> **Explanation:** Not detected. An example of a reference to a variant is the passing of the variant to a formal VAR parameter.

The value of a variable to which a pointer refers (p^) is NIL.

> **Explanation:** Usually detected at run time. Always detected if pointers checking was enabled during compilation.

The value of a variable to which a pointer refers (p^) is undefined.

> **Explanation:** Not detected.

The DISPOSE procedure is called to dispose of a heap-allocated variable while a reference to the variable exists.

> **Explanation:** Not detected. Examples of such references are passing the variable, or a component of it, to a formal VAR parameter, or using the variable in a WITH statement (if the variable is a record).

The value of file f changes while a reference to f^ exists.

> **Explanation:** Not detected. An example of a reference to f^ is the passing of f^ by reference to a routine; until the routine has ceased execution, you cannot perform any operation on file f.

The ordinal type of an actual parameter is not assignment compatible with the type of the corresponding formal parameter.

> **Explanation:** Detected at run time if subrange checking was enabled during compilation of the called routine.

The set type of an actual parameter is not assignment compatible with the type of the corresponding formal parameter.

> **Explanation:** Detected at run time if subrange checking was enabled during compilation of the called routine.

A file is not in generation mode when a PUT, WRITE, WRITELN, or PAGE procedure is attempted.

> **Explanation:** Detected at run time.

A file is in undefined mode when a PUT, WRITE, WRITELN, or PAGE procedure is attempted.

> **Explanation:** Not detected.

The result of an EOF function is not TRUE when a PUT, WRITE, WRITELN, or PAGE procedure is attempted.

> **Explanation:** Detected at run time. The operation is illegal only when the file is accessed sequentially.

The value of the file buffer variable is undefined when a PUT procedure is attempted.

> **Explanation:** Not detected.

A file is in undefined mode when a RESET procedure is attempted.

Explanation: Not detected.

A file is not in inspection mode when a GET, READ, or READLN procedure is attempted.

Explanation: Detected at run time.

A file is in undefined mode when a GET, READ, or READLN procedure is attempted.

Explanation: Not detected.

The result of an EOF function is TRUE when a GET, READ, or READLN procedure is attempted.

Explanation: Detected at run time.

The type of the file buffer variable is not assignment compatible with the type of the variable that is a parameter to a READ or READLN procedure.

Explanation: Detected at run time.

The type of the expression being written by a WRITE or WRITELN procedure is not assignment compatible with the type of the file buffer variable.

Explanation: Detected at run time.

The current variant does not exist in the list of variants specified with the NEW procedure.

Explanation: Not detected.

The DISPOSE(p) procedure is called to deallocate a pointer variable that was created using the variant form of the NEW procedure.

Explanation: Not detected.

The variant form of the DISPOSE procedure does not specify the disposal of the same number of variants that were created by the variant form of the NEW procedure.

Explanation: Not detected.

The variant form of the DISPOSE procedure does not specify the disposal of the same variants that were created by the variant form of the NEW procedure.

Explanation: Not detected.

The value of the parameter to the DISPOSE procedure is NIL.

Explanation: Detected at run time.

The value of the parameter to the DISPOSE procedure is undefined.

Explanation: Not detected.

A variant record created by the NEW procedure is accessed as a whole, rather than one component at a time.

Explanation: Not detected.

In the PACK(a,i,z) procedure, the type of the index value i is not assignment compatible with the index type of a.

Explanation: Detected at run time if subrange checking was enabled during compilation.

The PACK procedure is attempted when the value of at least one component of a is undefined.

Explanation: Not detected.

The index value i in the PACK procedure is greater than the upper bound of the index type of a.

Explanation: Detected at run time if subrange checking was enabled during compilation.

In the UNPACK(z,i,a) procedure, the type of the index value i is not assignment compatible with the index type of a.

Explanation: Detected at run time if subrange checking was enabled during compilation.

The UNPACK procedure is attempted when the value of at least one component of z is undefined.

Explanation: Not detected.

The index value i in the UNPACK procedure is greater than the upper bound of the index type of a.

> **Explanation:** Detected at run time if subrange checking was enabled during compilation.

The resulting value of SQR(x) does not exist.

> **Explanation:** Detected at run time for integers if overflow checking was enabled during compilation; always detected at run time for real numbers.

In the expression LN(x), the value of x is negative.

> **Explanation:** Detected at run time.

In the expression SQRT(x), the value of x is negative.

> **Explanation:** Detected at run time.

The resulting value of TRUNC(x) does not exist after the following calculations have been done: if the value of x is positive or zero, then $0 <= x-TRUNC(x) < 1$; otherwise, $-1 < x-TRUNC(x) <= 0$.

> **Explanation:** Detected at run time if overflow checking was enabled during compilation.

The resulting value of ROUND(x) does not exist after the following calculations have been done: if the value of x is positive or zero, then ROUND(x) is equivalent to TRUNC(x + 0.5); otherwise, ROUND(x) is equivalent to TRUNC(x—0.5).

> **Explanation:** Detected at run time if overflow checking was enabled during compilation.

The resulting value of CHR(x) does not exist.

> **Explanation:** Detected at run time if subrange checking was enabled during compilation.

The resulting value of SUCC(x) does not exist.

> **Explanation:** Detected at run time if subrange checking was enabled during compilation.

The resulting value of PRED(x) does not exist.

> **Explanation:** Detected at run time if subrange checking was enabled during compilation.

The function EOF(f) is called when the file f is undefined.

> **Explanation:** Not detected.

The function EOLN(f) is called when the file f is undefined.

> **Explanation:** Not detected.

The function EOLN(f) is called when the result of EOF(f) is TRUE.

> **Explanation:** Not detected.

A variable is not initialized before it is first used.

> **Explanation:** Not detected.

In the expression x/y, the value of y is zero.

> **Explanation:** Detected at run time.

In the expression i DIV j, the value of j is zero.

> **Explanation:** Detected at run time.

In the expression i MOD j, the value of j is zero or negative.

> **Explanation:** Detected at run time if subrange checking was enabled during compilation.

An operation or function involving integers does not conform to the mathematical rules for integer arithmetic.

> **Explanation:** Detected at run time if overflow checking was enabled during compilation.

A function result is undefined when the function returns control to the calling block.

> **Explanation:** Not detected.

The ordinal type of an expression is not assignment compatible with the type of the variable or function identifier to which it is assigned.

> **Explanation:** Detected at run time if subrange checking was enabled during compilation.

The set type of an expression is not assignment compatible with the type of the variable or function identifier to which it is assigned.

> **Explanation:** Detected at run time if subrange checking was enabled during compilation.

None of the case labels is equal in value to the case selector in a CASE
statement.

Explanation: Detected at run time if case selector checking was
enabled during compilation.

In a FOR statement, the type of the initial value is not assignment
compatible with the type of the control variable, and the statement
in the loop body is executed.

Explanation: Detected at run time if subrange checking was
enabled during compilation. Assignment compatibility is not
enforced if the statement in the loop body can never be executed.

In a FOR statement, the type of the final value is not assignment compatible
with the type of the control variable and the statement in the loop
body is executed.

Explanation: Detected at run time if subrange checking was
enabled during compilation. Assignment compatibility is not
enforced if the statement in the loop body can never be executed.

When an integer is being read from a text file, the digits read do not
constitute a valid integer value. (Initial spaces and end-of-line
markers are skipped.)

Explanation: Detected at run time.

When an integer is being read from a text file, the type of the value read is
not assignment compatible with the type of the variable.

Explanation: Detected at run time if subrange checking was
enabled during compilation.

When reading a real number from a text file, the digits read do not
constitute a valid real number. (Initial spaces and end-of-line
markers are skipped.)

Explanation: Detected at run time.

The value of the file buffer variable is undefined when a READ or READLN
procedure is performed.

Explanation: Not detected.

A WRITE or WRITELN procedure specifies a field width in which the integers representing the total width and the number of fractional digits are less than 1.

Explanation: Not detected.

The bounds of an array passed to a conformant array parameter are outside the range specified by the conformant array's index type.

Explanation: Detected at run time if bounds checking was enabled during compilation.

Index

G

H

I

O

P

T

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	———	Local Digital subsidiary or approved distributor
Internal[1]	———	USASSB Order Processing - WMO/E15 or U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

[1] For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

What I like best about this manual is _____

_____

What I like least about this manual is _____

_____

I found the following errors in this manual:
Page   Description

_____   _____

_____   _____

_____   _____

Additional comments or suggestions to improve this manual:

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____
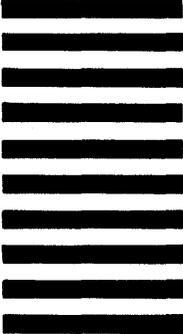
Company _____ Date _____

Mailing Address _____

_____ Phone _____