HEWLETT PACKARD

HP 92832A Pascal/1000

Reference Manual

Pascal/1000

Reference Manual

HEWLETT-PACKARD COMPANY
Data Systems Division
11000 Wolfe Road
Cupertino, California 95014

Library Index No.
2RTE.320.92832-90001

MANUAL PART NO. 92832-90001
Printed in U.S.A. May 1980

PRINTING HISTORY

The Printing History below identifies the Edition of this Manual and any Updates that are included. Periodically, Update packages are distributed which contain the latest replacement pages and write-in instructions to be merged into the manual, including an updated copy of this Printing History page.

To replenish stock, this manual will be reprinted as necessary. Each such reprinting will incorporate all past Updates, however, no new information will be added. Thus, the reprinted copy will be identical in content to prior printings of the same edition with its user-inserted update information.

To determine the specific manual edition and update which is compatible with your current software revision code, refer to the appropriate Software Numbering Catalog.

First Edition ..... May 1980

Preface

When Niklaus Wirth invented a new computer language in 1968, he named it in honor of Blaise Pascal, one of the great philosophers of the 17th century. In addition to inventing one of the first calculating machines, Blaise Pascal made important contributions to mathematics, physics, and linguistics.

Wirth designed Pascal primarily as a vehicle for teaching programming as a discipline based on the fundamental concepts of structure and logical integrity. However, the range of basic capabilities built into Pascal made it suitable for wider use. Pascal has been successfully implemented on a wide variety of computers, from microprocessors to mainframes, and has been proven to be both reliable and efficient for a variety of programming tasks.

Pascal/1000 provides all of the capabilities of the language as originally designed by Wirth, as well as several significant extensions which take advantage of HP 1000 system capabilities.

Information contained in this manual closely conforms to the proposed "standard" Pascal under consideration by the American National Standards Institute (ANSI) and the British Standards Institute (BSI). The HP extensions added to the "standard" Pascal are noted as such to facilitate program transportability.

TABLE OF CONTENTS

Chapter 1
General Information

Chapter 2
General Form

Chapter 3
Compilation Units

Chapter 4
Declarations

Chapter 5
Executable Parts

Chapter 6
Files

Chapter 7
Standard Procedures and Functions

Chapter 8
Implementation Considerations

Chapter 9
How To Use Pascal/1000

Appendix A
Syntax Diagrams

Appendix B
Run-Time Errors and Warnings

Appendix C
Compile-Time Errors

Appendix D
Compiler Options

Appendix E
Program to Program Communication

Appendix F
User-Callable Pascal/1000 Library Routines

Appendix G
Pascal/1000 Cross-Referencer

List of Illustrations

List of Tables

Chapter 1
General Information

Introduction

Pascal/1000 is a high-level, block structured programming language. It can be used for program development on HP 1000 computer systems. It is implemented with the HP Pascal/1000 compiler, which translates Pascal source code to RTE Assembly code, which is then automatically assembled to produce the object code. Pascal/1000 object programs can be executed on HP 1000 computer systems operating under the HP Real Time Executive operating system (e.g.,RTE-L and RTE-IVB). Pascal/1000 fully implements the "standard" language defined by Niklaus Wirth as well as important HP extensions which take full advantage of the HP 1000 computer system capabilities.

Pascal/1000 combines good control structures and powerful data structuring with simplicity of use. It is easy to learn and programs written in it are easy to read and debug. A great deal of compile-time and run-time checking can be done to promote program correctness. Its logical, block-structured organization facilitates documentation, modification and maintenance of programs. Pascal/1000 is compatible with HP 1000 software subsystems, such as IMAGE/1000, GRAPHICS/1000, and DS/1000.

Extensions to Standard Pascal

Pascal/1000 is a superset of "standard" Pascal as defined by Jensen and Wirth in the Pascal User Manual and Report (second edition), published by Springer and Verlag, 1976.

To avoid unintentional inclusion of Pascal/1000 extensions in a program that must be transportable to a system that executes only "standard" Pascal, the Pascal/1000 compiler includes an option for flagging any feature in a Pascal/1000 program that is not part of "standard" Pascal.

Pascal/1000 extensions include:

1. Double-precision floating point data types.

2. Additional predefined I/O procedures.

3. Separate routine compilation with load-time linking of modules (In "standard" Pascal, the entire program must be compiled at the same time). Segmentation can be accomplished in a simple and straightforward manner.

4. The CASE statement can have subrange labels and an OTHERWISE clause.

5. Constant-valued expressions are allowed in most places that constants are allowed in "standard" Pascal.

6. Structured constants allow arrays, records, and sets to be easily initialized.

7. Declarations may be in any order, except that LABEL must be first if it is used at all. More than one of each declaration section is allowed, except LABEL (such as two or more sections of TYPE, CONST, and/or VAR declarations).

8. Identifiers may be up to a source line in length with all characters significant (as opposed to only the first eight in "standard" Pascal).

9. A function may return any type of data (including arrays, records, or sets), except files or data types containing files.

10. MARK and RELEASE procedures supplement the "standard" Pascal facilities for dynamic memory management.

11. A program may call external EXEC, FMP, Pascal, FORTRAN, or Assembly language routines.

12. The heap and stack can reside in logical address space, or in the Extended Memory Area (EMA) for Pascal programs running under RTE-IVB.

Summary of the Pascal/1000 Program

A Pascal/1000 program may be viewed as a description of actions to be performed on data in a precise, unambiguous way.

The data is represented by constants and variables. Each constant or variable in the program must be defined in a CONSTANT DEFINITION SECTION or declared in a VARIABLE DECLARATION SECTION before it is used. The CONSTANT DEFINITION SECTION associates values with identifiers. The VARIABLE DECLARATION SECTION associates data types with identifiers. A DATA TYPE may be predefined or user-defined. Each user-defined data type must be defined in a TYPE DEFINITION SECTION where the set of values which may be assumed by a variable of that type is described before it is used.

A data type may be SIMPLE (variables of that type have only one component), or STRUCTURED (variables of that type have multiple components).

Five simple types are predefined by the compiler and are known as STANDARD TYPES. They are: REAL and LONGREAL (subsets of the arithmetic set of real numbers), INTEGER (a subset of the arithmetic set of integers), CHAR (the characters defined by the eight-bit ASCII character set) and BOOLEAN (the values true and false).

Other simple types may be defined by the programmer in the type definition section. These are known as USER-DEFINED TYPES and are defined by ENUMERATION in one of two ways. One is to list all allowable values for a variable of that type. The other is to declare it to be a SUBRANGE of a previously defined ORDINAL type by specifying the limits of the subrange. The ordinal type may be BOOLEAN, INTEGER, CHAR, a user-defined enumeration type or a subrange type.

STRUCTURED TYPES are defined by describing a structuring method and the type of each component. In Pascal, there are four methods of structuring: ARRAY, RECORD, SET, and FILE STRUCTURES.

The components of an ARRAY STRUCTURE (called ELEMENTS) are all of the same type, and the number of elements must be specified when the array is defined. An array element is accessed by an ARRAY SELECTOR, which consists of the identifier associated with the array and an ARRAY INDEX of an ordinal type. Since the time needed to access an array element is independent of its position in the array, the array is a RANDOM ACCESS STRUCTURE.

The components of a RECORD STRUCTURE (called FIELDS) may be of different types. A field is accessed by a FIELD SELECTOR, which consists of the identifier associated with the record and the identifier associated with the field of the record. A record is also a random access structure.

A record may contain one or more VARIANTS. These allow a single record to have several possible structures, each differing in number and/or type of components. A record containing a variant part may also contain a TAG FIELD associated with that variant.

The components of a SET STRUCTURE are SET ELEMENTS which define a BASE TYPE. The base type must be a scalar or subrange type. A variable of a set type takes values which are subsets of the base type elements. (In the terms of mathematical set theory, a set type defines the POWER SET of the set elements; that is, the set of all possible subsets of them, including the empty set.)

A FILE STRUCTURE is a collection of components of the same type. The number of components is not specified when the file is defined. At any one time, only one component may be accessed. Before a file can be used, it must be opened; that is, made available for accessing. The file can be opened by one of several predefined procedures, and the procedure used to open the file determines how that file may be accessed.

If a file is opened for either read or write access only, it is a SEQUENTIAL FILE. Components of a sequential file are made available by progressing sequentially through the file. The size of a sequential file is altered by appending components at its end.

If a file is opened for both read and write access, it is a DIRECT ACCESS file. Components of a direct access file need not be accessed sequentially. The size of a direct access file is altered by appending components at its end.

Variables which are explicitly declared in a VARIABLE DECLARATION SECTION are known as STATIC DATA STRUCTURES. A static data structure does not change during program execution. Data structures generated by executable statements are known as DYNAMIC DATA STRUCTURES. Their structure is determined by POINTERS. Each pointer is allowed to refer to variables of only one type or to the value NIL, in which case it points to no variable. Since a pointer need not refer to the same variable throughout program execution, a dynamic data structure may change during program execution.

STATEMENTS are the executable parts of the program. They either manipulate data or affect the program's flow of control. Statements may be SIMPLE or STRUCTURED. A simple statement is one which contains no other statement. The simple statements are the EMPTY, GOTO, ASSIGNMENT, and PROCEDURE CALL STATEMENTS.

An empty statement contains no characters. It may contain SEPARATORS (the blank character, end-of-line marker, or comment).

A GOTO statement contains a LABEL and unconditionally transfers control to the statement associated with that label. A label must be declared in the LABEL DECLARATION SECTION and be associated with only one statement in its scope.

A procedure call statement contains a procedure identifier and causes program control to be transferred to that PROCEDURE. A procedure contains an optional declaration section and a COMPOUND STATEMENT (together these form a BLOCK). After the compound statement has been executed, program control returns to the statement following the procedure call statement.

An assignment statement contains a variable and an EXPRESSION. Expressions consist of variables, constants, sets, OPERATORS and FUNCTION CALLS. An operator describes a mapping from the operand type(s) to the result type. Four types of operators are defined in Pascal/1000: ARITHMETIC OPERATORS (addition, subtraction, unary negation, multiplication, division using real numbers, division using integers, and modulus), BOOLEAN OPERATORS (and, negation, and inclusive or), SET OPERATORS (union, intersection, and set difference), and RELATIONAL OPERATORS (equality, inequality, ordering, set membership, and set inclusion). Functions are similar to procedures in that control is transferred to a function, but after the function has been executed, control is returned to the function call and the value is assigned to the function identifier.

A structured statement contains one or more simple or structured statements. There are four types of structured statements: COMPOUND, CONDITIONAL, REPETITIVE and WITH STATEMENTS.

A compound statement is a group of simple or compound statements that have been combined to form a single statement. This allows many separate statements to be treated as a single statement.

The conditional statements are the IF STATEMENT and the CASE STATEMENT. The IF statement contains a BOOLEAN EXPRESSION (an expression which evaluates to a Boolean value), and one or two statements. The Boolean expression is evaluated, and if the result is true the first statement is executed. If a second statement is included, and the Boolean expression is evaluated as false, the second statement will be executed.

The CASE statement contains a CASE SELECTOR, one or more statements (each statement preceded by a CASE LABEL LIST), and an optional OTHERWISE clause. The case selector, which is an expression, is evaluated. Its value may be in at most one case label list -- none of them may overlap. If the value is in a case label list, the statement following that list is executed. If the value is in no case label list and there is an OTHERWISE clause, the statements of the OTHERWISE clause are executed. If the value is in no case label list and there is no OTHERWISE clause, an error results.

Statements may be consecutively executed several times through the use of repetitive statements. The WHILE, REPEAT, and FOR STATEMENTS are the repetitive statements.

The WHILE statement contains one Boolean expression and one statement. The Boolean expression is evaluated first. If the Boolean expression is evaluated as false, the included statement is never executed. If the Boolean expression is evaluated as true, the included statement is executed and the Boolean expression is again evaluated. This test-execute cycle is continued until the Boolean expression is evaluated as false.

The REPEAT statement contains one or more included statements and a Boolean expression. The statements are first executed and then the Boolean expression is evaluated. If the Boolean expression is evaluated as true, the REPEAT statement is exited. If the Boolean expression is evaluated as false, the included statements are again executed. This execute-test cycle continues until the Boolean expression is evaluated as true.

The FOR statement contains a statement, a CONTROL VARIABLE, and a range of values which the control variable may assume. The control variable is assigned the initial value of the range and the included statement is executed repetitively while the control variable remains within that range. After each execution of the included statement, the control variable is incremented or decremented.

The WITH statement is used with records and contains one or more record identifiers, and a statement. Within the statement, fields of the specified records can be accessed by their field selectors alone.

Reference Manual Organization

The remaining chapters of this manual contain the reference material for Pascal/1000. Chapters 2 through 7 are organized according to the Pascal/1000 constructs. Figure 1-1 shows the topics covered in these chapters and the major headings in each chapter.

The general form of the Pascal/1000 program acceptable to the compiler is described in Chapter 2. The compilation units in Pascal/1000 are discussed in Chapter 3. Declarations and definitions of the objects to be used by a program or routine are described in Chapter 4. The executable parts of a program, routine, or subprogram are described in Chapter 5. Program interfacing with outside objects is accomplished via input/output files. These files are described in Chapter 6. Routines that are predefined to perform common tasks are called procedures. If a routine requires a return value, it is called a function. Standard procedures and functions are described in Chapter 7.

Chapter 8 presents a detailed discussion of system implementation considerations.

Chapter 9 provides program development information. Included are procedures for program creation, compilation, and execution. Also provided are error messages, program debugging, and selected applications.

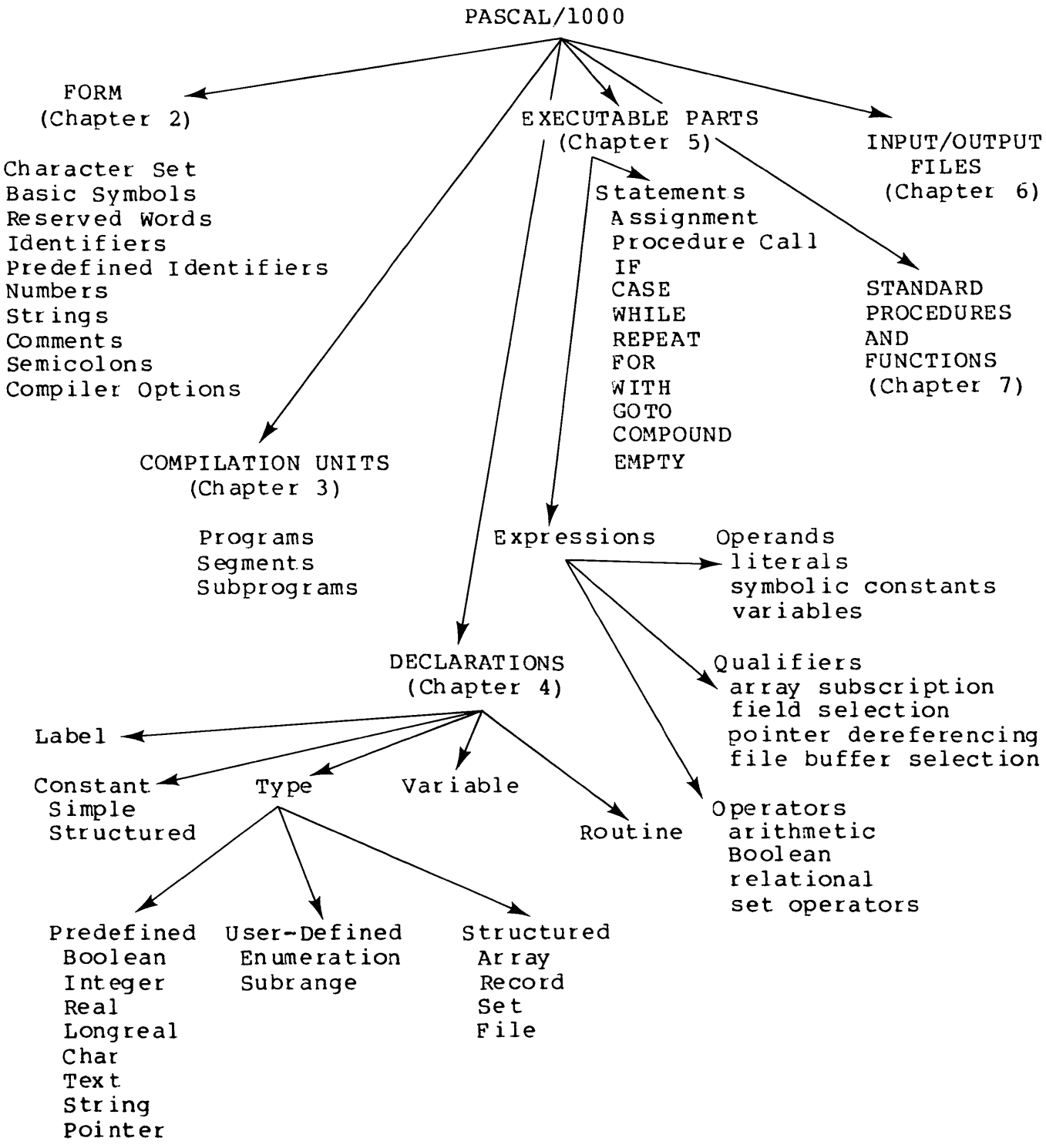


Figure 1-1. Pascal/1000 Language Constructs.

Program Vocabulary

A tabulation of the Pascal/1000 program vocabulary is given in Table 1-1. In the table, all reserved words, predefined data types, and key words are given in capital letters, although that would not be necessary in an actual program. The vocabulary item being described is in bolder type. Vocabulary items that are extensions to "Standard" Pascal are flagged with a ♦.

Table 1-1. Pascal/1000 Program Vocabulary Items

PROGRAM VOCABULARY ITEMS	USES
PROGRAM, PROCEDURE, AND FUNCTION STATEMENTS	
PROGRAM name (input, output, otherfile1, otherfile2);	Identifies program by name and specifies the logical names of the standard files and/or any user-defined files it uses.
PROCEDURE name (VAR num, den: INTEGER);	Identifies procedure by name, which may be followed by a list of parameters.
FUNCTION name (value: REAL): REAL;	Identifies function by name, along with parameter(s) used, followed by the type of function.
DECLARATIONS	
LABEL 2	Declares a label to be used in the program as the destination point of a GOTO.

CONST
pi = 3.1415926;
multiplier = 0.8598;
i = −10;
j = 20;
k = i+j;
	Defines the names and values of constants used.

TYPE SUB1 = 1..5; SUB2 = 10..20; MATRIX = ARRAY[SUB1, SUB2] OF REAL;	Defines data types, such as arrays, records, sets, files, scalars, and subranges.
VAR rpm : INTEGER; oiltemp, watertemp : REAL; charging : BOOLEAN; P : PACKED ARRAY [i . . j] OF SUB1;	Declares the names and types of variables used in the program.
GENERAL PROGRAM STATEMENTS	
BEGIN * * **END**	Delimits a series of program statements, such as the executable part of a program, procedure, function, or the statements following FOR-DO, WHILE-DO, or IF-THEN-ELSE statements.
REPEAT * * **UNTIL** m > n;	Delimits a series of program statements that are executed repeatedly until the condition specified in the UNTIL statement becomes true.

FOR n := 1 **TO** m **DO**
or
FOR n := 50 **DOWNTO** m **DO**
	Sets up repeated execution of one or more statements until the specified condition (n = m in this example) is satisfied.

WHILE m < n **DO**	Sets up repeated execution of one or more statements while the specified condition (m <n in this example) continues to be true.
IF condition **THEN** action **ELSE** other action	Sets up execution of alternate actions contingent upon result of one or a series of IF condition tests.
CASE expression **OF** list of values: action; list of values: action; * * list of values: action; ♦ **OTHERWISE** default action; **END;**	Sets up execution of one of several actions according to the value of a scalar or subrange expression.
variable **:=** expression	Assigns the value of the expression to the variable.
GOTO i	Causes a direct transfer to the statement labelled with the integer constant i.
WITH recordident **DO**	Identifies a series of records whose fields can be accessed without having to re-specify the record variable names.
{comment} or (* comment *)	Braces or (* and *) provide delimiters for comments.

♦ *Identifies extension to Wirth Pascal.*

Table 1-1. Pascal/1000 Program Vocabulary Items (Continued)

PROGRAM VOCABULARY ITEMS	USES
ARITHMETIC OPERATORS	
NOTE: In general, if both operands are of type integer, type real, or type longreal, the result is of that type. If one of the operands is real or longreal, the result is of that type.	
result := value1 **+** value2	Addition of real, longreal, or integer values.
result := value1 **−** value2	Subtraction of real, longreal, or integer values.
result := multiplicand ***** multiplier	Multiplication of real, longreal, or integer values.
result := dividend/divisor	Division of real, longreal, or integer values. The result is always real or longreal.
result := dividend **DIV** divisor	Division of integer values with integer result in which any fractional part is truncated.
result := number **MOD** divisor	Gives remainder of an integer number after division by an integer divisor.
BOOLEAN OPERATORS	
NOT empty	Negation of Boolean operand.
finished **AND** empty	Logical AND.
coffee **OR** tea	Logical OR.
SET OPERATORS	
[....]	Brackets function as set delimiters.
[apples] **+** [sugar] = [apples, sugar]	Set union operator (+) and set equality (=).
[paper,pen] **−** [pen] = [paper]	Set difference operator (−) and set equality (=).
[h2,0] ***** [h2,cl] = [h2]	Set intersection operator (*) and set equality (=).
[apples] **<=** [apples, apricots, oranges, peaches]	First set is contained in (<=) the second set.
[apples, appricots, peaches] **>=** [apricots]	First set contains (>=) the second set.
pen **IN** [pen,paper,ink]	Tests whether an element is included in a set.
[icecream] **<>** [ice,cream]	First set is not equal to the second set.
[ ]	Empty set.
RELATIONAL OPERATORS (produce Boolean results)	
dividend < 0	Less than (<).
result **<=** mininumber	Less than or equal to (<=).
number **=** result − 0.456	Equal to (=).
press1 **<>** press2	Unequal to (< >).
oiltemp **>=** 220	Greater than or equal to (>=).
watertemp > 220	Greater than (>).
STANDARD ARITHMETIC FUNCTIONS	
abs (x)	Computes absolute value of real, longreal, or integer x with result of same type as x.
sqr (x)	Computes square of real, longreal, or integer x with result of the same type as x.
sin (x)	Computes sine of x radians for real, longreal, or integer x with real or longreal result.
cos (x)	Computes cosine of x radians for real, longreal, or integer x with real or longreal result.
exp (x)	Computes base e exponential value of real, longreal, or integer x with real or longreal result.
ln (x)	Computes base e logarithm of real, longreal, or integer x with real or longreal result.
sqrt (x)	Computes square root of real, longreal, or integer x with real or longreal result.
arctan (x)	Computes arc tangent of real, longreal, or integer x with real or longreal result in radians.
STANDARD PREDICATE FUNCTIONS	
odd (x)	Tests integer x with result true if x is odd and false otherwise.
eof (f)	Indicates whether file f is at the end of a file.
eoln (f)	Indicates whether text file f is at the end of a line.
STANDARD TRANSFER FUNCTIONS	
trunc (x)	Converts real or longreal x to an integer result which is the integral part of x (deletes fractional part of x).
round (x)	Converts real or longreal x to an integer result that is the value of x rounded to the nearest integer.

Table 1-1. Pascal/1000 Program Vocabulary Items (Continued)

PROGRAM VOCABULARY ITEMS	USES
STANDARD ORDINAL FUNCTIONS	
ord (x)	Returns an integer result that is the ordinal number of x in its defined list of values.
chr (x)	Returns the character value whose ordinal number is equal to the value of the integer expression x, if there is such a value.
succ (x)	Returns a value whose ordinal number is one greater than that of the ordinal value of expression x, if there is such a value.
pred (x)	Returns a value whose ordinal number is one less than that of the ordinal value of expression x, if there is such a value.
◆ **linepos** (f)	Returns an integer number of characters read from, or written to, textfile f, since the last eoln.
◆ **position** (f)	Returns an integer representing the current position of the file buffer in direct access file f, starting from 1. This is the index of the next component which will be read or written by a call to read or write.
◆ **maxpos** (f)	Returns an integer representing the position of the last element of direct access file f which may ever be accessed.
INPUT AND OUTPUT PROCEDURES	
read (fn,p1,p2,...pn)	Reads input data or text from a named file (fn).
readln (fn,p1,p2,...pn)	Similar to read, but for text files only with skipping to the start of the next line after the last-specified parameter has been read in the current line.
◆ **readdir** (f,k,v1,..., vn)	Reads values v1 through vn from direct access file f, starting at component k.
write (fn,p1,p2,...pn)	Writes output data or text to a named file.
writeln (fn,p1,p2,...pn)	Similar to write, but for text files only with addition of an end-of-line marker after the last parameter.
◆ **writedir** (f,k,v1, ...,vn)	Writes values v1 through vn to direct access file f, starting at component k.
FILE HANDLING PROCEDURES	
rewrite (f)	Opens file f for writing, with index positioned to the first component; any previously existing information in the file is discarded.
◆ **append** (f)	Opens file f for writing, with index positioned just beyond the last-written component for addition of components to f.
reset (f)	Opens file f for reading, with index positioned at the first component.
put (f)	Writes the value of buffer variable f↑ to file f and advances to the next component.
get (f)	Advances, then assigns the value of the current file f component to buffer variable f↑ if the component exists, and advances to the next component; if the component does not exist, the eof condition is set for f.
◆ **close** (f)	Closes file f. A second parameter can be used to cause the closed file to be either saved or purged.
◆ **open** (f)	Opens non-text file for direct access, positioned at the first component.
◆ **seek** (f,k)	Positions direct access file f at component k. If k > component bound of f, eof is set.
DYNAMIC ALLOCATION PROCEDURES	
new (p)	Allocates new variable v and assigns a pointer to v to the pointer variable p.
new (p,t1,...,tn)	Allocates a variable of record type with tag fields t1,...,tn.
dispose (p)	Releases storage such that it is available for re-use by a subsequent call to new.
dispose (p,t1,...,tn)	Releases storage for variable previously allocated using the new (p,t1,...,tn) procedure.
◆ **mark** (p)	Marks the state of the heap in the variable p, which may be of any pointer type.
◆ **release** (p)	Restores the state of the heap to the value in the variable p. This has the effect of disposing of all heap objects created by the new procedure since p was marked.

◆ *Identifies extension to Wirth Pascal.*

Table 1-1. Pascal/1000 Program Vocabulary Items (Continued)

PROGRAM VOCABULARY ITEMS	USES
DATA TRANSFER AND MISCELLANEOUS PROCEDURES	
pack (a,i,z)	Packs array a into array z, using index factor i.
unpack (z,a,i)	Unpacks array z into array a, using index factor i.
◗ **halt** (i)	Causes abnormal termination of program with display of integer i,
page (f)	Causes skipping to the top of a new page when text file f is printed.
◗ **prompt** (fn,p1,p2,...,pn)	Similar to writeln, but no line marker is written.
◗ **overprint** (fn,p1,p2,...,pn)	Similar to writeln, but the next line prints over the current line.

◗ *Identifies extension to Wirth Pascal.*

Chapter 2
General Form

Introduction

The general form of source code acceptable to the Pascal/1000 compiler is described in this chapter. The compiler accepts as input a sequence of lines from one or more source code files. These lines are processed as a stream of characters organized into the following groups:

basic symbols
reserved words
identifiers
numbers
strings
comments
compiler options
separators

Basic Symbols

The basic symbols consist of letters, digits, and special symbols.

The letters include both upper and lower case letters A through Z. The Pascal/1000 compiler does not distinguish between upper and lower case letters (with the exception of characters within a string).

The digits are 0 through 9.

The special symbols are characters (or character groups) that have special meaning in Pascal/1000. These symbols are shown in Table 2-1 which gives their syntactic significance.

Reserved Words

Reserved words are symbols that have special meaning in Pascal/1000. They are indivisible and cannot be used as identifiers. They may however, be used within comments. A list of reserved words with brief descriptions is given in Table 2.2.

Table 2-1. Special Symbols

SYMBOL	DESCRIPTION	
+	Add.	Arithmetic operator, set union.
−	Subtract/Negate.	Arithmetic operator, set difference.
*	Multiply.	Arithmetic operator, set intersection.
/	Real divide.	Arithmetic operator.
=	Equality.	Boolean Operator.
<	Less than.	Boolean operator.
<=	Less than or equal to. Subset of.	Boolean operator.
<>	Not equal.	Boolean operator.
>=	Greater than or equal to. Superset of.	Boolean operator.
>	Greater than.	Boolean operator.
()	Indicates an expression group or a parameter list.	
[]	Set, structured constant, and array index delimiters.	
,	Argument, structured constant, and enumeration separator.	
;	Statement separator, parameter separator.	
.	Field selector. Decimal point. End of program, subprogram, or segment delimiter.	
:	Case or statement label delimiter. Field width delimiter. Identifier list delimiter.	
↑	Indicates pointer dereferencing or file buffer accessing.	
:=	Assignment.	
..	Subrange.	
'	String delimiter.	
#	Indicates non-printing character in string constant.	
$	Compiler option delimiter.	
_	Allowed in identifiers (but not as first character).	
{}	Comment delimiters.	
(* *)	Comment delimiters.	

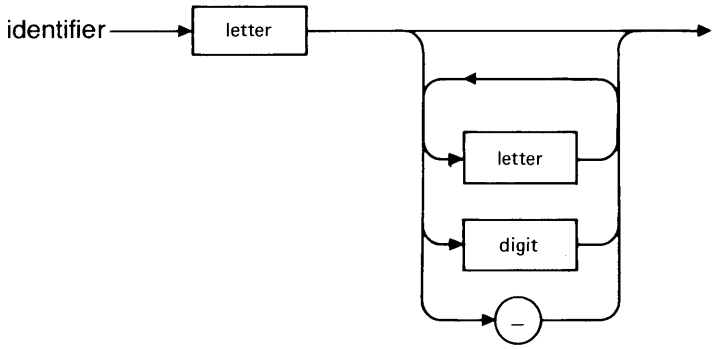
Table 2-2. Reserved Words

WORD(s)	DESCRIPTION
AND	Boolean conjunction operator.
ARRAY	A structured type.
BEGIN,END	Delimit a compound statement.
CASE,OF,OTHERWISE	A conditional statement.
CONST	Indicates constant definition section.
DIV	Integer division operator.
FILE	A structured type.
FOR,TO,DOWNTO,DO	A repetitive statement.
FUNCTION	Indicates a function declaration.
GOTO	Control transfer statement.
IF,THEN,ELSE	A conditional statement.
IN	Set inclusion operator.
LABEL	Indicates label definition section.
MOD	Integer modulus operator.
NIL	Special pointer value.
NOT	Boolean negation operator.
OR	Boolean disjunction operator.
PACKED	Controls storage allocation for structured types.
PROCEDURE	Indicates a procedure declaration.
PROGRAM	Program heading.
RECORD	A structured type.
REPEAT,UNTIL	A repetitive statement.
SET	A structured type.
TYPE	Indicates a type definition section.
VAR	Indicates a variable declaration section.
WHILE,DO	A repetitive statement.
WITH,DO	Opens record scope(s).

Identifiers

Identifiers are used to denote constants, types, variables, procedures, functions, and programs. They consist of a sequence of characters which can be upper or lower case letters, digits, or the underscore character (_). The first character must be a letter.

Syntax:



An identifier may be up to a source line in length, with up to 150 significant characters. The number of significant characters in identifiers can be changed to any value between 1 and 150 with the IDSIZE compiler option. The default is 150 significant characters.

Since upper and lower case letters are not distinguished within identifiers, the following identifiers all refer to the same object.

```
        Ident
        IDENT
and     ident
```

A reserved word cannot be used as an identifier; however, the sequence of characters which make up a reserved word can be used within an identifier. For example,

```
modern
arrayptr
divisor
```

are all valid identifier names.

Each identifier must be unique within its scope (i.e., within the procedure, function, record, or program in which they are declared).

All identifiers must be defined before they are used, except that a pointer type identifier may refer to a type which is defined later in the same declaration section and a program parameter may refer to a file variable which is declared in the program declaration part.

Further information on identifier definition and scope can be found in Chapter 4.

Examples

The following are legal identifiers:

```
total
voltage
counter
ok_
final_score
try496
a_1_and_a_2
```

The following are not legal identifiers:

```
1_or_2          {begins with a number       }
test case       {contains a space           }
part#           {contains an illegal symbol}
array           {is a reserved word         }
_first_word     {begins with an underscore  }
```

Predefined Identifiers

The following identifiers are predefined in Pascal/1000. This does not prevent the user from redefining them.

Predefined Symbolic Constants (refer to EXPRESSIONS in Chapter 5).

Symbol	Type
FALSE	BOOLEAN
TRUE	BOOLEAN
MAXINT	INTEGER
MININT	INTEGER

Predefined Types (refer to TYPE DEFINITIONS in Chapter 4).

Symbol	Type
INTEGER	MININT..MAXINT
REAL	Subset of the real numbers.
LONGREAL	Subset of the real numbers with extended precision.
BOOLEAN	(FALSE,TRUE).
CHAR	The 8-bit ASCII character set.
TEXT	FILE OF CHAR (with additional attributes).

Predeclared Variables (refer to TEXT FILES in Chapter 6).

Symbol	Type
INPUT	TEXT
OUTPUT	TEXT

Predefined Procedures and Functions (refer to Chapter 7).

abs	eoln	odd	pred	reset	succ
arctan	exp	open	prompt	rewrite	trunc
append	get	ord	put	round	unpack
chr	halt	overprint	read	seek	write
close	ln	pack	readdir	sin	writedir
cos	mark	page	readln	sqr	writeln
dispose	maxpos	position	release	sqrt	
eof	new				

Directives

The following predefined identifiers are referred to as directives.

EXTERNAL
FORWARD

Directives indicate to the compiler where the body of a procedure or function is to be found. The directive EXTERNAL is used when the body is external to the program (separately compiled or assembled). FORWARD indicates that the body is in the current compilation unit but not immediately following the procedure heading. (See Chapter 4.)

Numbers

The usual decimal notation is used for numbers, which are constants of the data types INTEGER, REAL, and LONGREAL.

Further information on numeric constants can be found in Chapter 5.

Strings

Sequences of characters enclosed by single quote marks are called strings. A string consisting of a single character is a constant of the standard type CHAR.

Both printing and non-printing ASCII characters may appear in string and character constants.

Further information on string and character constants can be found in Chapter 5.

Comments

Comments are sequences of characters used to document a program. The comments are ignored by the compiler and may appear anywhere in a program where a blank can appear.

A comment is a sequence of characters surrounded by the delimiters '{' and '}' or '(*' and '*)'. Comments do not have to be on lines by themselves and are permitted to cross line boundaries.

If the comment begins with '(*' it must be terminated using '*)'. Similarly, if it begins with '{' it must be terminated with '}'.

Comments using the same type of delimiters cannot be nested.

For example, the compiler would interpret the sequence of characters

```
{this comment {ends here} not here}
```

as the comment "this comment {ends here" and would generate syntax errors when the characters "not here}" were encountered.

The comment

```
{this comment (*contains a comment*) and ends here}
```

would be acceptable to the compiler, but the included comment must be completely contained by the outer comment. Therefore, the sequence of characters

```
{this comment (*contains a comment} but not entirely*)
```

will generate syntax errors.

Examples of comments:

```
{Increment the count by the number of elements found}

(*Program to find the averages of
  input numbers*)
```

Separators

Separators are used to separate reserved words, identifiers, numbers, strings, and special symbols. They consist of blanks, comments, compiler options and the end of a line. At least one separator must appear between any pair of such symbols (although any number are permitted) and no separator may appear within a symbol.

Compiler Options

Compiler options direct the action of the compiler in processing the source program. They may be inserted between any two identifiers, numbers, strings or special symbols. Refer to Appendix D for descriptions of the compiler options.

Below is a list of all compiler options:

ALIAS	KEEPASMB	SUBPROGRAM
ANSI	LINESIZE	SUBTITLE
ASMB	LIST	TABLES
AUTOPAGE	LIST_CODE	TITLE
BUFFERS	MIX	TRACE
CODE	PAGE	VISIBLE
DIRECT	PARTIAL_EVAL	WIDTH
EMA	PASCAL	
ERROREXIT	RANGE	
HEAP	RECURSIVE	
HEAPPARMS	SEGMENT	
IDSIZE	STATS	
INCLUDE		
IMAGE		

Chapter 3
Compilation Units

Introduction

A Pascal/1000 program may be subdivided into the following compilation units which are compiled separately:

1) main program unit

2) subprogram unit(s)

3) segment unit(s)

Every program must have a main program unit. A program may be segmented (i.e., contain segment units). The main program unit and its segment units (if any) may each be combined with subprogram units. Since subprogram and segment units are optional, many Pascal/1000 programs will contain only a main program unit. Separately-compiled units can aid program development. If errors have been found and corrected in a program unit, only that unit needs to be recompiled. The entire program is then reloaded.

The main program unit together with its subprogram units (along with any non-Pascal and library routines) constitute the main area. A segment unit together with its subprogram units (along with any non-Pascal and library routines) constitute a segment overlay. Refer to Memory Configuration in Chapter 8 for further details.

At load time, the loader combines the compilation units of a program. Commands to the loader specify which subprogram units are to be combined with the main program unit to form the main area, and which subprogram units are to be combined with each segment unit to form a segment overlay. The result is a disk-resident absolute module. Refer to Loading a Program in Chapter 9. Figure 3-1 shows the structure of a Pascal/1000 program.

Note: A distinction should be made here between the terms "routine" and "subprogram unit". A routine is either a procedure or a function that is included in a compilation unit. A subprogram unit is a compilation unit that contains routines and is combined by the loader with either the main program unit or a segment unit.

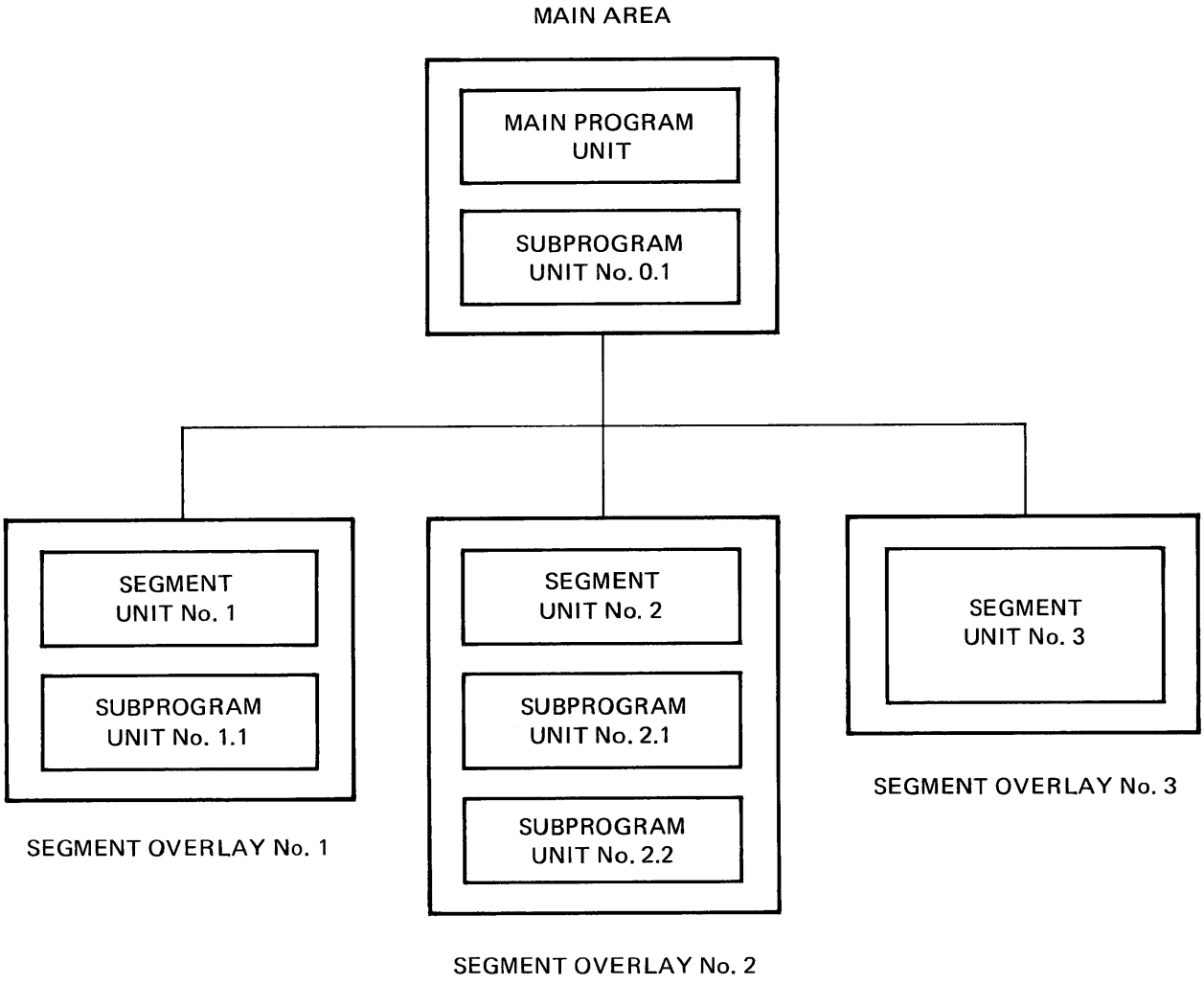


Figure 3-1: Structure of a sample Pascal/1000 program.

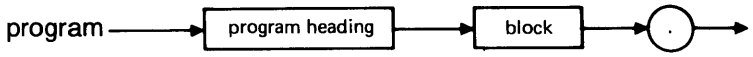
In this program the main program unit and segment unit #1 have each been combined at load time with a subprogram unit, segment unit #2 has been combined with two subprogram units, and segment unit #3 has not been combined with any subprogram units. At run time, the main area remains in memory, and each segment overlay is loaded from the disk into memory as required.

Note: Non-Pascal and library routines in the main area and each segment overlay are not shown.

Main Program Unit

A Pascal/1000 main program unit consists of a program heading and a main program block followed by a period. As mentioned above, a main program unit often constitutes the entire program.

Syntax:



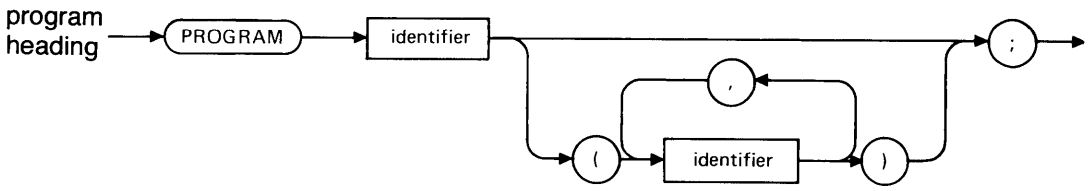
Example of a simple program:

```
PROGRAM adder (input, output);    {Program heading}

                                  {Block}
TYPE
   INT = -32768..32767;

VAR
   num1, num2, total : INT;

FUNCTION sum
   (x, y : INT) : INT;

   BEGIN
      sum := x + y;
   END;

BEGIN
   prompt ('Enter numbers: ');
   readln (num1, num2);
   total := sum (num1, num2);
   writeln ('Their sum is ',
            total);
END.                              {Period}
```

Program Heading

A program heading associates an identifier with the program and specifies a list of the program parameters.

Syntax:



The program identifier is the name by which the program is known to the operating system. It has no other significance within the program, except that it must be different in the first five characters from the name of any segment unit, or any level-1 routine.

Each program parameter is an identifier of type FILE. The files named by these identifiers are used by the program to interface to its external environment. The identifiers may name any combination of the predefined files (input and output) and user-defined files. The same identifier may not appear more than once in the parameter list. The order and number of the program parameters listed in the program heading is significant since there must be a one-to-one correspondence between the parameters in the heading and the parameters in the run string when the program is run.

The program parameter list is an exception to the rule that identifiers must be declared before being used. The identifiers of any such user-defined files must be declared in the declaration section of the main program block.

Examples of program headings:

```
PROGRAM test;

PROGRAM area (input, output);

PROGRAM lister (data, output);

PROGRAM file_merge (master_file, updates_file, new_master);
```

Main Program Block

A main program block consists of a declaration section followed by a compound statement. The "global area" (or "globals") of a program consists of the constants and types that are defined and the labels and variables that are declared in the declaration section of the main program block. The procedure and functions of the main program are declared after the global area. The declaration section is described in detail in Chapter 4.

The compound statement of the main program block is referred to as the program body. Execution begins with this statement whenever the program is run. The compound statement is described in detail in Chapter 5.

Example:

```
PROGRAM circle (data, output);          {Program heading          }
                                        {Block                    }
CONST                                      {Declaration section   }
   pi = 3.14159;                              {Global Area        }

TYPE
   COUNT = 0..10;

VAR
   data     : TEXT;
   radius   : REAL;
   number   : COUNT;

FUNCTION area                                  {Level-1 function}
   (r : REAL) : REAL;

   BEGIN
      area := pi*r*r;
   END;

BEGIN                                       {Compound statement   }
   number := 0;
   reset (data);
   REPEAT
      number := number + 1;
      read (data, radius);
      writeln ('The area of circle #',
               number:2, ' is ',
               area (radius):10:2);
   UNTIL number = 10;
END.                                    {Period                   }
```

Subprogram Unit

A subprogram unit contains a collection of level-1 routines that are compiled together. Subprogram units allow a user to group together logically-related procedures and functions. Each subprogram unit is combined with either the main program unit or a segment unit at load time by the loader (see Figure 3-1). Refer to Loading a Program in Chapter 9 for a description of how to load a subprogram.

A subprogram unit is similar to a main program unit; it consists of a program heading and a block followed by a period. The block of a subprogram unit, however, does not contain a compound statement, and a subprogram unit must include the SUBPROGRAM compiler option before the program heading.

It is important to note that a subprogram unit is not the same as a library. While the unit can be searched by the loader (refer to Loading a Program in Chapter 9), its level-1 routines cannot be selectively extracted; i.e., if any one routine in the unit is required, the entire unit will be loaded.

Subprogram Unit Program Heading

The program heading of a subprogram unit associates an identifier with the unit and contains the program parameters. Its syntax is the same as that of the program heading of the main program unit.

The subprogram identifier is the name by which the subprogram unit is known to the operating system. It has no significance within the program.

The parameter identifiers are the names of the program parameters. Since these are the same files listed in the program heading of the main program unit, the identifiers in the program headings of the subprogram unit and the main program unit must agree in name, order, number, and type. The only case where it is not necessary to list the program files in the program heading of a subprogram unit is when the unit does not refer to any of the program files or any global label, constant, or variable.

Examples:

Main program heading	Subprogram unit program heading
`PROGRAM main;`	`PROGRAM sub;`
`PROGRAM main (input, output);`	`PROGRAM sub (input, output);`
`PROGRAM main (file1, file2, output);`	`PROGRAM sub (file1, file2, output);`
`PROGRAM main (file1, file2, file3, file4);`	`PROGRAM sub; {No program files or globals referenced}`

Subprogram Unit Block

The block of a subprogram unit differs from the block of the main program unit in that it consists only of a declaration section; it contains no compound statement. The declaration section of the subprogram block consists of global definitions, declarations, and the subprogram unit's level-1 procedures and functions.

Storage for each global is allocated only once in the program, although its declaration is repeated for each compilation unit. If a compilation unit refers to any global label, constant, variable, or program parameter, then the entire global area must be reproduced exactly in that unit. No other declarations (except those within routines) are allowed in the unit. This is required to ensure the proper alignment of these global objects among the compilation units. Thus, the only case where the global area can be omitted in its entirety is when none of these global objects are used in the subprogram.

The INCLUDE compiler option is useful for reproducing globals in compilation units.

Example: The source of the global declarations is kept in a file named GLOBAL:

```
$ INCLUDE 'GLOBAL' $
```

The declarations of the subprogram unit's level-1 procedures and functions follow the global area. These routines are accessible from every compilation unit of the program. An external declaration must be made for any routine that is called but not declared in the compilation unit.

Example:

```
$SUBPROGRAM$                        {Compiler option}

PROGRAM imaginary;                  {Subprogram unit program heading}

TYPE                                {Global type definition}
   COMPLEX =
      RECORD
         re : REAL;
         im : REAL;
      END;

PROCEDURE cadd                      {Level-1 procedure}
   (x, y, z : COMPLEX);

   BEGIN
      z.re := x.re + y.re;
      z.im := x.im + y.im;
   END;

PROCEDURE csub                      {Level-1 procedure}
   (x, y, z : COMPLEX);

   BEGIN
      z.re := x.re - y.re;
      z.im := x.im - y.im;
   END;

                                    {Period}
```

Note that the final period follows the semicolon that ends the final level-1 routine, and that there is no compound statement.

Segment Unit

Segmentation allows a program to run in a partition that is smaller than the size of the program, since only part of the executable code is in memory at any time. A segment is loaded into memory only when needed for execution. When a program is run, the main area is first loaded from disk into memory where it will remain throughout the execution of the program. If a program is segmented, then at the start of each execution, all of its segment overlays initially remain on the disk. During run time the program must load each segment overlay into memory whenever that overlay is required. Only one overlay can be in memory at one time. When a segment overlay is loaded it replaces whatever overlay was already in memory.

A segment unit must begin with the SEGMENT compiler option. Otherwise, it is syntactically the same as a subprogram unit; it consists of a program heading and a block followed by a period. The block does not contain a compound statement.

Segment Unit Program Heading

A segment program heading differs from a subprogram program heading only in that the segment identifier has significance outside of the segment; the name is used by other compilation units to load the segment overlay at run time. The segment identifier must be different in the first five characters from the name of the program, any other segment unit, or any level-1 routine.

Other than the above difference, the program heading of a segment is the same as that for a subprogram.

Segment Unit Block

A segment block differs from a main program block in that it consists only of a declaration section; it contains no compound statement. It is syntactically similar to a subprogram block. Refer to Subprogram Block in this chapter.

Example:

```
$SEGMENT$                          {Compiler option}

PROGRAM Extrema;                   {Segment unit program heading}

TYPE                               {Global type definition}
   INT = -32768..32767;

   VAR                             {Global variable declaration}
      x : INT;

FUNCTION Most                      {Level-1 function}
   (y, z : INT) : INT;

   VAR
      t : INT;

   BEGIN
      t := x;
      IF y > t THEN t := y;
      IF z > t THEN t := z;
      Most := t;
   END;

FUNCTION Least                     {Level-1 function}
   (y, z : INT) : INT;

   VAR
      t : INT;

   BEGIN
      t := x;
      IF y < t THEN t := y;
      IF z < t THEN t := z;
      Least := t;
   END;

.                                  {Period}
```

Note that the final period follows the semicolon that ends the final level-1 routine, and that there is no compound statement.

Loading Segment Overlays at Run Time

A segment unit together with any subprogram units, non-Pascal routines, and library routines with which it is combined at load time constitute a segment overlay. The program must ensure that a segment overlay has already been loaded before any of its routines are invoked.

Loading a segment overlay is accomplished with a call to the Pascal/1000 library procedure @SGLD. @SGLD loads the segment overlay, then control returns to the statement following the call to @SGLD. No level-1 routine contained in the segment overlay is invoked by procedure @SGLD. Once the segment overlay has been loaded at run time, any of its routines can be called from the main area, and any routine in the main area can be called from the overlay until the overlay is replaced by another overlay. Since @SLGD is not a valid Pascal/1000 identifier, the ALIAS compiler option must be used to rename the routine. A string containing the upper case representation of the first five characters of the segment identifier is passed as a value parameter to @SGLD.

```
TYPE
   STRING5 = PACKED ARRAY [1..5] OF CHAR;

PROCEDURE load_segment
   $ALIAS '@SGLD'$
   (segment_name : STRING5);
   EXTERNAL;
```

To load segment overlay Extrema (from previous example), the following call is required:

```
load_segment ('EXTRE')
```

After this call, the functions Most and Least can be invoked until segment overlay Extrema is replaced.

Because only one segment overlay can be in memory at one time, a segment overlay should not call @SGLD to load another overlay. @SGLD should be called only from the main program unit and its subprogram units.

Chapter 4
Declarations

In Pascal/1000 every program object must have an identifier associated with it. This includes the program itself, types, variables, constants, procedures, and functions.

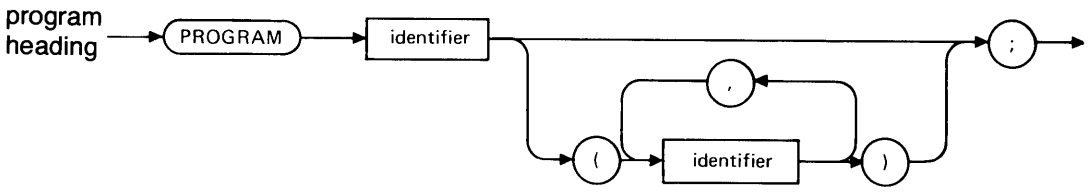
Some identifiers are predefined in Pascal/1000 (see Chapter 2), although they may be redefined by the user. All identifiers that are not predefined must be defined before they are used. There are two exceptions to this rule: program parameters (see Chapter 3) and pointer types (see below). Labels (which strictly speaking are not identifiers) must also be declared before they are used.

This chapter describes the syntax and effects of the declaration sections for all program objects.

Program Heading

The program heading associates an identifier with the program and specifies a list of the program parameters.

Syntax:

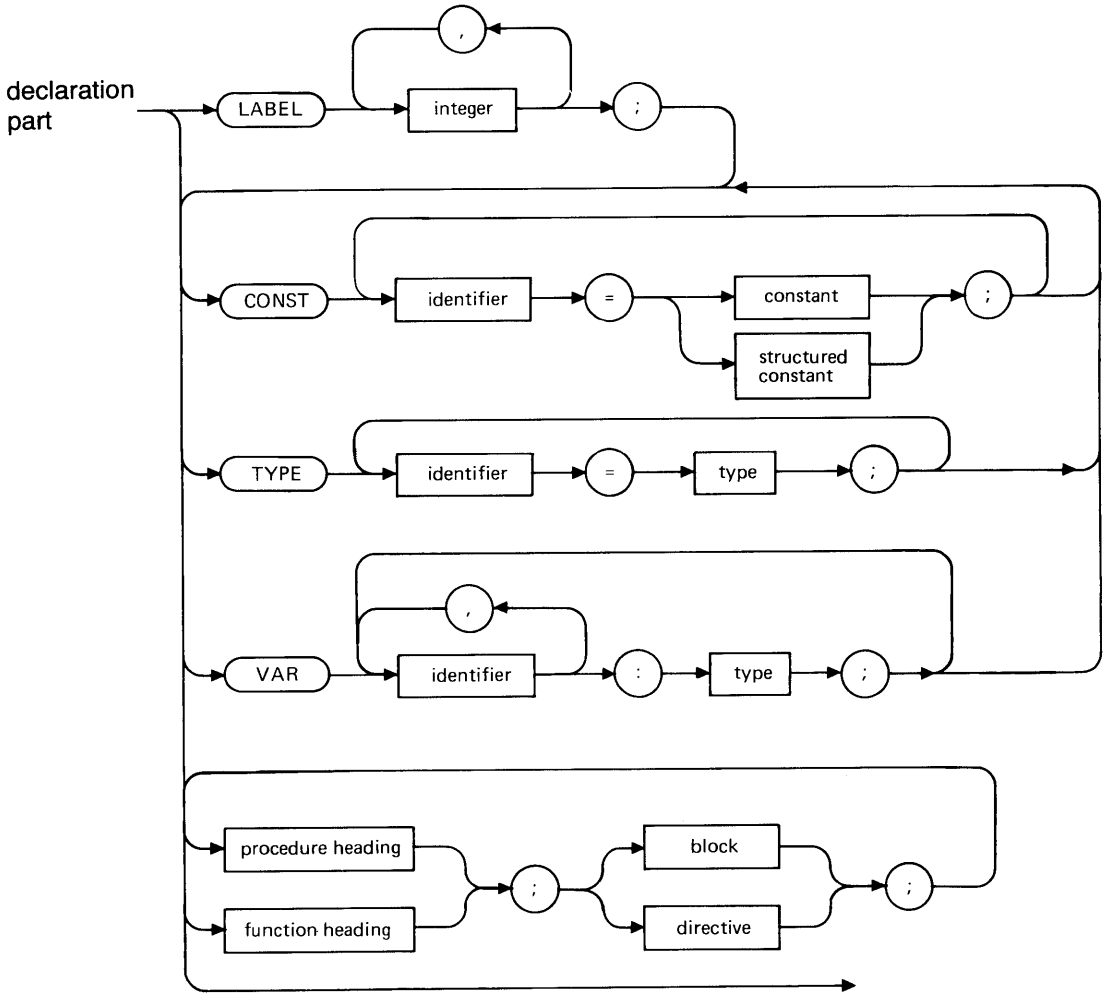


The program heading is described in Chapter 3.

Declaration Part

Each program heading (as well as PROCEDURE or FUNCTION declaration, to be described below) is followed by a declaration part.

Syntax:



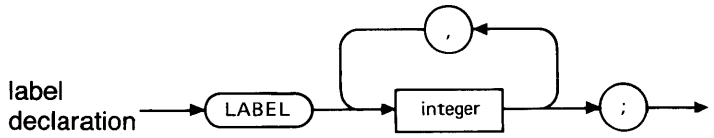
The LABEL declaration (if any) must come first. CONST, TYPE, and VAR sections may follow, in any order, and may be repeated as often as required. This is an extension of Pascal/1000; "standard" Pascal allows zero or one occurrence of CONST, TYPE, and VAR and requires that they appear in that order.

PROCEDURE and FUNCTION declarations may follow and may be repeated as often as required. This is not an extension and is permitted in "standard" Pascal.

Label Declaration

A label declaration is used to specify labels which will be used to mark statements. A label is used with the GOTO statement to transfer control to a marked statement. This is the only valid use of a label.

Syntax:



A label is an integer in the range 0 to 9999.

The labels 6 and 00000000006 are identical.

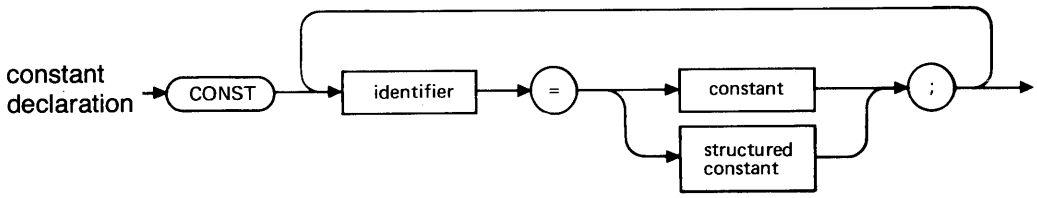
Example:

```
LABEL
   0, 00000000006, 28, 496, 8128, 9999;
```

Constant Definition

A constant definition introduces an identifier as a synonym for a constant value. The identifier may then be used in place of that value. The value of a symbolic constant cannot be changed by a subsequent constant definition or by an assignment statement.

Syntax:



Simple Constants

A simple constant is a constant expression of an unstructured type (e.g. INTEGER, BOOLEAN, CHAR, REAL, LONGREAL, subrange, or enumerated type) or a string constant. The constant expression may contain other previously defined simple constants.

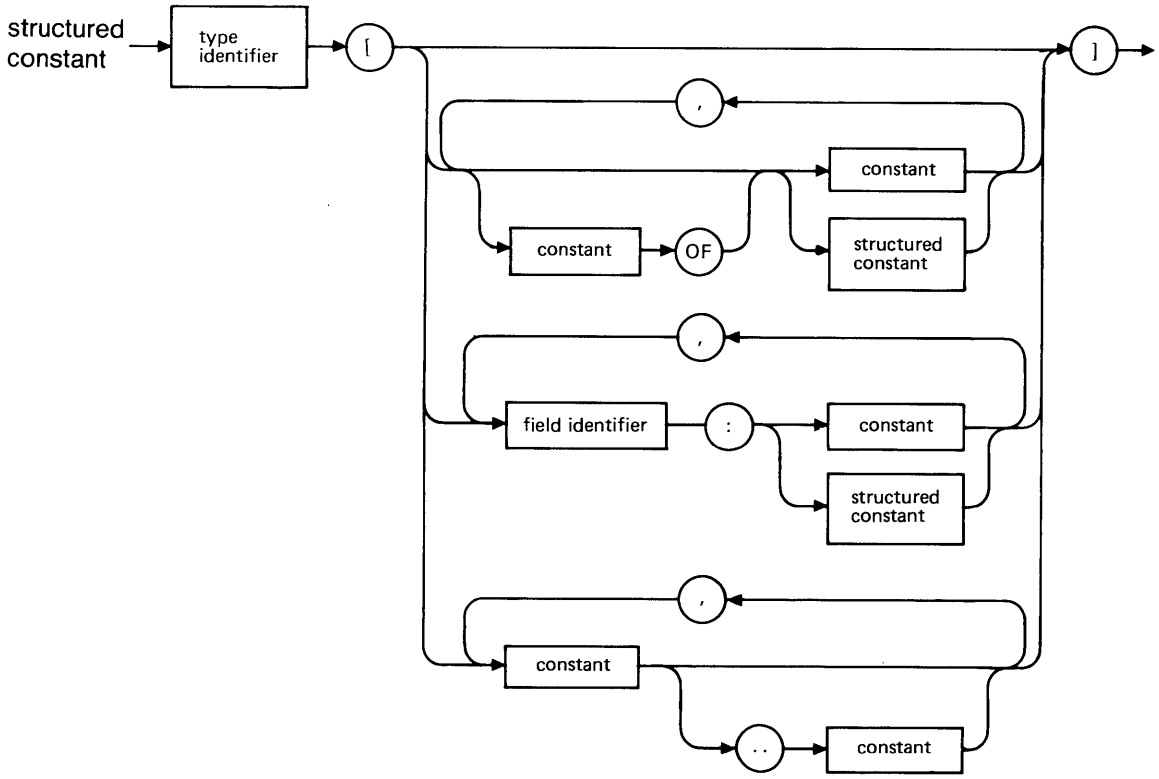
Example:

```
CONST
   pagesize = 60;
   headsize = 10;
   lines    = pagesize - headsize;
   debug    = true;
   pi       = 3.14;
   neg_pi   = -pi;
   star     = '*';
   title    = 'A Simple Test Program';
```

Structured Constants

A structured constant is a constant of a structured type (e.g. SET, RECORD, or ARRAY). The definition consists of a previously defined type identifier followed by a list of values. Values for all elements of the structured type must be specified and must have a type identical to the type of the corresponding element. Structured constants can be used to initialize variables of structured types. The individual elements of a structured constant are also available as constants. Note that structured constants are a Pascal/1000 extension.

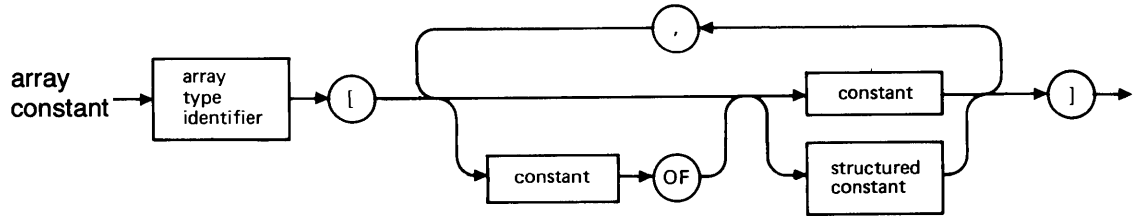
Syntax:



Array Constant

The definition of an ARRAY constant consists of the ARRAY type identifier followed by the list of values which are to be included in the constant array.

Syntax:



Examples:

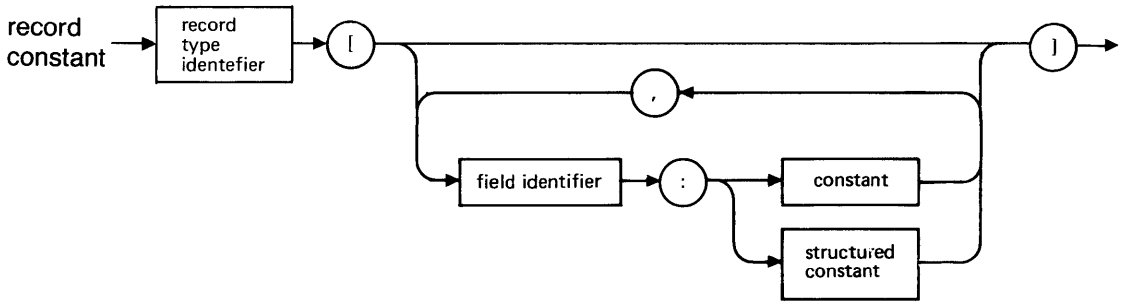
```
TYPE
   BOOLEAN_TABLE = ARRAY [1..5] OF BOOLEAN;
   TABLE         = ARRAY [1..100] OF INTEGER;
   ROW           = ARRAY [1..5] OF INTEGER;
   MATRIX        = ARRAY [1..5] OF ROW;
   COLOR         = (red, yellow, blue);
   COLOR_STRING  = PACKED ARRAY [1..6] OF CHAR;
   COLOR_ARRAY   = ARRAY [COLOR] OF COLOR_STRING;

CONST
   true_values   = BOOLEAN_TABLE [true, true, true, true, true];
   init_values1  = TABLE [100 OF 0];
   init_values2  = TABLE [60 OF 0, 40 OF 1];
   identity      = MATRIX [ ROW [1, 0, 0, 0, 0],
                            ROW [0, 1, 0, 0, 0],
                            ROW [0, 0, 1, 0, 0],
                            ROW [0, 0, 0, 1, 0],
                            ROW [0, 0, 0, 0, 1]];
   colors        = COLOR_ARRAY [COLOR_STRING ['RED', 3 OF ' '],
                                COLOR_STRING ['YELLOW '],
                                COLOR_STRING ['BLUE', 2 OF ' ']];
```

Notice that in the last example, where the array element was an ARRAY OF CHAR, that a combination of strings and characters can be used. This is the only case where the constant (string) is permitted to be of a type different than the element type (CHAR).

Record Constant

The definition of a RECORD constant consists of the RECORD type identifier followed by a list of the values to be assigned to the fields of the constant record. Each value is preceeded by the name of the field which it initializes. All fields must be initialized and may be specified in any order, except that a tag field (if present) must be initialized before any variant fields. Once the tag is initialized only the variant fields associated with that value of the tag may be initialized. If a variant is present, but no tag exists (a tagless variant), then the first variant field initialized selects the variant as if a tag had been initialized.

Syntax:



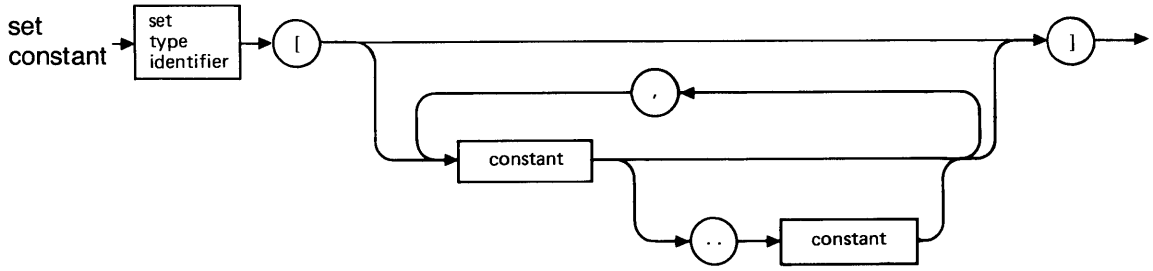
Examples:

```
TYPE
   COUNTER_RECORD = RECORD
                       pages: INTEGER;
                       lines: INTEGER;
                       characters: INTEGER;
                    END;
   REPORT_RECORD  = RECORD
                       revision: CHAR;
                       price:    REAL;
                       info:     COUNTER_RECORD;
                       CASE secret: BOOLEAN OF
                          true: (code: INTEGER);
                    END;

CONST
   no_count = COUNTER_RECORD [pages: 0, characters: 0, lines: 0];
   big_report = REPORT_RECORD
                   [revision: 'C',
                    price: 27.50,
                    info: COUNTER_RECORD
                             [pages: 6, lines: 28, characters: 496],
                    secret: true,
                    code: 8128];
```

Set Constant

The definition of a SET constant consists of the SET type identifier followed by the list of values which are to be included in the constant set.

Syntax:



Examples:

```
TYPE
   DIGITS  = SET OF 0..9;
   CHARSET = SET OF CHAR;

CONST
   all_digits = DIGITS  [0..9];
   odd_digits = DIGITS  [1, 3, 5, 7, 9];
   letters    = CHARSET ['a'..'z', 'A'..'Z'];
   no_chars   = CHARSET [];
```

Type Definition

Every literal, constant, variable, function, and expression is of one and only one type. The type defines a set of attributes:

a. The set of permissible operations that may be performed on an object of that type.

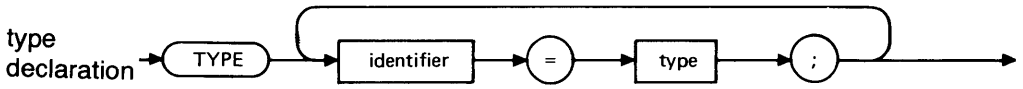
b. The set of values that an object of that type may assume.

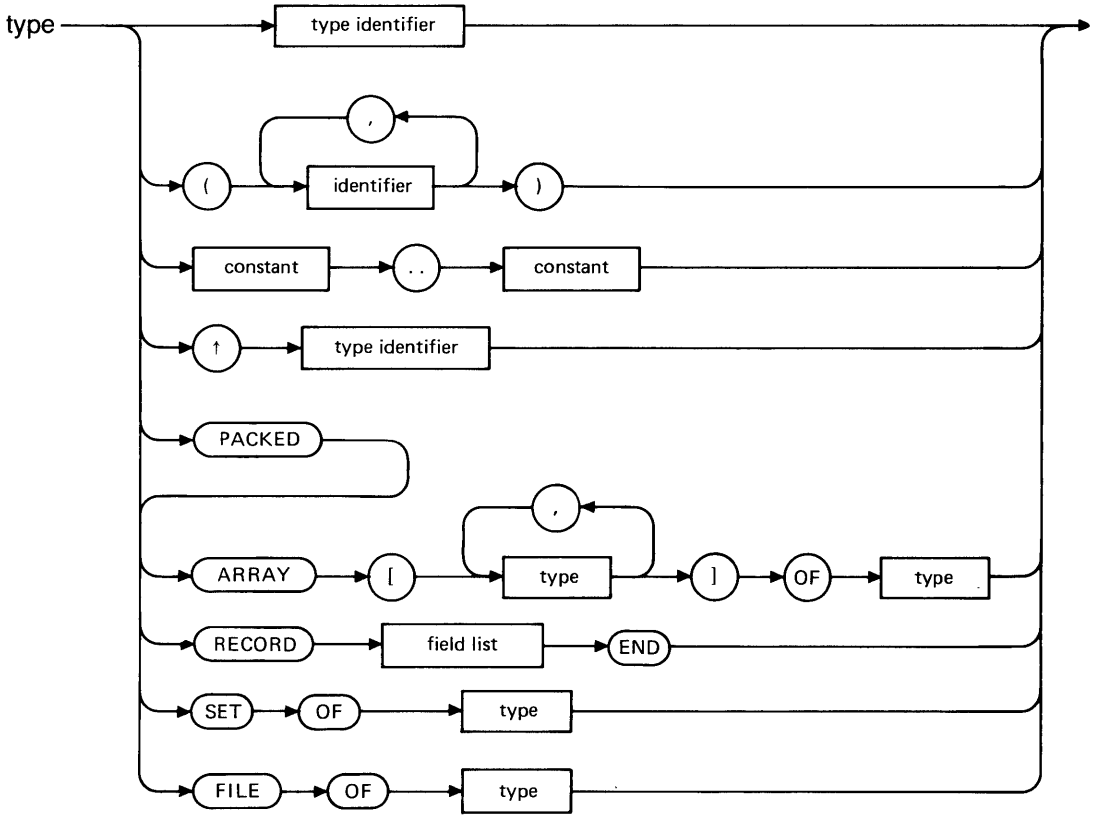
c. The amount of storage that variables of that type require.

The type of a literal is an inherent property of the literal. Pascal/1000 also predefines several commonly-used types, although these may be redefined by the user. All other types must be defined before they can be associated with a variable, constant, or function (with one exception, see Pointer Type below).

The set of permissible operations for the predefined and user-defined types is discussed in detail in Chapter 5 and summarized briefly below.

Syntax:


type
declaration
TYPE
identifier
=
type
;


type
type identifier
,
(
identifier
)
constant
..
constant
↑
type identifier
PACKED
,
ARRAY
[
type
]
OF
type
RECORD
field list
END
SET
OF
type
FILE
OF
type

Predefined Types

Boolean

The Boolean type is predefined as:

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

Variables of type Boolean normally are represented in the low order bit of one 16-bit word. The operators defined for Boolean operands and the operations that result in Boolean values are summarized below.

a. Assignment operator (:=)
b. Boolean operators (AND, OR, NOT)
c. Relational operators (<, <=, =, <>, >=, >, IN)
d. Predefined functions (eoln, eof, odd, ord, pred, succ)

Char

The CHAR type comprises the ASCII 8-bit character set.

Variables of type CHAR are normally represented in the low order 8-bit byte of one 16-bit word. The operators defined for CHAR operands and the operations that result in CHAR values are summarized below.

a. Assignment operator (:=)
b. Relational operators (<, <=, =, <>, >=, >, IN)
c. Predefined functions (chr, ord, pred, succ)

Integer

The INTEGER type is predefined as a subrange of the negative and positive integers:

```
CONST
   minint = -2147483648;
   maxint =  2147483647;
TYPE
   INTEGER = minint..maxint;
```

Minint and maxint are predefined constants. Variables of type INTEGER are normally represented in two 16-bit words. The operators defined for INTEGER operands and the operations that result in INTEGER values are summarized below.

a. Assignment operator (:=)
b. Relational operators (<, <=, =, <>, >, >=, IN)
c. Arithmentic operators (+, -, *, /, DIV, MOD)
d. Predefined functions (abs, arctan, chr, cos, exp, linepos, ln, maxpos, odd, ord, position, pred, round, sin, sqr, sqrt, succ, trunc)

Real

The REAL type is predefined as a subset of the real numbers. This subset covers the range:

```
-1.70141E+38 to -1.4693683E-39
0.0
1.4693679E-39 to 1.70141E+38
```

Variables of type REAL are represented in two 16-bit words and have an accuracy of approximately 6.9 decimal digits. The operators defined for REAL operands and the operations that result in REAL values are summarized below.

a. Assignment operator (:=)
b. Relational operators (<, <=, =, <>, >=, >)
c. Arithmetic operators (+, -, *, /)
d. Predefined functions (abs, arctan, cos, exp, ln, round, sin, sqr, sqrt, trunc)

Longreal

The LONGREAL type is predefined as a subset of the real numbers. This subset covers the range:

```
-1.701411834604692311L+38 to -1.46936793852785946L-39
0.0
1.46936793852785938L-39 to 1.701411834604692271L+38
```

Variables of type LONGREAL are represented in four 16-bit words and have an accuracy of approximately 16.5 decimal digits. The operators defined for LONGREAL operands and the operations that result in LONGREAL values are summarized below.

a. Assignment operator (:=)
b. Relational operators (<, <=, =, <>, >=, >)
c. Arithmetic operators (+, -, *, /)
d. Predefined functions (abs, arctan, cos, exp, ln, round, sin, sqr, sqrt, trunc)

Note that the LONGREAL type is a Pascal/1000 extension and is not found in "standard" Pascal.

Text

The type TEXT is predefined as:

```
TYPE
   TEXT = FILE OF CHAR;
```

with some additional special attributes, and is provided for doing common types of character- and line-oriented input and output. Variables of type TEXT are termed "text files". Each component of a text file is of type CHAR, but the sequence of characters in a text file is divided into lines. All operations applicable to a FILE OF CHAR can be performed on text files. Certain additional operations are also applicable.

One of the special attributes of text files is the ability to perform conversion from the internal form of certain types to an ASCII character representation and vice versa.

The procedure READ, when applied to a text file, can convert from an ASCII character representation to the internal form of:

a. CHAR
b. INTEGER
c. REAL
d. LONGREAL (Pascal/1000 extension)
e. subrange of INTEGER
f. PACKED ARRAY OF CHAR (Pascal/1000 extension)

The procedure WRITE, when applied to a text file, can convert from the internal form to an ASCII character representation of:

a. CHAR
b. INTEGER
c. REAL
d. LONGREAL (Pascal/1000 extension)
e. subrange of INTEGER
f. PACKED ARRAY OF CHAR (Pascal/1000 extension)
g. BOOLEAN

with additional information controlling the formatting of the ASCII character representation.

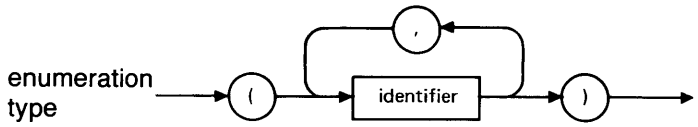
Further information on the special attributes of text files will be found in Chapter 7.

User-Defined Types

Enumeration

An enumerated type defines an ordered set of values by enumeration of the identifiers which denote these values. The ordering of the values is determined by the sequence in which the identifiers are listed.

Syntax:



In Pascal/1000 the enumerated identifiers are defined as constants, the first being assigned the integer value zero, and the others receiving successive integer values in the order of their specification.

An enumerated type may contain up to 32768 elements.

Variables of an enumerated type are normally represented as one 16-bit word. The operators defined for enumerated type operands and the operations that result in enumerated type values are summarized below.

a. Assignment operator (:=)
b. Relational operators (<, <=, =, <>, >=, >, IN)
c. Predefined functions (ord, pred, succ)

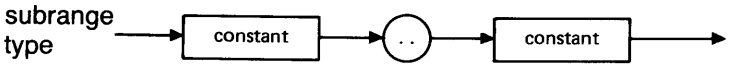
Example:

```
TYPE
   DAYS = (sunday, monday, tuesday, wednesday,
           thursday, friday, saturday);
   FRUIT = (apple, banana, cherry, grape, orange, pear);
   COLOR = (red, orange, yellow, green, blue, indigo, violet);
   FOREST_ANIMALS = (lions, tigers, bears);
```

Subrange

A subrange type is a sequential subset of another type, referred to as the base type. A subrange type is defined by specifying two elements of the base type as upper and lower bounds of the subrange.

Syntax:



where the lower bound is less than or equal to the upper bound.

A variable of a subrange type possesses all the attributes of the base type with the following exceptions:

a. Its values are restricted to the specified closed range.

b. Smaller amounts of storage may be required by variables which are components of a PACKED type.

Subrange types may only be defined over the predefined types BOOLEAN, CHAR, INTEGER, and user-defined enumeration or subrange types.

Example:

```
TYPE
   WEEKDAYS    = monday..friday;
   DAY_OF_YEAR = 1..366;
```

INTEGER subrange variables have the special property that they will be represented by one 16-bit word if the bounds lie within the subrange -32768..32767.

INTEGER Subrange	Number of words allocated for a variable of that subrange
-1000..1000	1
64000..70000	2
0..40000	2
-32768..32767	1

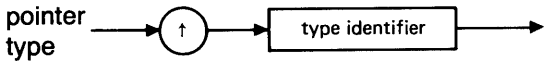
Pointer

There are two types of variables in Pascal/1000: statically-allocated and dynamically-allocated. Statically-allocated variables exist during the entire invocation of the program, procedure, or function in which they were declared, and are referred to by their identifiers.

In contrast, variables may also be created dynamically during execution. These variables are not referred to by their identifiers (for they have none), but are referred to through pointers which point to them. The creation of dynamic variables is discussed in Chapters 7 and 8.

Thus a pointer "points" to a dynamically-allocated variable. The pointer is associated with a base type (any type except FILE or an ARRAY or RECORD type containing a FILE), and may point only to dynamic variables of that type.

Syntax:



The pointer value NIL is a member of every pointer type; it points to no dynamic variable.

Variables of type POINTER normally are represented in either one or two 16-bit words (See the HEAP compiler option). The operators defined for POINTER operands and the operations that result in POINTER values are summarized below.

a. Assignment operator (:=)
b. Relational operators (=, <>)
c. Dereference operator (^)
d. Predefined procedures and functions (new, dispose, mark, release)

Pointers are an exception to the rule that identifiers must be defined before they are used. (Program parameters are the only other exception to this rule.) The base type of a pointer need not be defined before it is used in a pointer definition. This allows two disjoint types to contain pointers to each other, as in the examples below.

Examples:

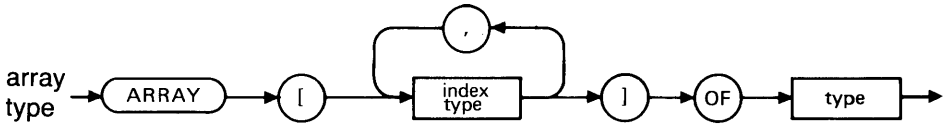
```
TYPE
   PTR1    = ^REC1;
   PTR2    = ^REC2;
   REC1    = RECORD
                f1, f2: INTEGER;
                link:   PTR2;
             END;
   REC2    = RECORD
                f1, f2: REAL;
                link:   PTR1;
             END;
```

Structured Types

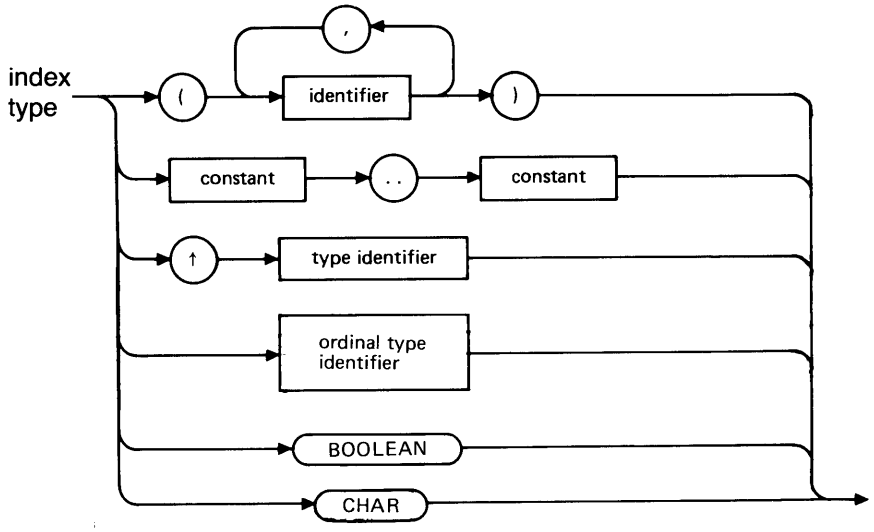
Array

An ARRAY type is a data structure consisting of a fixed number of elements which are all of the same type, called the "element" type. The elements are enumerated by an "index" type. The ARRAY type definition specifies both the element type and the index type.

Syntax:



Syntax (Cont.):



An "ordinal type identifer" is an identifier previously defined as an enumeration or subrange type.

Elements can be of any type, including FILE, RECORD, and ARRAY.

Variables of type ARRAY are normally represented as a sequence of 16-bit words.

number of words = (words per element) * (number of elements)

The operators defined for ARRAY type operands and the operations that result in ARRAY type values are summarized below.

a. Assignment operator (:=)
b. Relational operators for strings only (see below)
(<, <=, =, <>, >=, >)
c. Predefined procedures (pack, unpack)

Examples:

```
TYPE
   ARECORD = RECORD
                name: PACKED ARRAY [1..30] OF CHAR;
                age: 1..100;
             END;
   LIST    = ARRAY [1..100] OF INTEGER;
   STRANGE = ARRAY [BOOLEAN] OF CHAR;
   FLAG    = ARRAY [(red, white, blue)] OF 1..50;
   FILES   = ARRAY [1..10] OF TEXT;
   PEOPLE  = ARRAY [0..999] OF ARECORD;
```

Strings -- As a special case, an array of the form

```
{ PACKED } ARRAY [m..n] OF CHAR
```

is referred to as a "string". The array may be either packed or unpacked and the two types are compatible. A packed string can be assigned to or compared with an unpacked string. Assignment requires that the target string be the same length or longer than the source string; the target will be blank padded if necessary. Comparison will cause the shorter string to be blank padded to the length of the longer string before the comparison occurs. This is a Pascal/1000 extension. "Standard" Pascal permits only one type of string. Its form is:

```
PACKED ARRAY [1..n] OF CHAR
```

A Pascal/1000 string constant has the form of a packed string.

Multiply-Dimensioned Arrays -- If more than one index type is specified or the elements of the array are themselves arrays, then the array is said to be multiply-dimensioned. There is no arbitrary limit on the maximum number of array dimensions.

Examples:

```
TYPE
   { equivalent definitions of MATRIX }
   MATRIX = ARRAY [0..9] OF ARRAY [0..9] OF INTEGER;
   MATRIX = ARRAY [0..9, 0..9] OF INTEGER;

   SPACE  = ARRAY [0..9] OF MATRIX;

   { equivalent definitions of TRUTH }
   TRUTH  = ARRAY [1..20] OF
               ARRAY [1..5] OF
                  ARRAY [1..10] OF BOOLEAN;
   TRUTH  = ARRAY [1..20] OF
               ARRAY [1..5, 1..10] OF BOOLEAN;
   TRUTH  = ARRAY [1..20, 1..5] OF
               ARRAY [1..10] OF BOOLEAN;
   TRUTH  = ARRAY [1..20, 1..5, 1..10] OF BOOLEAN;
```

Similarly, the elements of such arrays can be indexed in different (but equivalent) ways:

```
VAR
   m: MATRIX;
   s: SPACE;

m[2, 5] is equivalent to m[2][5];
s[2, 3, 9] is equivalent to s[2][3][9] or s[2, 3][9] or s[2][3, 9]
```

Multiply-dimensioned arrays have the additional property that each dimension can be treated as an object.

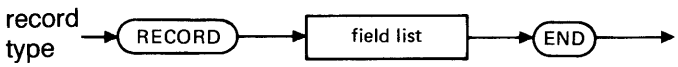
```
m[2, 5]    is an element of the second "row" of "m"
m[2]       is the second "row" of "m"
m          is the entire array
s[2, 3, 9] is an element of the third "row" of the second "plane"
             of "s"
s[2, 3]    is the third "row" of the second "plane" of "s"
s[2]       is the second "plane" of "s"
s          is the entire array
```

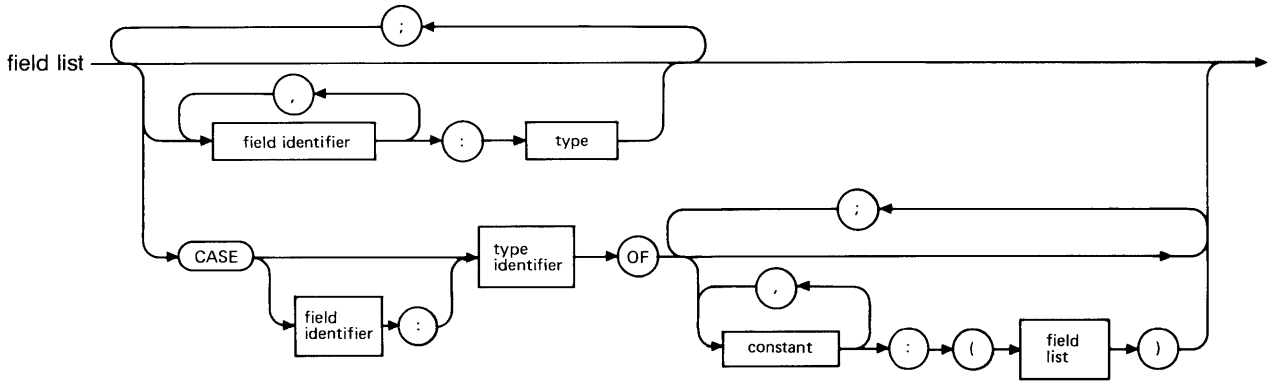
Array "rows" and "planes" can be assigned to identical "rows" and "planes", and passed as parameters to identical "rows" and "planes". This can be generalized to any number of dimensions.

Record

A RECORD type is a data structure consisting of a number of elements which are not necessarily of the same type. The RECORD type definition specifies for each element, called a "field", its type and its field identifier. The elements of records are accessed using these field identifiers.

Syntax:





A RECORD type definition may contain a "variant" part. This enables variables of type RECORD, although of identical type, to exhibit structures that differ in the number and type of their component parts. The "variant" part may contain an optional "tag" field. The value of the tag field indicates which of the variants is currently valid. If a tag field is not specified, then determination of which variant is currently valid is left to the programmer. (Actually Pascal/1000 does not check the tag field when a variant field is used. The responsibility for proper access of variants is always left to the programmer).

Each label in the variant CASE declaration must be of the same type as the tag type and subrange labels are not allowed (as they are in a CASE statement). Fields of type FILE or types which contain files are not permitted in the variant part of a RECORD. The label OTHERWISE is not allowed in the variant CASE declaration.

Variables of type RECORD are normally represented as a sequence of 16-bit words. The total number is the sum of the number of words required for the fixed part (and optional tag) plus the number of words required by the largest variant (if any).

The operator defined for RECORD types and the operation that results in a RECORD type value is:

Assignment operator (:=)

Examples:

Record with fixed part only:

```
TYPE
   TRIANGLE = RECORD
                 base,
                 height: INTEGER;
              END;
```

Record with variant part only (with tag field):

```
TYPE
   WORD_TYPE = (int, chr);
   WORD      = RECORD
                  CASE word_tag: WORD_TYPE OF
                     int: (number: INTEGER);
                     chr: (chars : PACKED ARRAY [1..2] OF CHAR);
               END;
```

Record with fixed and variant part (without tag):

```
TYPE
   POLYS   = (circle, square, rectangle, triangle);
   POLYGON = RECORD
                poly_color: (red, yellow, blue);
                CASE POLYS OF
                   circle:    (radius: INTEGER);
                   square:    (side: INTEGER);
                   rectangle: (length, width: INTEGER);
                   triangle:  (base, height: INTEGER);
             END;
```

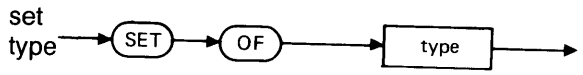
Record with nested variant part:

```
TYPE
   NAME_STRING    = PACKED ARRAY [1..30] OF CHAR;
   DATE_INFO      = PACKED RECORD
                       mo: (jan, feb, mar, apr, may, jun,
                            jul, aug, sep, oct, nov, dec);
                       da: 1..31;
                       yr: 1900..2100;
                    END;
   MARITAL_STATUS = (married, single, divorced);
   PERSON_INFO    = RECORD
                       name: NAME_STRING;
                       born: DATE_INFO;
                       CASE status: MARITAL_STATUS OF
                          married,
                          divorced: (when:   DATE_INFO;
                                     CASE has_kids: BOOLEAN OF
                                        true: (how_many: 1..50);
                                    )
                    END;
```

Set

A SET type defines a powerset (set of all subsets) of an enumeration or subrange type called the "base" type.

Syntax:



The base type of a set may contain up to 32767 elements.

Variables of type SET are normally represented as a series of Boolean values (each 1 bit) which indicate the presence or absence in the set of each element of the base type. The amount of storage required for a variable of type SET is determined as follows:

For a SET of 16 or less elements:

One 16-bit word.

For a SET of more than 16 elements:

((number of elements + 15) DIV 16 + 1) 16-bit words

The operators defined for SET operands and the operations that result in SET values are summarized below.

a. Assignment operator (:=)
b. Union operator (+)
c. Intersection operator (*)
d. Difference operator (-)
e. Subset relational operator (<=)
f Superset relational operator (>=)
g. Equality relational operators (=, <>)
h. Element inclusion (IN)

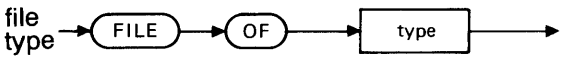
Examples:

```
TYPE
   CHARSET   = SET OF CHAR;
   FRUIT     = (apple, banana, cherry, peach, pear, pineapple);
   FRUITSET  = SET OF FRUIT;
   SOMEFRUIT = SET OF apple..cherry;
   CENTURY20 = SET OF 1901..2000;
```

File

A FILE type definition specifies a data structure consisting of a sequence of components which are all of the same type. Files are usually associated with peripheral storage devices, and their length is not specified in the program.

Syntax:



The component type of a FILE can be any type except FILE or a type which contains a file.

The operations allowed on variables of type FILE are described in Chapter 7.

Examples:

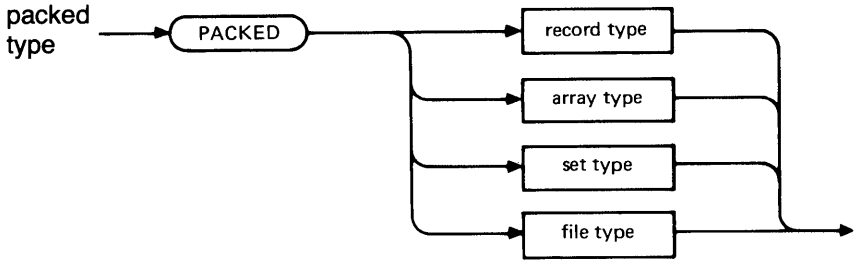
```
TYPE
   PERSON       = RECORD
                     name: PACKED ARRAY [1..30] OF CHAR;
                     age:  1..100;
                  END;
   BIT_VECTOR   = ARRAY [1..100] OF BOOLEAN;
   PERSON_FILE  = FILE OF PERSON;
   DATA_FILE    = FILE OF INTEGER
   VECTOR_FILE  = FILE OF BIT_VECTOR;
```

Packed Type Modifier

The representation of a variable in Pascal/1000 is usually determined by the compiler. Ease of access is given priority over storage compactness. For example, Boolean variables occupy a 16-bit word instead of a single bit, and character variables occupy a 16-bit word instead of an 8-bit byte.

There are times, however, when the programmer needs smaller amounts of storage allocated to certain data items, even if this requires less efficient access. The programmer can indicate this to the compiler by prefixing the definition of a structured type with the symbol PACKED.

Syntax:



Non-structured components of PACKED structured types are allocated the smallest amount of storage required to represent all the possible values of each component in a manner consistent with the following rules.

a. A component which requires more than one 16-bit word of storage will begin on a 16-bit word boundary.

b. A component which requires one 16-bit word or less of storage will not cross a word boundary.

c. A component which is a set of more than 16 elements will use a whole number of words, even if all of the last word is not required by the set.

Structured components of a structured type are not affected by the PACKED type modifier.

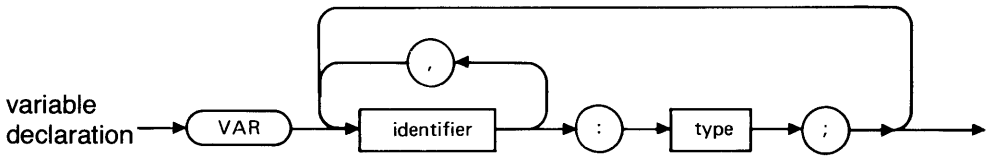
The operations allowed on data of a PACKED data type are the same as those allowed for data that is not PACKED, with the exception that components of a packed structure cannot be passed as VAR (call-by-reference parameters). This exception applies as well to the standard procedures READ and READLN.

The standard procedures PACK and UNPACK can be used to assign components from a unpacked array to a packed array, and vice-versa (See Chapter 7).

Variable Declaration

A variable declaration introduces an identifier as a variable of a specified type.

Syntax:



Each variable is a statically-declared object which occupies storage and is accessible for the duration of the program, procedure, or function in which it is declared.

Every declaration of a file variable F with type FILE OF T implies the declaration (by the compiler) of a buffer variable of type T. This variable, denoted F^, is used to access the components of the file F.

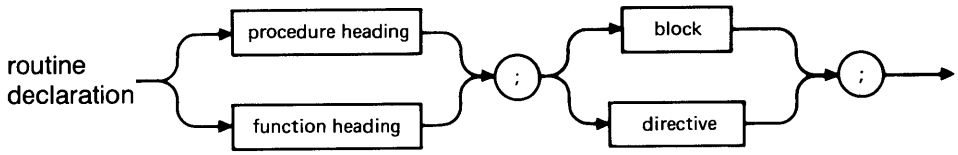
Examples:

```
VAR
   { predefined types }
   pagecount,
   linecount,
   charcount:           INTEGER;
   currentgrade:        CHAR;
   average:             REAL;
   standard_deviation:  LONGREAL;
   debugging, done:     BOOLEAN;
   terminal_file:       TEXT;
   some_files:          ARRAY [0..4] OF TEXT;
   many_numbers:        ARRAY [-1000..1000] OF INTEGER;
   many_chars:          PACKED ARRAY [1..10000] OF CHAR;
   many_truths:         ARRAY [0..1999] OF BOOLEAN;

   { user-defined types }
   today, tomorrow:     DAYS;
   beware_of:           FOREST_ANIMALS;
   ordinal_date:        DAY_OF_YEAR;
   first_recl:          PTRI;
   friends:             PEOPLE;
   shape:               POLYGON;
   temp_shape:          ^POLYGON;
   we_have:             FRUITSET;
   number_right:        0..100;
   personnel_file:      PERSON_FILE;
   have_seen:           SET OF FOREST_ANIMALS;
   saved_shapes:        FILE OF POLYGON;
   stores_have:         ARRAY [1..200] OF FRUITSET;
```

Routine Declaration

A routine is a named block that is activated by referring to its identifier. A routine can be either a procedure or a function. Procedures serve to define parts of programs which can be activated by procedure statements. Functions serve to define parts of programs which compute a single value of any type (except FILE or any type containing a FILE) for use in evaluating an expression. A function is activated by using the function identifier within an expression.

Syntax:

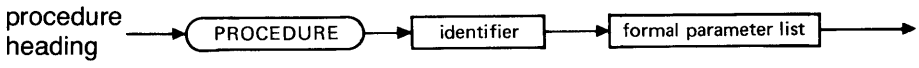


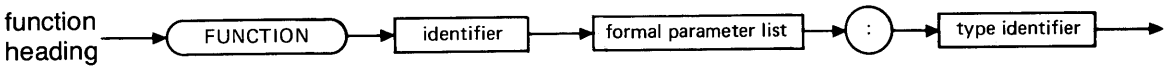
The routine heading specifies the identifier to be associated with the routine, any parameters to the routine, and the type of the result if the routine is a function. The routine block contains a declaration part which specifies the labels, constants, types, variables, and routines which are local to the routine being declared, and a compound statement (body) describing the executable statements of the routine.

Routine Heading

The heading of a procedure or function defines the manner in which the routine interacts with other routines and the main program.

Syntax:



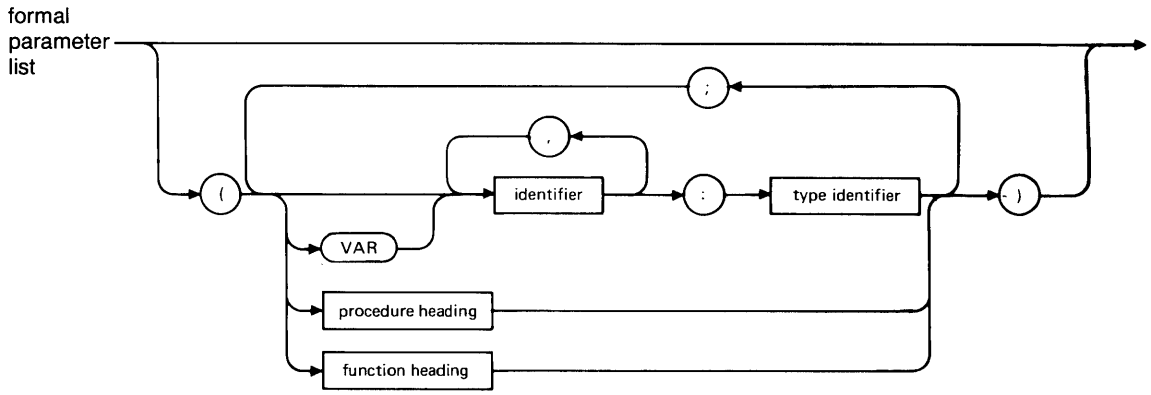


The identifier following the reserved word PROCEDURE or FUNCTION is the name by which the routine is known in the source code. Certain restrictions apply to level-1 routine names. These are discussed in section Level-1 Routines of this Chapter.

Formal Parameter List

This is a list of the formal parameters of the routine. When the routine is activated, a list of actual parameters is provided, and these are substituted (as specified below) for the corresponding formal parameters. The correspondence is established by the ordering of the parameters in the list. The list of actual parameters must be compatible with the formal parameter list. This compatibility is described in section Parameter List Compatibility.

Syntax:



There are four kinds of parameters: value, variable, procedure and function. In this chapter, whatever is true for a parameter is also true for each parameter in the same parameter group (i.e. what is true for x of x,y,z:REAL is also true for y and z).

Value Parameters

The actual parameter corresponding to a formal value parameter must be an expression (of which a variable is a simple case) which is assignment compatible with the type of the formal value parameter. The corresponding formal parameter represents a local variable in the activated routine. As its initial value, this local variable receives the value of the expression used as the actual parameter. The routine may change the value of this local variable without affecting the actual parameter.

A formal value parameter of type "string" is compatible with an actual parameter of type "string" which has the same or fewer elements. If the actual parameter has fewer elements, the extra elements of the formal parameter will be blank filled.

Components of a packed type can only be passed as value parameters.

In programs using HEAP 2, the size of an actual value parameter in EMA cannot be more than 1023 16-bit words of memory.

Variable Parameters

A variable parameter is often referred to as a "call-by-reference" parameter. The actual parameter corresponding to a formal variable parameter must be a variable. The corresponding formal parameter must be preceded be the reserved word VAR and it represents the actual parameter in the activated routine. Any operation performed on the formal parameter is performed directly on the actual parameter.

File parameters and parameters containing files may only be passed as variable parameters.

Procedure and Function Parameters

A formal parameter can be a routine heading. The corresponding actual parameter is the routine identifier of a routine with a compatible parameter list. The formal routine parameter represents the actual routine during the activation of the called routine in which it appears as a parameter.

Example of program using functions as parameters:

```
PROGRAM sample (input, output);

VAR
   test: BOOLEAN;

FUNCTION chek1 (x, y, z: REAL): BOOLEAN;
BEGIN
   {perform some type of validity check on x, y, z
    and return appropriate value}
END;

FUNCTION chek2 (x, y, z: REAL): BOOLEAN;
BEGIN
   {perform an alternate validity check on x, y, z
    and return appropriate value}
END;

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

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

Parameter List Compatibility

An actual and formal parameter list are compatible if they contain the same number of parameters and the corresponding parameters match. Parameters match when:

a. They are both value parameters of assignment compatible types.

b. They are both variable parameters of identical type.

c. They are both procedure parameters with compatible parameter lists.

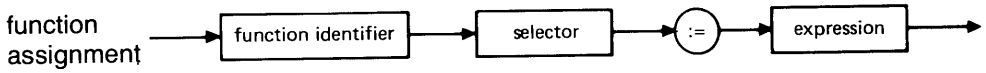
d. They are both function parameters with compatible parameter lists and identical result types.

Function Results

The heading of a function specifies the function identifier, the formal parameters of the function, and the function type. The type of a function may be any type, except a file type or a type containing a file.

Within the function body there must be at least one assignment statement assigning a value to the function identifier.

Syntax:



This assignment statement is a simple statement that determines the function result. The selector can be used when the type of the function result is a structured type. For more information on the use of selectors, refer to section Expressions of Chapter 5.

A compile-time error occurs if the body of the function does not contain an assignment to the function identifier.

Routine Declaration Part

The declaration part of a procedure or function contains the declarations of local constants, types, labels, variables, and routines. The routine declaration part has the same form as the program declaration part.

Routine Body

The body of a procedure or function is a compound statement which describes the operations on global, intermediate, and local identifiers. The syntax for constructing a routine body is the same as that for constructing a program body and is discussed in detail in Chapter 5. If the routine is a function, there must be an assignment statement within the body which assigns a value to the function identifier.

Level-1 Routines

Level-1 routines, are routines that are not declared within any other routine. The Pascal/1000 compiler creates entry points for level-1 routines so they are accessible from outside the compilation unit in which they are declared. Because of this the operating system requires that level-1 routines must have names which are unique within the first five characters. The program name must also be unique, within the first five characters, with respect to all level-1 routine names. Level-1 routines can be made "local" to a compilation unit (not entry points) with the VISIBLE compiler option and are then not subject to the five character uniqueness limitation.

Alias

The ALIAS compiler option allows a routine to be referred to by a name in the source code that is different from the name used in the object code. Some examples of the necessary uses of ALIAS are:

a. Using library routines which have names which are not legal Pascal/1000 identifers (see EXTERNAL directive).

b. Changing the name of a level-1 routine so that it will be unique within the first five characters.

c. Defining several routines with different parameter lists to represent the same routine (e.g. the EXEC system routine).

Only level-1 routines can be given an ALIAS. Such a routine then has two names. Each is recognized in its own domain. In the heading

```
PROCEDURE try_test_1; $ALIAS 'TRY1'$
```

the name try_test_1 is recognized as the routine's name in the Pascal/1000 source code, and TRY1 is the name used for all references to the routine in the code generated by the compiler. Nowhere in the source code may the name TRY1 be used to refer to the routine. Further information on ALIAS and VISIBLE may be found in Appendix D.

Examples:

```
a. TYPE
      NAME_TYPE = PACKED ARRAY [1..5] OF CHAR;
   PROCEDURE load_segment $ALIAS '@SGLD'$
      (name: NAME_TYPE); EXTERNAL;

b. PROCEDURE try_test_1; $ALIAS 'TRY1'$
   PROCEDURE try_test_2; $ALIAS 'TRY2'$

c. TYPE
      INT = -32768..32767;
      NAME_TYPE = PACKED ARRAY[1..5] of CHAR;
      TIME_ARRAY = PACKED ARRAY[1..5] of INT;
   PROCEDURE execl1 $ALIAS 'EXEC'$
      (VAR code:INT; time:TIME_ARRAY);
   PROCEDURE execl2 $ALIAS 'EXEC'$
      (VAR icode:INT; name:NAME_TYPE; resl,mult,ofst:INT);
```

Directives

All routines must be declared before they are called. If the routine's block does not immediately follow the routine heading then a directive must be used to inform the compiler of the location of the block.

FORWARD

Calls to a routine may precede the full definition of the routine if a FORWARD declaration comes before the first call to the routine. A FORWARD declaration consists of the routine heading followed by the predefined identifier "FORWARD". The routine must be fully declared before the end of the current scope. In the routine heading of the full declaration only the word PROCEDURE or FUNCTION followed by the routine identifier is allowed. The parameter list (and result type in the case of functions) may not be respecified.

Example:

```
FUNCTION exclusive_or (x,y: BOOLEAN): BOOLEAN;
   FORWARD;

.
.
.

FUNCTION exclusive_or;
  BEGIN
    exclusive_or := (x AND NOT y) OR (NOT x AND y);
  END;
```

EXTERNAL

External routines are routines that are declared outside of the compilation unit in which they are called. External routines may be a part of the operating system, part of a library, part of a Pascal/1000 subprogram or segment, or a routine written in FORTRAN or ASSEMBLY. Before a routine of this type can be called, an EXTERNAL declaration must be made. This declaration consists of the routine heading, including formal parameter list and result type (for functions only), followed by the predefined identifier "EXTERNAL".

Example:

```
PROCEDURE external_routine (VAR a, b, c: REAL);
   EXTERNAL;
```

Only level-1 routines can be declared external.

Recursive Routines

A routine that calls itself is a recursive routine. Use of the routine's identifier within the routine's body indicates recursive execution of the routine. It is also possible for a routine R to call a routine Q which in turn calls routine R. This is called indirect recursion and is often a place where the FORWARD directive is useful. Any Pascal procedure or function may be called recursively if the RECURSIVE compiler option is ON at the time the routine is first declared (See Appendix D).

Recursion is accomplished by generating new local variables dynamically when a routine is called recursively. This is discussed further in section Data Management of Chapter 8.

Example:

```
{ calculate factorial recursively }

FUNCTION factorial (n: INTEGER): INTEGER;
BEGIN
   IF n = 0 THEN
      factorial := 1
   ELSE
      factorial := n * factorial(n-1);
END;
```

Scope

Certain objects in a Pascal/1000 program are associated with a "SCOPE". These objects are:

a. labels
b. constants
c. types
d. variables
e. formal parameters
f. routines

The scope of an object is the part of the program in which an object can be used. In Pascal/1000 an object's scope is defined from its point of declaration or definition.

The precise scope rules for Pascal/1000 are:

a. The scope of an object extends over the whole of the program, procedure, function, or record definition in which it is declared, with the exception noted in (b).

b. An object defined at some outer level of scope is inaccessible from an inner level if the same identifier is used to define a new object.

c. No two identifiers may have the same spelling in a scope. Once an identifier is used or defined in a scope it may not be redefined.

d. The definition of an object must precede its use, with the exception of pointer-type identifiers, program parameters, and forward-declared procedures or functions.

Within a routine declaration, the declaration part specifies local labels, constants, types, variables, and routines. The body of the routine specifies the actions of the routine. Operations in the body may use variables, constants, labels, types, and parameters that are:

a. Global objects: declared in the program level declaration part.

b. Intermediate objects: declared in an enclosing routine declaration part (including parameters to the enclosing routine).

c. Local objects: declared in the routine's declaration part (including parameters to the routine).

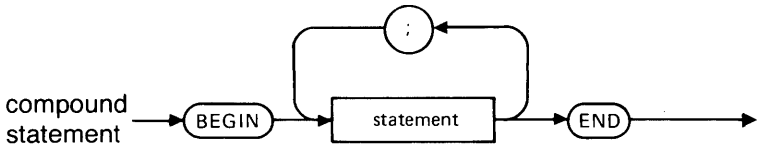
Example:

```
PROGRAM levels;

{"global" label, constant, type, and variable definitions and }
{declarations can use predefined types and constants          }

   PROCEDURE proc1 (a, b, c: INTEGER);

   {proc1 local labels, constants, types, and variables}
   {can use global types and constants                 }

      PROCEDURE proc2 (x, y, z: REAL);

      {proc2 local labels, constants, types, and variables}
      {can use proc1 and global types and constants       }

      BEGIN {proc2}
        {can use proc2                                                }
        {can use proc2: local labels, constants, types, variables,}
        {               and parameters x,y,z                          }
        {can use proc1                                                }
        {can use proc1: local labels, constants, types, variables,}
        {               and parameters a,b,c                          }
        {can use global labels, constants, types, and variables     }
      END;  {proc2}

   BEGIN {proc1}
     {can use proc1                                                  }
     {can use proc1: local labels, constants, types, variables,}
     {               and parameters a,b,c                            }
     {can use proc2                                                  }
     {can use global labels, constants, types, and variables     }
   END;  {proc1}

BEGIN {program levels}
  {can use global labels, constants, types, and variables}
  {can use proc1                                         }
END.
```

Chapter 5
Executable Parts

The executable part, or body, of a program, procedure, or function is a compound statement containing a sequence of Pascal statements. Without a body, the program or routine would perform no useful work.

Syntax:



When control is passed to the program or routine, the statements in the body are executed in the order specified. However, certain statements may alter this normal flow of control in order to achieve effects such as conditional branching, looping, or invoking a procedure or function. After the last statement in the body of a routine has executed, control is returned to the point in the program from which the routine was called. After the program's last statement has executed, the program terminates.

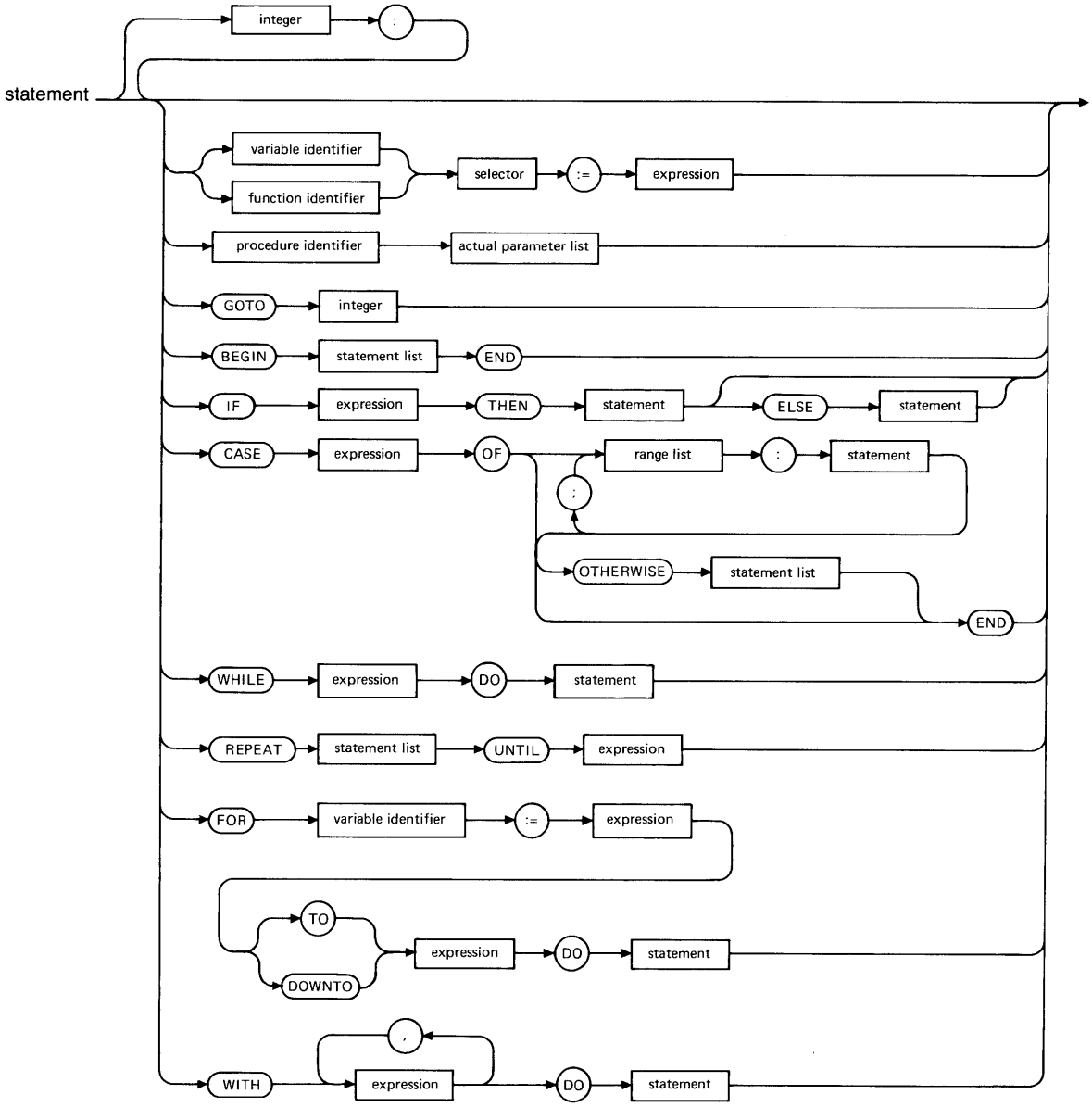
Statements

A statement is a sequence of symbols, reserved words, and expressions, used to perform a specific set of actions on a program's data, or control the program's flow.

The following can be performed using Pascal statements:

a) assign a value to a variable (Assignment statement)
b) invoke a procedure (Procedure statement)
c) choose a certain set of actions based on certain conditions (IF and CASE statements)
d) repeat a set of actions (WHILE, REPEAT, FOR statements)
e) allow record fields to appear without naming the record (WITH statement)
f) transfer control to another part of the program (GOTO)
g) treat a group of statements as one (Compound statement)
h) do nothing (Empty statement)

Syntax


statement
integer
:
variable identifier
function identifier
selector
:=
expression
procedure identifier
actual parameter list
GOTO
integer
BEGIN
statement list
END
IF
expression
THEN
statement
ELSE
statement
CASE
expression
OF
range list
:
statement
;
OTHERWISE
statement list
END
WHILE
expression
DO
statement
REPEAT
statement list
UNTIL
expression
FOR
variable identifier
:=
expression
TO
DOWNTO
expression
DO
statement
WITH
,
expression
DO
statement

The assignment, procedure, GOTO, and empty statements are commonly called "simple statements". The IF, CASE, WHILE, REPEAT, FOR, and WITH statements are referred to as "structured statements" because they may themselves contain other statements. Pascal/1000 puts no restrictions on the number of structured statements that may be nested, nor on the number of statements in a body.

Statement Labels

A statement label may be associated with any statement in a program or routine body.

The appearance of a label before a statement serves to associate it with the statement.

The label must have appeared in the LABEL declaration section of the program or routine in which it is defined. The label is used as the object of a GOTO statement.

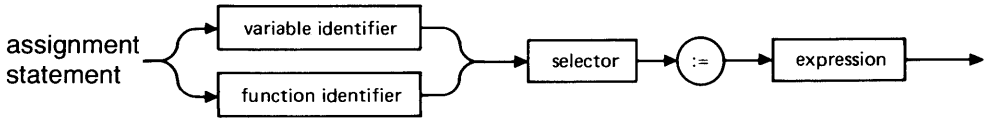
Example:

```
PROCEDURE show_labels;
LABEL 500, 501;
TYPE
   INDEX = 1..10;
VAR
   i: INDEX;
   target: INTEGER;
   a: ARRAY [INDEX] OF INTEGER;
BEGIN {show_labels}
   ...
   FOR i := 1 TO 10 DO
      IF target = a [i] THEN
         GOTO 500;
   writeln (' Not found');
   GOTO 501;
 500:
   writeln (' Found');
 501:
END;   {show_labels}
```

Assignment Statement

The assignment statement is used to replace the old value of a variable with a new value. The new value is computed from an expression prior to the assignment.

Syntax:



The variable can be of any type except a file type, or a structure containing a file.

The variable's type and the result type of the expression must be "assignment compatible" (refer to Type Compatibility). This means the types must be identical except for a few cases in which either an implicit conversion is done, or a run-time check is performed which verifies that the value of the expression is assignable to the variable. These conversions are further discussed under Arithmetic Operators, Set Operators, and Relational Operators.

The function identifier is subject to the same restrictions as variables. In addition, the function identifier may be assigned a value only within the body of the function or within the body of a routine enclosed by the function. An assignment in the function's body must always be made.

If the function returns a structured type, it is sufficient to assign a value to only one of its components. If this is done note that the rest of the structure remains undefined.

In Pascal/1000 an additional restriction exists for HEAP 2 (EMA) programs: a variable, or component of a variable, in the heap requiring 1024 or more 16-bit words of memory cannot be assigned. In order to assign a large structure such as this, it must be done one component (< 1024 words) at a time.

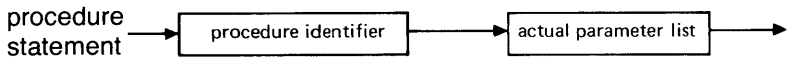
Example:

```
FUNCTION show_assign: INTEGER;

TYPE
   REC = RECORD
      f: INTEGER;
      g: REAL;
   END;

   INDEX = 1..3;
   TABLE = ARRAY [INDEX] OF INTEGER;

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

VAR
   s: INTEGER;
   a: TABLE;
   i: INDEX;
   r: REC;
   pl,
   p: ^INTEGER;


   FUNCTION show_structured: REC;
   BEGIN
      show_structured.f := 20;    {assign part of the record  }
      show_structured := cr;      {assign the whole record    }
      show_assign := 50;          {assign to an outer function}
   END;    {show_structured}


BEGIN
                     {Assign to a:}

   s := 5;                 {simple variable              }
   a := ct;                {array variable               }
   a [i] := s + 5;         {subscripted array variable   }
   r := cr;                {record variable              }
   r.f := 5;               {selected record variable     }
   p := pl;                {pointer variable             }
   p^ := r.f - a [i];      {dereferenced pointer variable}
   show_assign := p^;      {function result variable     }

END;   {show_assign}
```

Procedure Statement

The procedure statement transfers control to a procedure. After the procedure has executed, control is returned to the statement following the procedure call.

Syntax:



The procedure identifier must be the name of either a predefined procedure or a procedure declared previously in a procedure declaration. The declaration may be an actual declaration (i.e. heading plus body), a forward declaration, an external declaration, or it may be the declaration of a procedural parameter.

If the formal declaration of the procedure includes a parameter list, the procedure statement must supply actual parameters to be substituted for the formal parameters in the body of the routine. The actual parameter list must agree in number, order and type with the formal list. There are four kinds of parameters, each of which has different effects and compatibility requirements (refer to Routine Declarations in Chapter 4).

Pascal/1000 provides several compiler options and directives associated with procedures that affect such things as:

a) procedure's calling sequence ($DIRECT$)
b) external name ($ALIAS$)
c) recursive attribute ($RECURSIVE$)
d) error return ($ERROREXIT$)
e) location of the procedure's body (FORWARD, EXTERNAL)
f) parameter addressing ($HEAPPARMS$)

These are discussed in Routine Declarations (Chapter 4), and in Compiler Options (Appendix D).

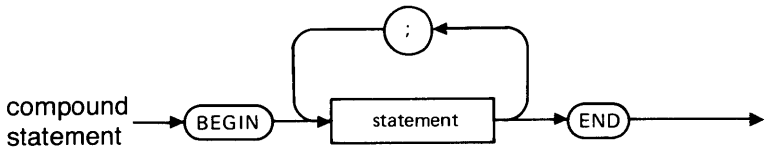
Example:

```
PROGRAM show_call (output);

PROCEDURE external_proc          {an external declaration          }
   (e1: INTEGER;
    e2: REAL);  EXTERNAL;

PROCEDURE forward_proc           {a forward declaration            }
   (f1: INTEGER;
    f2: REAL); FORWARD;

PROCEDURE actual                 {an actual procedure declaration}
   (a1: INTEGER;
    a2: REAL);
BEGIN
   IF a2 < a1 THEN
      actual (a1, a2-a1)         {a recursive call                 }
   ...
END;

PROCEDURE outer                  {another actual declaration       }
   (a: INTEGER;
    PROCEDURE proc
       (p1: INTEGER;
        p2: REAL));

   PROCEDURE inner;              {a nested procedure               }
      BEGIN
         actual (50, 50.0);
      END;

BEGIN                               {Calling:}
   writeln (output, 'Hi');       {a predefined procedure           }
   actual (2, 4.0);              {a procedure whose body has been seen}
   inner;                        {an inner procedure               }
   external (2, 4.0);            {an external procedure            }
   proc (2, 4.0);                {a procedural parameter           }
END;    {outer}

PROCEDURE forward;               {the actual declaration for a forward
                                  procedure                        }
BEGIN                            {call a routine with procedural
                                  parameters:                      }
   outer (10, external);         {external,                       }
   outer (20, forward);          {forward,                        }
   outer (30, actual);           {and actual procedures           }
END;

BEGIN  {show_call}
   forward (3, 5.0);
END.   {show_call}
```

Compound Statement

The compound statement is used as a means of treating a group of statements as a single statement.

Syntax:



The statements within the BEGIN...END block are executed in the order written. The compound statement has two primary uses:

1) The body of a procedure, function, or program is a compound statement.

2) Structured statements may themselves contain other statements. Usually where a sub-statement is allowed, the syntax calls for a single statement. The compound statement may be used in these places in the event that several statements need to be executed instead of just one.

Compound statements can be used as part of IF, CASE, WHILE, REPEAT, FOR, and WITH statements. They can also be used inside of other compound statements to logically group statements together. There are two places in the language, however, where a compound statement is allowed but unnecessary. Neither of the following groups of statements need be bracketed by BEGIN...END.

1) The statements between REPEAT and UNTIL.

2) The statements between OTHERWISE and the END of the CASE statement.

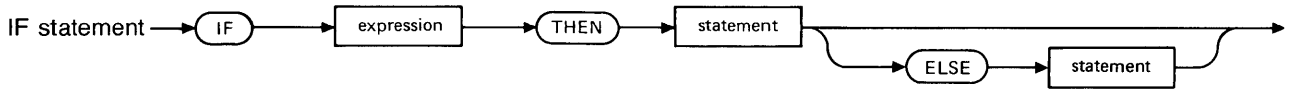
Examples:

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


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

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

IF Statement

The IF statement is used to perform one of two possible actions based on a given condition.

Syntax:



The expression must be of type Boolean. The statements may be any Pascal statements, including other IF statements.

When the IF statement is executed, the Boolean expression is evaluated to either true or false. One of three actions is then performed:

1) If the value was true, the statement following the THEN is executed.

2) If the value was false and ELSE was specified, the statement following the ELSE is executed.

3) If the value was false and no ELSE was specified, no action is taken.

After one of the above has been performed, execution resumes at the statement following the entire IF statement.

The following IF statements are equivalent:

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

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

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

Note that a semicolon may not separate the statement after the THEN and the ELSE part of the same IF statement.

A common use of the IF statement is to select an action based on several choices, similar to the use of the CASE statement. This may be expressed in the following form:

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

This form is often useful where CASE statements cannot be used (CASE selectors cannot be of type real or any string type, for example).

Unless the PARTIAL_EVAL compiler option is off, the Boolean expression is evaluated using partial, or "shortcircuit" evaluation. (refer to Boolean Operators) Partial evaluation usually results in more efficient code, but it is is also a great convenience to the programmer. For example, the statement:

```
IF index IN [lower..upper] THEN
   IF ptr_array [index] <> nil THEN
      IF ptr_array [index]^ = 5 THEN
         found_it := true;
```

can, with partial evaluation turned on, be written as:

```
IF  (index IN [lower..upper])
AND (ptr_array [index] <> nil)
AND (ptr_array [index]^ = 5) THEN
   found_it := true;
```

In the first example, nested IF statements are required in order to prevent run-time errors from occurring: if index is not between lower and upper, then the reference to ptr_array [index] would fail; if index is valid, but ptr_array [index] is nil, then ptr_array [index]^ would fail. Using partial evaluation, the nested IF's are unnecessary because evaluation of the Boolean expression stops when the result is known. Thus if index is invalid, the expression (ptr_array [index] <> nil) is never evaluated, preventing a range violation. Likewise, if ptr_array [index] is nil, the expression (ptr_array [index]^ = 5) is never evaluated, preventing a pointer violation.

It should be noted, however, that not all Pascal compilers do partial evaluation, and programs relying on this feature may not work when compiled with another compiler.

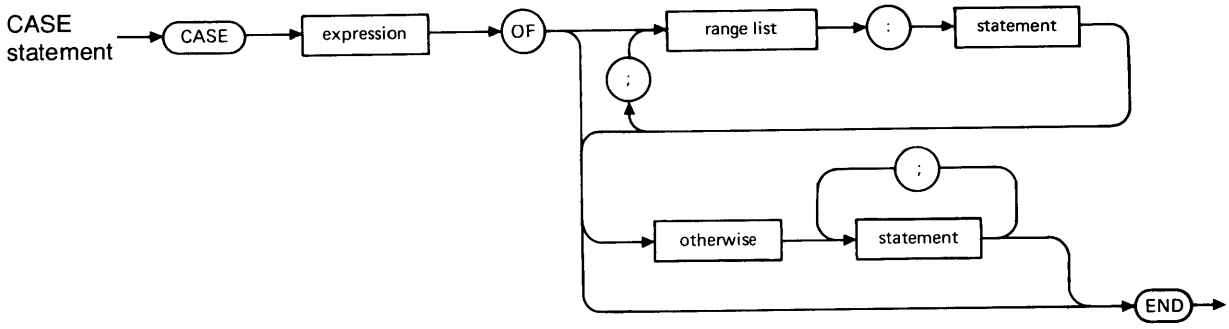
Example:

```
PROGRAM show_if (input, output);

VAR
   i,j: INTEGER;
   s,t: PACKED ARRAY [1..5] OF CHAR;
   found: BOOLEAN;

BEGIN
   { ... }

   IF i = 0 THEN writeln ('i = 0');    {IF with no ELSE        }

   IF found THEN                       {IF with an ELSE part   }
      writeln ('Found it')
   ELSE
      writeln ('Still looking');



   IF i = j THEN                       {Select among different}
      writeln ('i = j')                {Boolean expressions    }

   ELSE IF i < j THEN
      writeln ('i < j')

   ELSE {i > j}
      writeln ('i > j');



   IF s = 'RED' THEN                   {This is similar to a   }
      i := 1                           {CASE statement for a   }
                                       {string expression      }
   ELSE IF s = 'GREEN' THEN
       i := 2

   ELSE IF s = 'BLUE' THEN
       i := 3
END.
```

CASE Statement

Like the IF statement, the CASE statement is used to select a certain action based upon the value of an expression. Instead of the type Boolean, however, the expression may be of any enumeration or subrange type, including Boolean, integer, character, and user-defined enumeration and subrange types.

Syntax:



The expression, called the selector, is used to select which statement is to be executed. Each constant expression in the lists of labels must be compatible with the type of the selector. A label may only appear in one list, and separate ranges may not overlap.

The statement associated with the label list containing the value matching the selector is executed. The statement associated with the OTHERWISE is executed if the selector doesn't match any of the labels. More precisely, when a CASE statement is executed,

1) The selector expression is evaluated.

2) If the value appears in a label list within the CASE statement, the statement associated with that list is executed. Execution then resumes at the statement following the CASE statement.

3) If the value does not appear in any label list, then either:

   a) If OTHERWISE is specified, the statements between the OTHERWISE and the END are executed, and execution resumes at the statement following the CASE statement.

   or b) If OTHERWISE is not specified, an error will occur.

CASE statements may be nested to any level.

Examples:

```
PROCEDURE scanner;
BEGIN
   get_next_char;
   CASE current_char OF

      'a'..'z',
      'A'..'Z':
         scan_word;

      '0'..'9':
         scan_number;

      OTHERWISE
         scan_special;
   END;
END;


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


FUNCTION op
   (operator: OPERATORS  {TYPE OPERATORS=(plus,minus,times,divide)}
    operand1,
    operand2: REAL)
   : REAL;

BEGIN
   CASE operator OF
      plus:   op := operand1 + operand2;
      minus:  op := operand1 - operand2;
      times:  op := operand1 * operand2;
      divide: op := operand1 / operand2;
   END;
END;
```

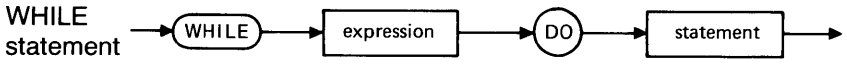
Another example:

```
PROGRAM show_case;
TYPE
   COLOR = (red, yellow, blue, orange, green, purple, none);
   BASICS = red..blue;
   COMPOUND = orange..purple;

VAR
   c: COLOR;

FUNCTION new_color
   (color1,
    color2: COLOR)
   : COLOR;

BEGIN
   CASE color1 OF

      red:
         CASE color2 OF
            red:       new_color := red;
            yellow:    new_color := orange;
            blue:      new_color := purple;
            OTHERWISE  new_color := none;
         END;

      yellow:
         CASE color2 OF
            red:       new_color := orange;
            yellow:    new_color := yellow;
            blue:      new_color := green;
            OTHERWISE  new_color := none;
         END;

      blue:
         CASE color2 OF
            red:       new_color := purple;
            yellow:    new_color := green;
            blue:      new_color := blue;
            OTHERWISE  new_color := none;
         END;

      OTHERWISE  new_color := none;
   END;
END;  {new_color}

BEGIN  {show_case}
   c := new_color (red, yellow);
END.   {show_case}
```

WHILE Statement

The WHILE statement is used to execute a statement repeatedly as long as a given condition is true.

Syntax:



When a WHILE statement is executed, the expression, or "condition", is evaluated, and must result in a Boolean value. Each time the expression evaluates to a true value, the statement is executed and the expression is re-evaluated. When the expression results in a false value, execution is resumed at the statement following the WHILE statement.

The statement:

```
WHILE condition DO statement
```

is equivalent to both of the following:

```
IF condition THEN BEGIN
   statement;
   WHILE condition DO statement
END;
```

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

Partial evaluation is used in evaluating the condition, unless the PARTIAL_EVAL compiler option is turned off.

Note that the statement should at some point modify data such that the condition will evaluate to false. Otherwise the statement will be repeated indefinitely.

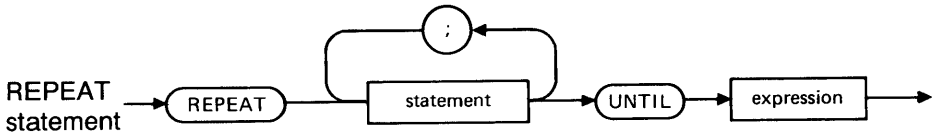
Examples:

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

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

REPEAT Statement

The REPEAT statement is used to execute a group of statements repeatedly until a given condition is true.

Syntax:



When a REPEAT statement executes, the statement sequence is first executed and then the expression is evaluated. Each time the expression is evaluated to a false value, the statement sequence is executed again and the expression is re-evaluated. When the expression results in a true value, execution resumes at the statement following the REPEAT statement.

The statement:

```
REPEAT
   statement;
UNTIL condition
```

is equivalent to both of the following:

```
BEGIN
   statement;
   IF NOT condition THEN BEGIN
      REPEAT
         statement
      UNTIL condition
   END
END;
```

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

Partial evaluation is used in evaluating the expression, unless the PARTIAL_EVAL compiler option is turned off.

Note that the statement should at some point modify data such that the condition will evaluate to a true value. Otherwise the statement sequence will be repeated indefinitely.

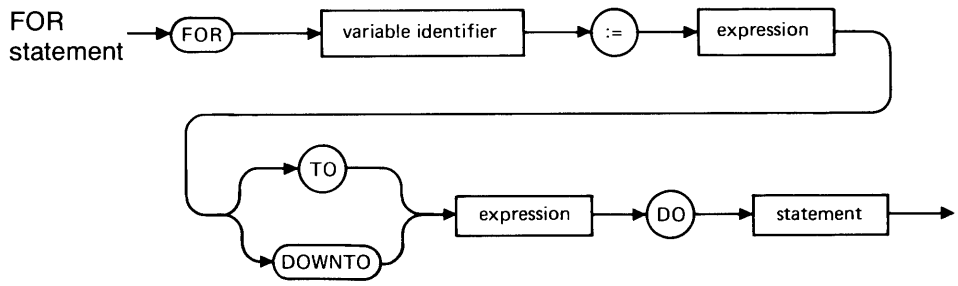
Examples:

```
REPEAT
   read (num_file, value);
   sum := sum + value;
   count := count + 1;
   average := sum / count;
   writeln ('value =', value, '    average =', average)
UNTIL eof (num_file) OR (count >= 100);

REPEAT
   writeln (real_array [index]);
   index := index + 1;
UNTIL index > limit;
```

FOR Statement

The FOR statement is used to execute a statement once for each value in a range, specified by initial and final expressions. A variable, called the "control variable", is assigned each value of the range before the corresponding iteration of the statement.

Syntax:



The control variable must be a local variable, and it also must be an entire variable, meaning it may not be a selected variable (array component, record component, heap variable, or file buffer). In addition, the control variable may be a local formal value parameter, but may not be a formal variable parameter.

Within the FOR loop, the control variable is protected from assignment at compile-time, and may not be passed as a variable parameter. It also may not appear as the control variable for a second FOR loop nested within the first. If the value of the variable is changed by some other means during the execution of the loop, the effect on the number of times the statement is executed is undefined.

The range of values assumed by the control variable is specified by two expressions, the "initial" and "final" expressions, which must be of an assignment compatible type with that of the control variable. These expressions are evaluated only once, before any assignment is made to the control variable. So the statement sequence

```
i := 5;
FOR i := pred (i) TO succ (i) DO writeln ('i=',i:0);
```

```
will write:          instead of:
   i=4                  i=4
   i=5                  i=5
   i=6
```

The statement is not executed if the initial expression is greater than the final (less than the final in the case of a FOR...DOWNTO statement). An assignment is made to the control variable only if the statement is executed. Thus the following statement sequence writes nothing, and leaves i containing the value 5.

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

If the FOR loop is exited using a GOTO statement, the value of the control variable outside the loop is the same as it was before the GOTO statement. The control variable is undefined, however, after a FOR loop is terminated normally.

The FOR statement

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

is equivalent to the statement

```
BEGIN
   temp1 := initial;          {evaluate the two expressions      }
   temp2 := final;            {(not evaluated for each iteration)}

   IF temp1 <= temp2 THEN BEGIN
      control_var := temp1;       {assign only if going thru loop}
      statement;

      WHILE control_var <> temp2 DO BEGIN
         control_var := succ (control_var);  {increment}
         statement;
      END;
   END
   ELSE BEGIN
      {don't go thru the loop at all                          }
      {leave control variable value as it was before loop}
   END;
END
```

The FOR statement

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

is equivalent to the statement

```
BEGIN
   temp1 := initial;    {evaluate the two expressions}
   temp2 := final;      {(not evaluated for each iteration)}


   IF temp1 >= temp2 THEN BEGIN
      control_var := temp1;  {assign only if going thru the loop}
      statement;

      WHILE control_var <> temp2 DO BEGIN
         control_var := pred (control_var);  {decrement}
         statement;
      END;
   END
   ELSE BEGIN
      {don't go thru the loop at all.                    }
      {leave control variable as it was before the loop}
   END;
END
```

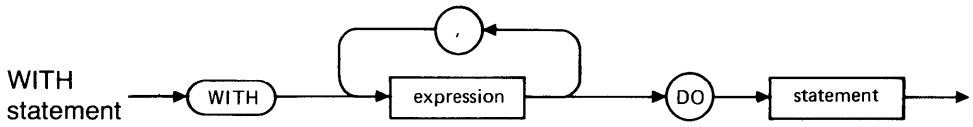
Examples:

```
FOR color := red TO blue DO
   writeln ('Color is ', color_to_string (color));


FOR i := 10 DOWNTO 0 DO
   writeln (i);
writeln ('Blast Off');


FOR i := (a[j] * 15) TO (f(x) DIV 40) DO
   IF odd (i) THEN
      x [i] := cos (i)
   ELSE
      x [i] := sin (i);
```

WITH Statement

The WITH statement is used to allow fields of a record to be accessed without mentioning the record itself.

Syntax:



Each record expression in the list is either a record variable, a record constant, or a reference to a function which returns a record. Within the WITH statement, any field of any of the records in the list may be accessed by using only its field name, instead of the normal field selection notation using the period between the record and the field name. The following statements are equivalent:

```
WITH rec DO BEGIN
   fieldl := el;
   writeln (fieldl*field2);
END;
```

```
BEGIN
   rec.fieldl := el;
   writeln (rec.fieldl
         * rec.field2);
END;
```

The record expressions are evaluated once and only once for the WITH statement. This evaluation occurs before the component statement is executed, so, if f is a field, then the statement sequence

```
i := il;
WITH a[i] DO BEGIN
   writeln (f);
   i:=i2;
   writeln (f)
END;
```

or

```
p := pl;
WITH p^ DO BEGIN
   writeln (f);
   p := p2;
   writeln (f)
END;
```

produces the same effect as:

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

or

```
writeln (pl^.f);
writeln (pl^.f);
```

Records having identical field names may appear in the same WITH statement, with the following interpretation resolving the ambiguity: the statement

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

is equivalent to

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

Thus if field f is a member of both record1 and record2, a reference to f within the statement above would be interpreted as a reference to "record2.f".

Also, this means that if r and f are records, and f is a field of r, then the statement

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

can be written as

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

If a local or global identifier has the same name as a field of a record appearing in a WITH statement, then the appearance of the identifier within the WITH statement is always a reference to the record field, making the local or global identifier inaccessible in the statement.

GOTO Statement

The GOTO statement is used in conjunction with a statement label to transfer control from one part of the program to the statement associated with the label.

Syntax:

The GOTO statement must appear in the same body as the label definition, or in any of the routines which are enclosed by the block containing the label declaration. The latter case is referred to as an "out-of-block" GOTO. Out-of-block GOTO statements should be used with care as normal procedure or function termination is bypassed. This can have two effects which may be undesirable:

1) If the routine is recursive, the current activation is left active. This means that on termination, the stack is not popped, and a subsequent activation of the routine will be a recursive one. So the program will work correctly, but may not be as efficient as it could be since stack space may be exhausted sooner than would normally be the case, and recursion overhead must be paid for every activation of the routine.

2) If the routine has any local files, their automatic closing before return is bypassed. This can be handled explicity by the programmer, if desired, by closing all files prior to the GOTO.

Further, both of these effects apply not only to the routine containing the GOTO, but also to any intermediate routines enclosing that routine which are also enclosed by the block in which the label is defined.

In standard Pascal, GOTO's may not lead into a component statement of a structured statement from outside that statement or from another component statement of that statement. For example, it is illegal to branch to the ELSE part of an IF statement from either the THEN part, or from outside the IF statement. Pascal/1000, however, does not prohibit these GOTO paths, but it is recommended that programs not be written which rely on this fact, as other implementations may be stricter.

GOTO statements may lead from a subprogram or segment unit into the main body of the program.

Example:

```
PROCEDURE show_goto;
LABEL 500, 501;

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

Empty Statement

The empty statement is denoted by no symbol and performs no action. It is often useful for indicating that no action is to be taken.

For example the two statements below

```
CASE i OF

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

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

explicitly specify no action when i contains 2,3,4,6,7,8,9, or 10.

Expressions

An expression is a construct composed of operators and operands, used to compute a value of some type. An operator defines an action to be performed on its operands. An operand may be an arithmetic, Boolean, relational, or set operator, or it may be a reference to a function.

Operands denote the objects which operators will use in obtaining a value, and may be literals, symbolic constants, or variables.

An expression's type is known when it is written and never changes. An expression's value, however, may not be known until the expression is evaluated and may be different for each evaluation.

Operands

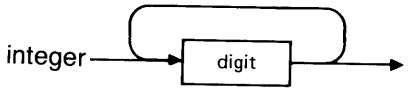
An operand may be acted upon by an operator. An operand is a literal symbolic constant, variable or the value of another expression.

Literals

A literal is a representation of one of the possible values of a certain type. The literal must conform to certain syntax rules for literals of that type. Literals in Pascal may be integer, real, or string literals.

Integer Literals--The usual decimal notation is used for numbers of type INTEGER. Spaces may not appear within an integer literal. Integers can only be represented in decimal notation.

Syntax:

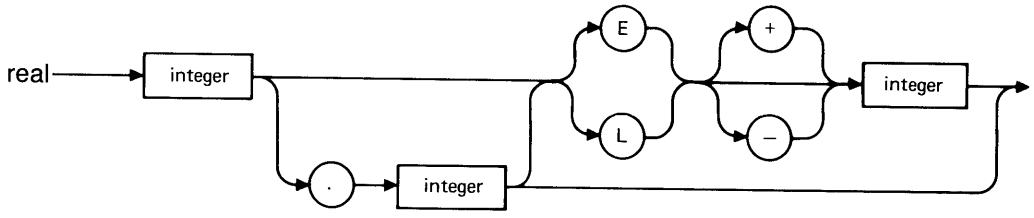


Examples:

```
100   1   2000000000   32768
```

Real Literals--Literals of the types REAL and LONGREAL are represented with decimal digits, a decimal point, and an optional scale factor.

Syntax:



The letter E (L) preceding a scale factor specifies an exponent of the form "times 10 to the power of" and indicates a constant of type REAL (LONGREAL). Lowercase "e" and "l" are legal. Decimal points must be preceded and followed by at least one digit. A number containing a decimal point and no scale factor is of type REAL. Spaces may not appear in numbers.

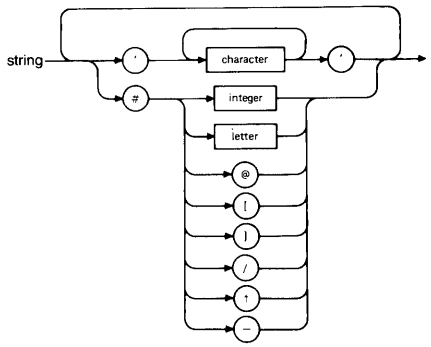
Examples:

```
0.1   5E-3   5L-3   496.28   87.35e+8   87.357535312L+8
```

String Literals--Sequences of characters enclosed by single quote marks are called strings. A string consisting of a single character is a constant of the type CHAR. Strings of two or more characters are constants of the type:

```
PACKED ARRAY [1..n] OF CHAR
```

Syntax:



Printable ASCII characters appear in strings in the normal manner with the exception of the single quote mark (′). If the single quote mark is to be included in a string it must appear twice.

Non-printing ASCII characters may be included in strings by using an extended string syntax employing the sharp sign (#). In this notation, the sharp sign is used to encode an ASCII control character when followed by a non-numeric character, or to encode any character by giving its decimal value (in the range 0 to 255). This notation is a Pascal/1000 extension, and is not allowed in standard Pascal.

Examples:

```
''''          {this represents the single quote character}
'A'
'This is a string'
'Don''t touch this string'
#27'that was an ESC char, and this is also'#[
'this string has five bells'#g#g#g#7#7' in it'
```

Symbolic Constants

A symbolic constant is an identifier that represents a literal, constant expression, or structured constant. It may also represent a component of a structured constant if it appears with the appropriate selector. The identifiers defined in an enumeration type definition are also symbolic constants.

Syntax:

The identifier is associated with a value in the CONST declaration section. This declaration also determines the constant's data type. The constant may be used in places where expressions are expected. It may also be used in TYPE definitions and other CONST definitions. A symbolic constant cannot appear on the left hand side of an assignment statement, as an actual variable parameter, or as a FOR loop control variable.

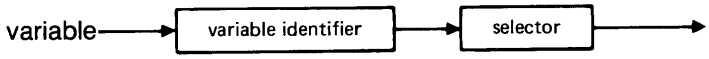
Examples:

```
PROGRAM show_constants (output);

CONST
   medal_name_length = 6;

TYPE
   MEDAL = (gold, silver, bronze);
   MEDAL_NAME = PACKED ARRAY [1..medal_name_length] OF CHAR;
   TRANSLATE = ARRAY [MEDAL] OF MEDAL_NAME;
CONST
   medals = TRANSLATE [MEDAL_NAME ['gold   '],
                       MEDAL_NAME ['silver'],
                       MEDAL_NAME ['bronze']];

VAR
   m: MEDAL;
   medal_table: TRANSLATE;
BEGIN
   medal_table := medals;          {Use an entire constant}
   m := gold;                      {Enumerated constant}

   writeln (medals [gold])         {Use a selected constant}
END.
```

Variables

A variable is an identifier that represents a non-constant, or changeable, data item. Before it is used, it must be declared and associated with a certain data type in the VAR declaration (refer to Declarations in Chapter 4). The variable identifier may denote a simple variable, such as an integer or character, or it may be a structured variable, such as an array or record. In either case, it is called an "entire variable". A variable may also denote a component of a structured variable if it appears with the appropriate selector. Such a variable is called a "component variable" or a "selected variable".

Syntax:



Examples:

Entire variables:

```
i        {simple variable                }
a        {structured (array) variable}
r        {record variable                }
p        {pointer variable               }
fl       {file variable                  }
```

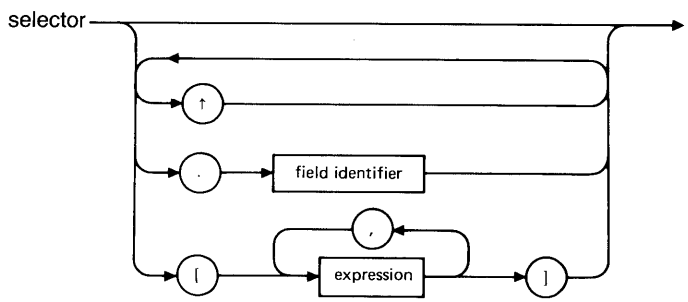
Selected variables:

```
a[i]     {indexed variable       (array component)  }
r.f      {field variable         (record component) }
p^       {referenced variable    (pointer object)   }
fl^      {buffer variable        (file component)   }
m^.n[5]  {indexed field of a referenced record     }
```

Selectors

A selector specifies a particular component of a structured variable. It may be applied to a structured variable or symbolic constant, or to a reference to a function which has a structured return type.

Syntax:



For the examples in this section, assume that the following have been defined.

```
TYPE
   DIM_1_ARRAY = ARRAY [1..10] OF REAL;
   DIM_2_ARRAY = ARRAY [1..20] OF DIM_1_ARRAY;
   SMALL_REC_PTR = ^SMALL_REC;
   SMALL_REC = RECORD
      s,
      t: CHAR;
      u: DIM_1_ARRAY;
   END;
   INT_FILE = FILE OF INTEGER;
   REC_PTR = ^REC;
   REC = RECORD
      f: INTEGER;
      g: SMALL_REC;
      n: DIM_1_ARRAY;
      q: REC_PTR;
      ff: INT_FILE;
   END;
   REC_ARRAY = ARRAY [1..3] OF REC;

CONST
   csr = SMALL_REC [ s: 'A',
                     t: 'B',
                     u: DIM_1_ARRAY [1, 2, 8 OF 3.0]];
VAR
   i: 1..3;
   a: DIM_1_ARRAY;
   aa: DIM_2_ARRAY;
   r: REC;
   f1: INT_FILE;
   p: REC_PTR;
   ra: REC_ARRAY;

FUNCTION func (fp: INTEGER): SMALL_REC; EXTERNAL;
FUNCTION pfunc: SMALL_REC_PTR; EXTERNAL;
```

Array Subscripts

Array components are selected using subscripts, denoted by square brackets ([]) and an expression. The subscript expression must be compatible with the index type appearing in the array's type definition. If the expression is a constant expression, its value is checked at compile time to make sure its value lies in the range specified in the index type. If the expression is non-constant, the value is checked at run time, unless the RANGE compiler option is turned off. The array denotation appearing before the brackets may itself be a selected variable, constant, or function reference.

Examples:

```
a[10]
a[(i*25 MOD 3 + 1)]
aa[1,20]          { These are  }
aa[1][20]         { equivalent }
aa[1]
csr.u[1]
r.n[i]
func(2).u[2]
```

Field Selection

A field of a record is selected by following the record with a period and the name of the field. The record appearing before the period may itself be a selected variable, constant, or function reference. The WITH statement may be used to "open the scope" of the record, making it unnecessary to mention the record when accessing its fields.

Examples:

```
r.f
r.g.s
ra[i].f
csr.s
func(2).s
```

```
WITH ra[i], g DO BEGIN
   f := 5;
   s := 'C';
END;
```

Pointer Dereferencing

A pointer points to, or "references" a variable in the heap. To access this variable, the pointer is followed by the carat (^) character. At run time, unless the RANGE compiler option is turned off, the value of the pointer is checked to make sure it isn't nil before accessing the heap variable. The pointer may itself be a selected variable, or function reference. It may not be a selected constant, as the only pointer constant is nil.

Examples:

```
p^
r.q^
ra[i].q^
ra[i].q^.q^
pfunc^
```

File Buffer Selection

Every file in a program has implicitly associated with it a "buffer variable". This is the variable through which data is passed to or from a file. The file component at the current position of the file can be read into the buffer variable or the next item to be written to the file may be assigned to the variable and then written. The buffer variable, which is of the same type as the file's base type, is denoted by following the file with the carat (^) character. The file appearing before the carat may itself be a selected variable, but may not be a selected constant or a selected function reference.

Examples:

```
f1^
r.ff^
```

Operators

Operators are used within expressions to specify certain actions on one or more operands, and to create a new value. The value is determined by the operator, its operands, and the definition of the effect of the operator. With each operator are associated the following:

1) number, order, and type of operands

2) result type

3) precedence

Operator precedences are used in determining the order of evaluation of elements in an expression.

Precedence	Operators
4	NOT
3	*, /, DIV, MOD, AND
2	+, -, OR
1	<, <=, <>, =, >=, >

A sequence of operators with differing precedences is evaluated such that the higher-precedence operators are evaluated first. Since * has a higher precedence than +, these expressions are evaluated identically:

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

A sequence of operators with equal precedence are evaluated in a "left-associative" manner. For example, these expressions are evaluated identically:

```
(x + y + z)    and    ((x + y) + z)
```

If an operator is commutative, the compiler may choose to evaluate the right operand first in order to produce more efficient code.

The order of evaluation of operators within a parenthesized expression is unaffected by the precedence of any operators outside the parentheses.

Operators may either be predefined or user-defined. Predefined operators are the arithmetic, boolean, set, and relational operators, and the predefined functions. User-defined operators are references to user-written functions, routines that compute and return a value. The value resulting from any operation may in turn be used as an operand for another operator.

The type of each operand is governed by a set of compatibility rules, defined in the Type Compatibility section.

Table 5-1 contains the predefined operators (excluding functions) and their meanings.

Operator:	Meaning:
+	numeric UNARY PLUS and ADDITION; set UNION
-	numeric UNARY MINUS and SUBTRACTION; set DIFFERENCE
*	numeric MULTIPLICATION; set INTERSECTION
/	numeric DIVISION
DIV	integer DIVISION
MOD	integer MODULUS
AND	logical AND
OR	logical INCLUSIVE OR
NOT	logical NEGATION
<	numeric, string, enumeration LESS THAN
<=	numeric, string, enumeration LESS THAN OR EQUAL; set SUBSET
=	numeric, string, enumeration, set, pointer EQUALITY
<>	numeric, string, enumeration, set, pointer INEQUALITY
>=	numeric, string, enumeration GREATER THAN OR EQUAL; set SUPERSET
>	numeric, string, enumeration GREATER THAN
IN	set MEMBERSHIP

Table 5-1. Pascal's Operators

Arithmetic Operators

Pascal defines a set of operators that perform integer and real arithmetic. These operators take numeric operands and produce a numeric result. A numeric type is the type REAL, LONGREAL, INTEGER, or any INTEGER subrange. Each numeric type has a "rank", defined as follows:

Type	Rank
LONGREAL	4
REAL	3
2-word INTEGER	2
1-word INTEGER	1

The rank of the type of the result value of an operator is the same as the highest rank of all the operand types. Operands having types whose ranks are less than the rank of the result type are converted prior to the operation such that they have a type with a rank equal to that of the result type. For example, if i is an INTEGER and x is a REAL in the expression (x + i), then i is converted to REAL before the addition. In short, the two operands to an arithmetic operator must be "expression compatible" (refer to Type Compatibility).

If one operand is of type:	the other operand is of type:	then the result is of type:
1-word INTEGER	2-word INTEGER	2-word INTEGER
INTEGER	REAL	REAL
INTEGER	LONGREAL	LONGREAL
REAL	LONGREAL	LONGREAL

Real division is an exception to this rule. If both operands are INTEGERs, then both are converted to REAL prior to the division.

A unary operator results in a value of the same type as its operand.

Unary + --The result of the unary + operator is the value of its operand. The operand may have any numeric type.

Unary - --The result of the unary - operator is its operand's negated value. The operand may have any numeric type.

Addition (+), Subtraction (-), and Multiplication (*) --The result of these operations is the sum (+), difference (-), or product (*) of the operator's two operands. The operands may have any numeric types.

Real division (/) --The real division operator calculates a value equal to the quotient of its two operands, which may have any numeric types. If both operands are of type INTEGER, then the result is of type REAL.

Integer division (DIV), and Integer modulus (MOD.) --DIV calculates the truncated quotient of two integers. The sign of the result is positive if the operands' signs are the same, and negative otherwise. The MOD operator computes the modulus function of two integers. The sign of the result is the same as the sign of the first operand.

```
i DIV j = trunc (i/j)
i MOD j = i - ((i DIV j) * j)
```

Both operands must be INTEGERs for both DIV and MOD.

Examples:	Result:
5 + 2	7
5 - 2	3
5 * 2	10
5.0 / 2.0	2.5
5 / 2	2.5
5.0L0 / 2	2.5L0
5 DIV 2	+2
5 DIV (-2)	-2
-5 DIV 2	-2
-5 DIV (-2)	+2
5 MOD 2	+1
5 MOD (-2)	+1
-5 MOD 2	-1
-5 MOD (-2)	-1

Boolean Operators

The Boolean operators perform logical functions on Boolean operators and result in a Boolean value.

NOT (logical negation) -- The NOT operator takes one Boolean operand and produces the Boolean result equal to the inverse of the operand.

a	NOT a
T	F
F	T

AND (logical and), OR (logical inclusive or) --The AND (OR) operator is used to perform the logical and (inclusive or) operation on two Boolean operands. The result is a Boolean value defined by the truth table:

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

Boolean expressions are evaluated using either partial or full evaluation, depending on the setting of the PARTIAL_EVAL compiler option. Under partial evaluation, expr2 in

expr1 AND expr2 and expr1 OR expr2

is not evaluated if expr1 is false (true in the second case). This results in more efficient code and in many cases eliminates the need for nested IF statements (refer to IF statement). Not all Pascal compilers do partial evaluation, and programs relying on this feature may not work when compiled with another compiler.

Partial evaluation is performed only for the operators AND and OR. Relational operators with Boolean operands are always fully evaluated.

AND, OR, and NOT cannot be used on operands of non-Boolean types, notably INTEGERs.

Set Operators

Three infix operators are defined in Pascal which manipulate two expressions having compatible set types and result in a third set.

Union (+) --The union operator creates a set whose members are all of those elements present in the first set operand plus those in the second, including members present in both sets.

Difference (-) --The difference operator creates a set whose members are those elements which are members of the first set but are not members of the second set.

Intersection (*) --The intersection operator creates a set whose members are all of those members present in both of its operand sets.

The two operands of a set operator must be expression compatible (refer to Type Compatibility). "Width" is a convenient term for the distance between the lower and upper bounds of a set's base type. Sets with the same width have base types whose lower and upper bounds are identical. One set is wider than a second set if every element in the second can be represented in the first. The set

```
wide: SET OF -100..100
```

is wider than the set

```
narrow: SET OF 1..10
```

The result of a set operation is a set whose lower bound is the minimum of the lower bounds of its two operands, and whose upper bound is the maximum of the two upper bounds. Before the operation is performed, if either operand has a width other than the result's width, it is automatically widened prior to the operation.

Given:

```
VAR                    where  TYPE
   neg: MINUS;                   MINUS = SET OF -10..-1;
   pos: PLUS;                    PLUS  = SET OF 1..10;
   crs: CROSS;                   CROSS = SET OF -5..5;
```

then the result of the expression (neg + pos) is a set whose base type is the range -10..10. The base type of the set resulting from (crs + pos) is the range -5..10.

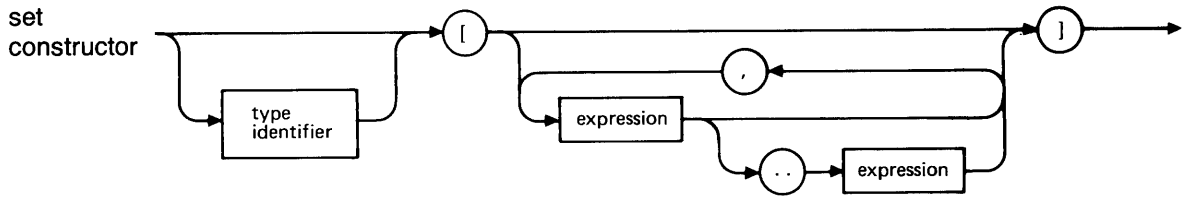
Narrowing of sets is also automatic for set assignments and actual value parameters. Compile- and run-time range checks are performed to verify that the narrowing of a set does not discard any elements. For example, in the assignment

```
crs := crs + pos
```

the result of the union is of type SET OF -5..10 and thus must be narrowed to SET OF -5..5. A run-time error will occur if the result has as members any of the numbers in the range 6..10.

Set Constructor--Another construct, the set constructor, is considered to be an operator and it too creates a set.

Syntax:



Each expression in the constructor is entered into the set. Every element between two expressions may be included by using the range (..) symbol between the two. The type identifier preceding the left bracket is used to specify exactly what type of set is to be created. If it is not supplied, one of three possibilities will occur, depending on the type T of the elements in the set:

1) If T is INTEGER, then the set created is of type

   SET OF 0..255

   Both compile- and run-time checks are performed, if necessary to ensure that specified elements are in this range. Thus the set [25, 0, 255] is legal, but [-1, 256] is not.

2) If T is any other ordinal type, the set created is a set whose base type is the entire ordinal type. The set ['A', 'T'], for example, has the type SET OF CHAR;

3) If the empty set [] is specified the type of the set will be determined from context.

The type identifier, then, is needed to construct integer sets outside the range 0..255. But it is also desirable to specify the type for sets over other subrange types for efficiency reasons. The set UPPER_CASE ['A'..'T'] requires much less storage than the set ['A'..'T'], and has a corresponding savings in manipulation time.

Examples:	Result:	Type of Result:
[1..5] + [5,10]	[1..5,10]	SET OF 0..255
PLUS [1,5] * PLUS [5,10]	PLUS [5]	PLUS
['a'..'z'] - ['a','z']	['b'..'y']	SET OF CHAR

Relational Operators

Relational operators are used to compare two operands and return a Boolean result. The operands may be INTEGERs, REALs, LONGREALs, Booleans, sets, or pointers. Relationals appear between two expressions, which must be compatible, and always result in a value of type Boolean. The relational operators are:

```
<  (less than)
<= (less than or equal)
=  (equal)
<> (not equal)
>= (greater than or equal)
>  (greater than)
IN (set membership)
```

Ordinal Relationals --The relationals that can be used with operands of type INTEGER, BOOLEAN, CHAR, or any enumeration or subrange type, are <, <=, =, <>, >=, and >. These operators carry the normal definition of ordering for numeric types, and CHAR relationals are defined by the ASCII collating sequence. The order of enumerated constants is defined by the order in which the constant identifiers are listed in the type definition. Thus the predefinition of BOOLEAN as

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

means that false < true. An expression having an ordinal type may also appear as the first operand of the IN operator.

Some Boolean functions may be performed using the relational operators with Boolean operands, as shown in the truth tables below: (note that false < true). In particular, <= is the implication operator, = is equivalence, and <> is exclusive or. (T = true, F = false.)

a	b	a<b	a<=b	a=b	a<>b	a>=b	a>b
T	T	F	T	T	F	T	F
T	F	F	F	F	T	T	T
F	T	T	T	F	T	F	F
F	F	F	T	T	F	T	F

Numeric operands are converted if necessary to a type of higher rank using the rules mentioned in the Arithmetic Operators section.

String Relationals --Arrays of characters can be compared using the operators =,<>,<,<=,>, or >=. If either string is shorter than the other, it is padded on the right with blanks before the comparison. Both packed and unpacked strings can be compared. If one string is packed and the other unpacked, the unpacked one is packed prior to the comparison. Thus a string comparison may involve both a blank filling and packing or unpacking operation as well as the actual comparison.

Pointer Relationals --Pointers can only be compared using the relationals = and <>. Two pointers are equal if they point to exactly the same object, and are not equal otherwise. Pointers of any type may be compared to the constant nil. Pointers can only be compared to other pointers, and their two pointer types must have identical base types.

Set Relationals --Two sets can be compared for equality and inequality with = and <>. In addition, the <= operator is used to denote the subset operation, and >= denotes the superset operation. One set is a subset of a second if every element in the first set is also a member of the second set. Also, if this is true, then the second set is said to be a superset of the first. Sets are "widened" if necessary (as described in the Set Operators section) before the relational operation. The < and > operators are not allowed on sets.

The IN operation is used to determine whether or not an element is a member of a set. The second operand has the type SET OF T, and the first operand has an ordinal type compatible with T. To test the negative of the IN operator, the following form is used:

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

Heap variables (dynamic variables in Heap 2 programs) that occupy 1024 or more words cannot be compared using relational operators.

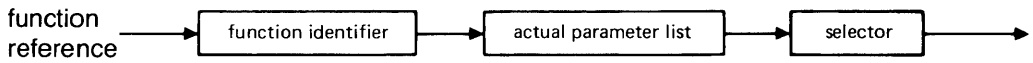
Examples:

```
PROGRAM show_relational;
TYPE
   COLOR = (red, yellow, blue);
VAR
   a,b,c: BOOLEAN;
   p,q: ^BOOLEAN;
   s,t: SET OF COLOR;
   col: COLOR;
BEGIN
   b := 5 > 2;
   b := 5 < 25.0L+1;
   b := a AND (b OR (NOT c AND (b <= a)));
   b := col > red;
   IF (p = q) AND (p <> nil) THEN p^ := a = b;
   b := s <> t;
   b := s <= t;
   b := col IN [yellow, blue];
   b := NOT (red IN s);
END.
```

Function References

A reference to a function can be thought of as an operator, whose operands are the actual parameters passed to the function.

Syntax:



The result, whose type is defined in the function heading, is treated identically to the result of any other operator, and may be used inside an expression. Actual parameters must match the function's formal parameters in number, order, and type (refer to Routine Declarations).

If the function's type is structured, then components of the result value may be accessed using an appropriate selector. Care must be taken to avoid inefficient use of this construct. It is usually better to copy the result of a structured function into a local variable before accessing.

Functions may be recursive.

Example:

```
PROGRAM show_function (INPUT,OUTPUT);

VAR
   n,
   coef,
   answer: INTEGER;


FUNCTION fact (p: INTEGER) : INTEGER;

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


FUNCTION binomial_coef (n, r: INTEGER) : INTEGER;

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


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

Constant Expressions

A constant expression is one that the compiler is able to evaluate at compile time. In the syntax diagrams in Appendix A, every reference to the non-terminal "constant" is calling for a constant expression (these are Pascal/1000 extensions; standard Pascal allows only signed and unsigned literals and constant identifiers). The syntax is no different from ordinary expressions, but there are restrictions on the operators and operands of a constant expression. Allowed in constant expressions are the following:

Operators
- \+ (unary and binary)
- \- (unary and binary)
- *
- /
- DIV
- MOD

Predefined functions:
- pred
- succ
- ord
- chr
- odd
- abs (except for REAL or LONGREAL operands)

Operands
- integer literals
- real and longreal literals
- string literals
- previously-defined constant identifiers

Other operators, such as the relationals, Boolean operators, and other predefined functions are not allowed. Neither are selected constants (e.g., 'table[5]' where table is a structured constant).

Structured constants are not constant expressions, and can only appear in CONST declarations.

Constant expressions are called for in CONST declarations, subrange definitions, the variant part of a field list, structured constants, and case statement label lists.

Examples:

```
CONST
   pi = +3.14159;
   pi_sqr = pi * pi;
   num_symbols = 5;

TYPE
   SYM_ARRAY = ARRAY [1..num_symbols+1] OF CHAR;

CONST
   syms = SYM_ARRAY [succ(2)  OF 'A',
                     abs (-3) OF 'B'];

VAR i: INTEGER;
BEGIN

   CASE i OF

      num_symbols * 2: BEGIN
         ...
      END;

      num_symbols DIV 2: BEGIN
         ...
      END;
   END;
END
```

Type Compatibility

Pascal defines a set of compatibility requirements for the operands of each operator, based both on the operator itself, and the types of its operands.

Relative to each other, two types in Pascal are either

1) identical,
2) compatible,
3) assignment compatible,
4) expression compatible,
or 5) incompatible

Identical Types

Two types are identical if either of the following is true

1) their types have the same type identifier.

2) if T1 and T2 are their two type identifiers, and they have been equivalenced by a definition of the form

```
TYPE T1 = T2
```

Compatible Types

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

1) T1 and T2 are identical types.

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

3) T1 and T2 are set types with compatible base types.

4) T1 and T2 are string types (need not match in length or packing representation - Pascal/1000 extension).

Assignment Compatible Types

T2 is assignment compatible with T1 (that is, a value of type T2 can be assigned to a variable of type T1) if one of the following are true:

1) T1 and T2 are identical types which are not files nor structures that contain files.

2) T1 is REAL and T2 is INTEGER or an INTEGER subrange.

3) T1 is LONGREAL and T2 is INTEGER or an INTEGER subrange.

4) T1 is LONGREAL and T2 is REAL.

5) T1 and T2 are compatible ordinal types and the value of type T2 is in the closed interval specified by the type T1.

6) T1 and T2 are compatible set types and all the members of the value of type T2 are in the closed interval specified by the base type of T1.

7) T1 and T2 are strings and the length of T2 is less than or equal to the length of T1.

For operations which require assignment compatibility, a compile- or run-time error will be produced if either:

a) T1 and T2 are compatible ordinal types and the value of type T2 is not in the closed interval specified by the type T1.

b) T1 and T2 are compatible set types and any member of the value of type T2 is not in the closed interval specified by the base type of the type T1.

For operations which require assignment compatibility, the following implicit conversions are performed prior to the operation:

a) 1-word INTEGER values are converted to 2-word INTEGER values.
b) 2-word INTEGER values are converted to 1-word INTEGER values.
c) INTEGER values are converted to REALs.
d) INTEGER values are converted to LONGREALs.
e) REAL values are converted to LONGREALs.
f) Set values are widened or narrowed to the type T1.
g) Unpacked strings are converted to packed strings, and vice versa, and shorter strings are converted to longer strings and filled on the right with blanks.

Expression Compatibility

Two types T1 and T2 are expression compatible if either of the following are true.

a) T1 is assignment compatible with T2

b) T2 is assignment compatible with T1

Special Cases

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

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

Chapter 6
Files

Although files are declared in the variable declaration section, they are very different from other variables. Their main purpose is to allow the program to communicate with its environment.

Unlike other variables, any information stored in a file by a Pascal/1000 program is located on a peripheral device, not in the program's partition or EMA. This is accomplished by associating the file identifier with a file NAMR, or logical unit number. Several file identifiers may be associated in this way allowing the program to accept input or send output to several file NAMR's or logical unit numbers. This association may be changed when the program is run, as well as during program execution.

There are two major file types; logical and physical files. A logical file is any file named in a Pascal/1000 program. There are various types of logical files; they differ both in the type of their components and in the way their components may be accessed. A physical file is any RTE file, identified by a file NAMR or a logical unit number. To direct normal program input and output to a file or device, the logical files are associated with physical files.

Logical Files

All files used in a Pascal/1000 program are referred to as logical files. Each logical file has a Pascal/1000 identifier associated with it. The logical file structure consists of a sequence of components of the same type. These components may be of a simple type, such as INTEGER, REAL, or CHAR, or they may be of structured types, such as arrays or records. The components cannot be of type FILE or a structured type which contains a file.

The identifiers associated with the file and the component type are declared in a variable declaration section. All files used in the program, except files INPUT and OUTPUT, must be declared before they are used. The files INPUT and OUTPUT are predefined:

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

and can be accessed in any routine or program body if declared in the program heading.

Examples of logical files:

```
TYPE
  string = PACKED ARRAY [1..70] OF CHAR;
  person = RECORD
    name : string;
    age_in_years : 0..120;
    employee_number : 0..5000
  END;
VAR
  people_file : FILE OF person;
  string_file : FILE OF string;
  int_file : FILE OF CHAR;
  num_file : FILE OF INTEGER
```

A logical file is made available or "opened" through the predefined procedures open, reset, rewrite or append. The file must be opened before it can be accessed. (Exceptions are the files INPUT and OUTPUT, which are opened at the beginning of program execution.) The procedure used determines how the file components may be accessed.

Sequential Files

Sequential files are logical files which have been opened through the procedure reset, rewrite, or append. Components in these files must be accessed in sequence. Sequential files may take up less space than direct-access files since they use only the amount of space necessary to store their components. However, access time can be slower. For large files, a great deal of time is wasted accessing unwanted components if the component required is at the end of the file. Once written, a sequential file can be changed by either rewriting the entire file, or by appending new information to it.

Text Files

TEXT files are sequential files which have been previously declared to be of type TEXT.

For example:

```
VAR
  in_file : TEXT;
  out_file : TEXT;
```

More information on Text File Declaration is contained in Chapter 4.

They are similar to sequential files of type CHAR, but are further structured into lines. The file buffer of a text file is of type CHAR.

Direct-Access Files

Direct access files are logical files opened through the procedure open.

The components of direct-access files are accessed differently from the components of sequential files. Some examples:

If a sequential file is in the read-only state, the next component available for reading in the file will be the one directly following the component last made available. They are made available in a sequential order. In direct-access files, however, a component can be made available for reading anywhere within the file, regardless of the position of the component last made available.

If a sequential file is in the write-only state, no component occurring before the current position can be changed. To correct an error, the file would have to be closed, reopened by the function rewrite (which would destroy all contents) and completely rewritten. Direct access files allow single components to be accessed and changed anywhere in the file, regardless of the current position.

A sequential file can be either in the read-only or write-only state, but not in the read-write state. Components of a direct-access file can be read from and written to.

A sequential file has no limit on the number of components. There is a maximum number of components in a direct-access file. The number depends on the size of the components and can be obtained using the maxpos function.

Logical File Characteristics

Every logical file is associated with a file buffer variable, current position pointer, and a state.

File Buffer Variable

The file buffer variable is of the same type as the file's component type. It is denoted:

```
f^
```

where f is the identifier associated with the file.

The file buffer variable is used to access the component to be read from or written to the file.

Once the file has been opened, the file buffer may be accessed by the program as a variable of the file component type.

The contents of a file buffer may be assigned to a variable through the assignment statement. For example:

```
variable_id := file_identifier^
```

would assign the contents of the file buffer to "variable_id". Note: The variable must be of a type which is assignment compatible with file's component type. (Files of the predeclared type TEXT have file buffers variable that are assignment compatible with variables of the predeclared type CHAR.)

Example:

```
TYPE
  book_info = RECORD
    title : PACKED ARRAY [1..50] OF CHAR;
    author : PACKED ARRAY [1..50] OF CHAR;
    number : 1..32000;
    status : (on_shelf,checked_out,lost,ordered)
  END;
VAR
  book : book_info;
  book_file : FILE OF book_info;
    .
    .
    .
  book := book_file^;
```

Current Position Pointer

The current position pointer marks a component of the file. It is used with the file buffer to access components of the file.

Mode or State

At any time, a logical file may be in one of four states; read-only, write-only, read-write, or closed.

Physical Files

A physical file is a disc file or device in the environment external to the Pascal/1000 program. It is identified by an RTE NAMR (which may be a logical unit number). Physical files are accessed in Pascal/1000 by associating this NAMR with a logical file identifier. A Pascal/1000 program makes no distinction between the two kinds of physical files, disc files or devices.

Opening Files

Although a file is recognized by the program after its declaration in a variable declaration section, it cannot be accessed until it has been opened. There are four predefined procedures which can be used to open a file. A file opened by the procedures rewrite, reset, or append, is a sequential file. A file opened by the procedure open is a direct-access file.

Reset

A file opened by the procedure reset(f) can be read by a Pascal/1000 program until the file is closed. It is opened in a read-only state, for sequential access, and no information can be written to it.

If parameter f is omitted, the file INPUT is assumed.

After the procedure reset(f) is called, if the file f is not empty, the current position points to the first component, the file buffer is assigned the value of the first component, and the function eof(f) returns FALSE. If the file is empty, the contents f^ is not defined and eof(f) returns TRUE.

A second parameter may be included in the procedure call. This is a string parameter described in Associating Files Through The String Parameter in this chapter.

The third parameter is also optional and of type string. The string may contain any of the following strings separated by commas.

CCTL - specifies the text file f has carriage control. If f has not been declared as a text file, an error will occur.

NOCCTL - specifies the text file f has no carriage control. If f is not a text file, an error will occur.

SHARED - specifies the file may be open to more than one program.

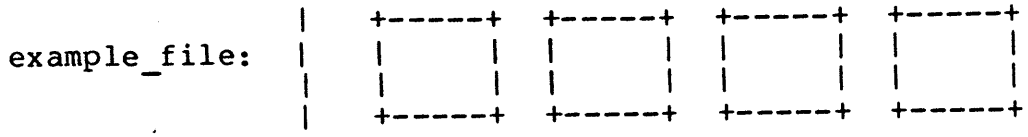
EXCLUS - specifies the file may by open only to one program at a time.

If file f is a text file in the write-only state, the defaults are EXCLUS and CCTL. Otherwise, the defaults are EXCLUS and NOCCTL.

NOTE: The strings CCTL, NOCCTL, SHARED and EXCLUS must be upper case.

A sample file opened by procedure reset(f) is shown below.

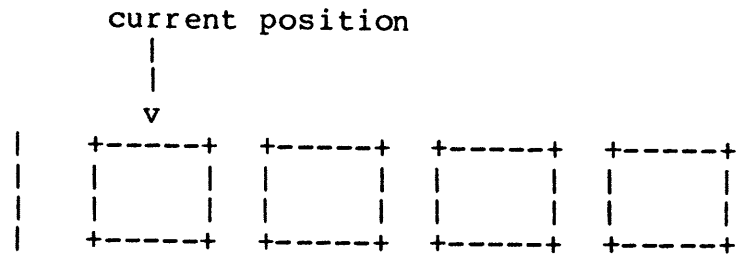
File example_file contains four components. It is in the closed state.



After the call

```
reset(example_file)
```

the file is in the read-only state containing the same number of components, the function eof(f) returns FALSE.



The file buffer, example_file^, will contain the first element.

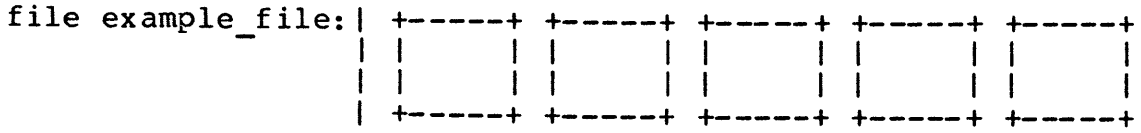
Rewrite

When a file f is opened by procedure rewrite(f), it is in the write-only state. Rewrite(f) discards any previously existing components of the file (the file is then empty) and the current position points to the first component of the file. The content of the file buffer f^ is undefined, and the function eof(f) returns TRUE. A second parameter may be included in the procedure call. This parameter is described in Association Files Through The String Parameter in the Chapter.

If the parameter f is omitted, the file OUTPUT is assumed.

A sample file opened with procedure rewrite is shown below.

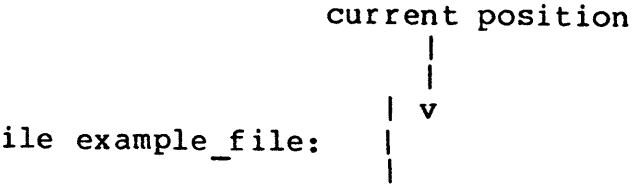
File example_file is a file containing five components. It is in the closed state.



After procedure call

```
rewrite(example_file)
```

the file now is empty (contains no components) and is in the write-only state.



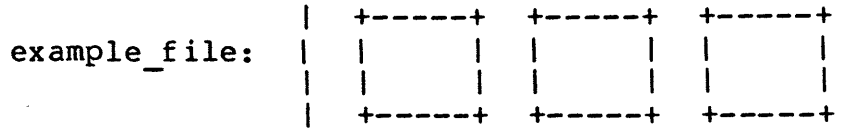
A third parameter may also be included. It is interpreted as it is by reset.

Append

A file opened by the procedure append(f) is in the write-only state. The file opened by procedure append is similar to that opened be procedure rewrite(f). However, the components in this file are not discarded. The content of the file buffer f^ is undefined and the function eof(f) returns TRUE. The current position pointer is moved to just beyond the last component. Anything written to the file at this point will be appended to what previously existed in the file.

A second parameter may be included in the procedure call. This is a a string parameter that will be described in Associating Files Through The String Parameter in this chapter. A sample file opened with procedure append is shown below.

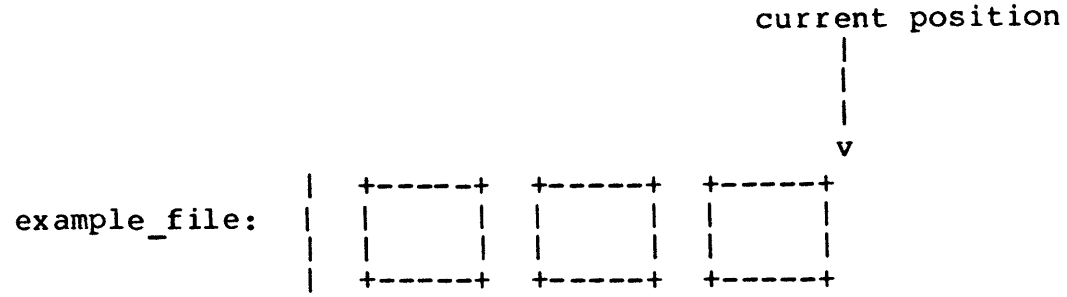
A third parameter may also be included. This parameter is interpreted as it is by reset. File example_file contains three components. It is in the closed state.



After the procedure call

```
append (example_file)
```

the file still has the same number of components and is in the write-only state. It can be written into after the last component. The function eof(f) will return TRUE.



Append is a Pascal/1000 extension.

Open

The procedure open(f) allows the file to be read from and written to. The file is said to be a direct-access file and is in the read-write state. The current-position pointer is at the first component. The file buffer is undefined immediately after the call to open. The function eof(f) returns FALSE.

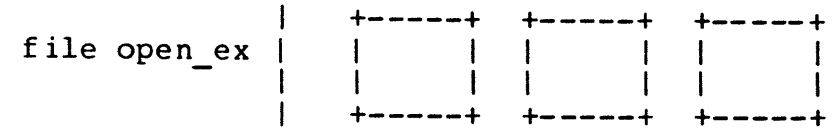
A second parameter may be included in the call to open. In the procedure call,

```
open(f,s)
```

string s is interpreted as described in Associating Files Through The String Parameter in this chapter. A third parameter as described with the procedure reset may also be used. Use of the strings 'CCTL' and 'NOCCTL', however, will cause an error.

Example:

Before the call to open, file open_ex contained three elements.

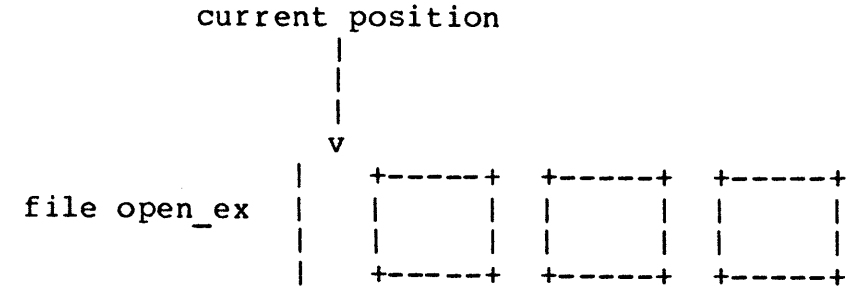


It is in the closed state and its file buffer is undefined.

After the call

```
open (open_ex)
```

the file is in the read-write state, the file buffer remains undefined, and the current-position pointer is moved to the first component.



Open is a Pascal/1000 extension.

Associating Logical and Physical Files

A physical file may be associated with a logical file in one of three ways:

1. If the logical file appears as a parameter in the program heading, an external name is bound to that file when the program is invoked, through the run string.

2. An external name may be supplied as a second parameter to the predeclared procedures append, open, reset and rewrite.

3. If no external name is supplied, a logical file will be bound to a scratch file by the use of append, open, reset, or rewrite.

Associating Files in the RU Command

File identifiers listed in the program heading are program parameters. All such files are associated with physical files at run time through the parameters of the RU command.

For example, a program with the heading:

```
PROGRAM file_example (INPUT, OUTPUT)
```

may be run by the following RU command:

```
:RU,FILE_,1,1
```

If the logical unit number 1 is associated with the user's terminal, the program file_example will accept any input from that terminal and display any output on that terminal.

The RU command allows the programmer to change the association between files listed as program parameters and physical files each time the program is run.

The number of files which can be associated in this way is limited by the maximum length of the RU command, which is 80 characters.

For example, the RU command:

```
RU,MANYF,1,1,1,1,1,1,1,1,1,1,1,1
```

will allow a maximum of twelve formal files to be associated with logical unit number 1.

The correspondence between logical files listed as program parameters and physical files passed through the RU command is positional. The first external name is bound to the first program parameter, the second external name to the second program parameter, and so on, until one or the other of the lists is exhausted.

The binding of a logical file to a physical file is done when a procedure call to append, open, reset, or rewrite is executed. For this reason, if the number of program parameter files exceeds the number of files in the RU command, a run-time error occurs when an attempt is made to open the file. If the number of files in the RU command is greater then the number of program parameter files, no error occurs.

For example, if a program with the heading:

```
PROGRAM example (INPUT, OUTPUT, data_file);
```

is called with the RU command:

```
RU,EXAMP,1,1,DFILE:LW:RL,PFILE:LW:RL
```

no error will occur and the last parameter, PFILE:LW:RL, in the RU string is ignored.

In a program with heading and declarations:

```
PROGRAM testcase (INPUT,file1,file2,file3,file4,file5);
  .
  .
  .
TYPE
  employee=RECORD
    name:ARRAY [1..30] OF CHAR;
    age:INTEGER;
    sex:CHAR;
    department:ARRAY [1..30] OF CHAR;
    END;

VAR
  file1,file2:FILE OF INTEGER;
  file3:TEXT;
  file4:FILE OF LONGREAL;
  file5:FILE OF employee;
  file6:FILE OF CHAR;
```

The files INPUT, file1, file2, file3, file4, and file5 must be associated with external files or logical unit numbers specified in the RU command.

A compile-time error will occur if file OUTPUT is referred to within the program. The file file6 may be associated with a physical file through the use of the string parameter.

NOTE: File identifiers are listed as program parameters before being defined. This is one of two places where identifiers may appear before being declared in the program's variable declaration section.

Associating Files Through the String Parameter

Another method of associating logical and physical files is through the string parameter. This method involves the use of the optional second parameter in the predefined procedures reset, rewrite, open and append.

The second parameter of the procedures reset, rewrite, open and append is a string which names a physical file (NAMR) to be associated with the logical file named by the first parameter. Note: Since the parameter is a string, any single quotes appearing in the file NAMR must appear in the string as two single quotes. Since upper and lower cases are significant in strings, all characters should be upper case in the string parameter to satisfy FMP requirements.

For example:

```
OPEN (direct_access_file, 'STFIL::JA')
```

will open the previously declared logical file, "direct_access_file" and associate that logical file with the physical file STFIL located on the cartridge JA.

```
reset (integer_file,'1')
```

will open the previously declared logical file "integer_file" and associate that logical file with logical unit number 1.

The string parameter allows the user to both specify and later change the association between logical and physical files within a program. If a logical file is opened through a procedure using a second parameter, any previous association between that logical file and a physical file is no longer in effect. The physical file is closed and a new physical file (named by the second parameter) is opened and associated with the logical file.

For example:

```
PROGRAM change_file;
TYPE
  small_integer =0..50;
VAR
  logical_file:FILE OF small_integer;
BEGIN
  reset (logical_file,'FILE1');  {logical_file is now associated}
    .                            {with physical file-FILE1.     }
    .
  reset (logical_file,'FILE2');  {the association between       }
    .                            {logical_file and FILE1 is     }
    .                            {broken, logical_file is now   }
    .                            {associated with a new file    }
    .                            {FILE2.                        }
  rewrite (logical_file,'FILE1'); {the association between      }
    .                             {logical_file and FILE1       }
    .                             {is reestablished.            }
END.
```

Scratch Files

If neither of the two methods for associating logical and physical files is used, the logical file is automatically associated with a scratch file.

This scratch file is placed on the first available cartridge and listed in the cartridge directory. It is given a unique name by Pascal/1000. Scratch file names fall within the ranges of PASC00 to PASC99, and PA0100 to PA9999. The compiler will first attempt to use the file name PASC00. If the name is not unique, the range of names is searched until a unique name is found.

Scratch files exist only during program execution unless the string SAVE is included as a second parameter in the predefined procedure close (see CLOSING FILES).

Example:

from RTE:

```
:RU,SORT,DATA::50,SDATA
```

in Pascal/1000:

```
PROGRAM sort (unsorted, sorted);  { Want to pass 2 files }
                                  { to this program      }
VAR
  scratchl: FILE OF REAL;         { A local file         }
  unsorted,                       { 2 formal files       }
  sorted: TEXT;

BEGIN
  reset (unsorted);               { opens DATA::50       }
  rewrite (sorted);               { opens SDATA          }
  open (scratchl);                { opens scratch file   }
    .
    .                             { associate files with    }
    .                             { specific physical files }
  reset (unsorted, 'NWDATA::GR');
  rewrite (sorted, '1');        { Associate file with terminal }
  open (scratchl, 'SCR1::50'); { opens SCR1::50                 }
    .
    .
    .
END.
```

Relationship Between Logical Files and FMP Files

Brief Summary of FMP File Types

Type	Description
0	A non-disc device.
1	A fixed length, 128 word record, non-extendable, random access file.
2	a fixed length, user defined record length, non-extendable, random access file.
3	A variable length, variable record length, extendable, sequential access file.
4	Same as 3, but usually contains ASCII data.
5	Same as 3, but usually contains relocatable binary code.
6	Same as 3, but usually contains a program in memory image format.
7	Same as 3, but usually conatins absolute binary code.
>7	Same as 3, but contents are user defined.

For more information, refer to the RTE-IVB Programmers Reference Manual.

Sequential Files

Any FMP file type may be accessed as a sequential file, with the exception that the procedure append may not be used with type 1 or 2 files.

If the physical file does not exist, and a file name (instead of an LU number.) is listed as a second parameter to the procedure rewrite or append, the file is created using the type and size specified. If the type is missing, it will default to 3, and if the size if missing, it will default to 24 blocks.

Direct-Access Files

A direct-access file must be a type 1 or 2 file. If the size of the file's component type is 128 words, a type 1 file is used, and a type 2 is used in all other cases. If the procedure open attempts to make available a type other than 1 or 2, by either accessing an existing file or by having a NAMR specified as the second parameter, an error will occur.

If the file does not exist, it is created with the type as specified by the NAMR provided, or if omitted, it is determined by the component size. The size of the file is either specified in the NAMR, or if the size is omitted, it defaults to the number of blocks necessary to contain 1024 records.

Logical file	Type of Physical File	Resultant Type of File
sequential	non-existant	3, with 24 blocks
	logical unit number	
	1 (1)	1
	2 (1)	2
	3	3
	4	4
	5	5
	6	6
	7	7
direct	non-existant	1 or 2, with 1024 records (2)
	logical unit number	error (3)
	1	1
	2	2
	3	error (3)
	4	error (3)
	5	error (3)
	6	error (3)
	7	error (3)

Notes:

1. If used with append, the following error will occur:

   *** PASCAL I/O ERROR ON FILE XXXXX
       FILE CANNOT BE TYPE 1 OR 2

2. If the size of the component type (which is the record length) is 128 words, the type is 1, otherwise, the type is 2.

3. The following error will occur:

   *** PASCAL I/O ERROR ON FILE XXXXX
       FILE MUST BE TYPE 1 OR 2

Interactive File I/O

The following should be taken into consideration when using interactive file I/O:

- If a string that contains an odd number of characters is read from a terminal, an additional blank will be appended to the sting. This is because FMGR will only accept an even number of characters.

- When input is being read from a terminal, a carriage return with no preceding characters will be interpreted as an end-of-file. This can be detected and corrected by using the statement:

  ```
  WHILE eof(f) DO reset(f.);
  ```

  where f is the identifier associated with the file being read.

Sequential File Operations

Once a file has been opened it can be accessed by the program. There are four predefined procedures in Pascal/1000 which can be used to access the components of a file. They are the procedures get(f), put(f), read(f,v) and write(f,v), where f is a file, and v is a variable of a type compatible with the file's components. Manipulation of files accessed using these procedures is described below.

Get(f)

The Procedure get(f) will

1. Advance the current-position pointer one component.

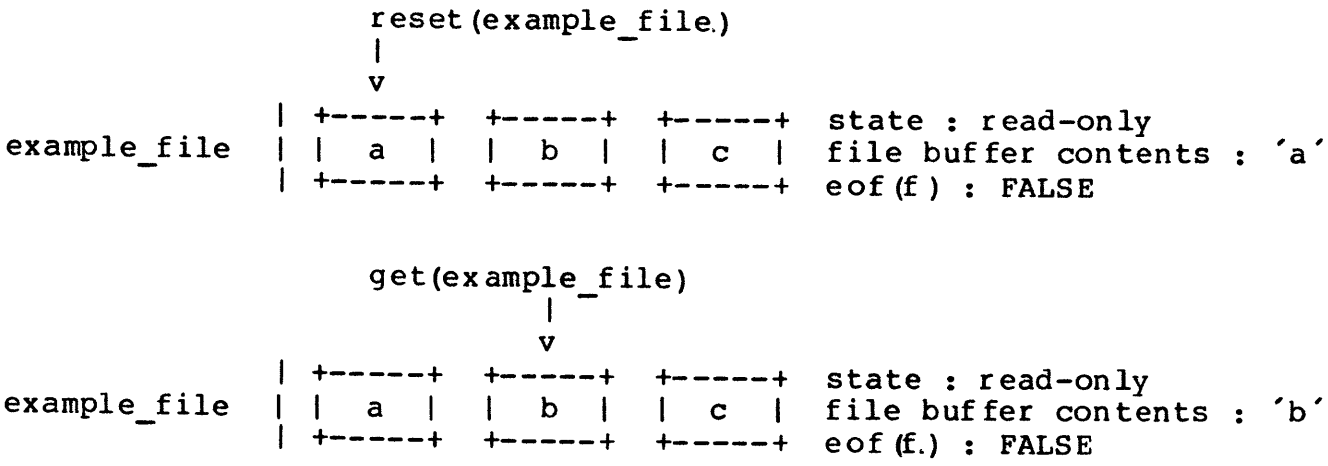
2. Assign that component to the file buffer.

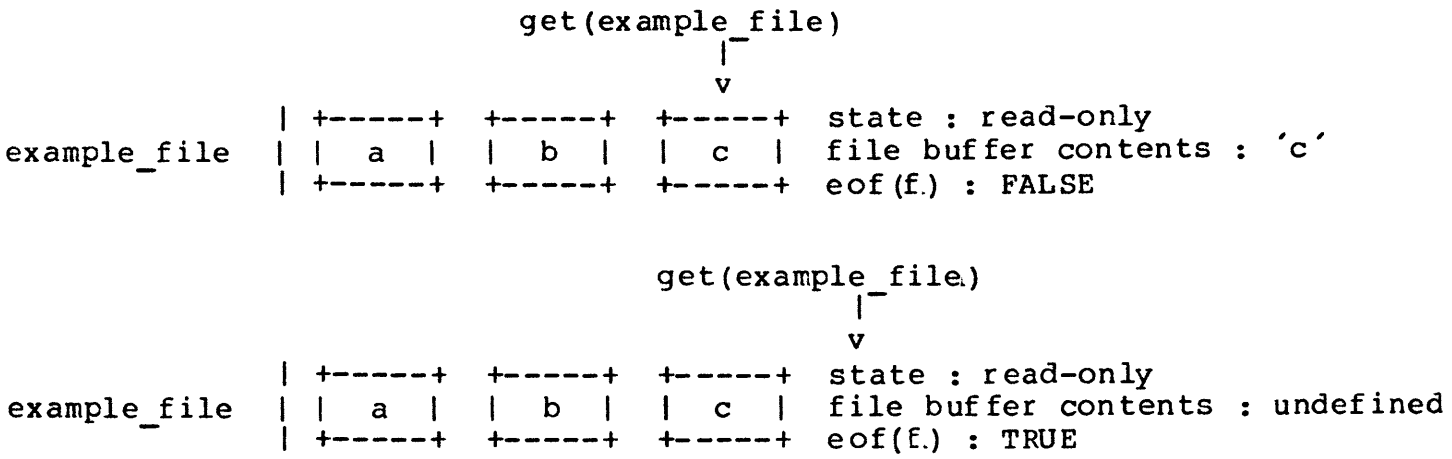
If the component placed in the file buffer is the last component of the file, the function eof(f) will return TRUE the next time get(f) is called. If the file was not opened in the read-only or read/write state, or if the eof(f) was true prior to the procedure call, an error will occur.

For example:

> The file "example_file" is a character file which has previously been filled with the three character components 'a', 'b', and 'c'.

The following figures show the results of a call to the procedure reset, and subsequent get(example_file) procedure calls.





Read (f,v)

For a file f, and a variable v of a type compatible with the type of the file's components, the procedure call

```
read(f,v)
```

will

1) assign the contents of the file buffer to variable v,

2) advance the current-position pointer one component, and

3) assign that component to the file buffer.

Since steps 2 and 3 are the same action performed by the procedure get, the statements,

```
v:=f^;
get(f.)
```

are equivalent to the procedure call

```
read(f,v)
```

If file f is omitted, then the file INPUT is assumed.

As with the procedure get(f), errors will occur if the file was not opened in the read-only or read/write state, or if eof(f) was TRUE prior to the call to read(f,v).

The procedure read(f,v) may contain additional parameters v1,...,vn where v1,...,vn are of a type assignment compatible with the file's components.

The procedure call

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

is equivalent to

```
read(f,v1);
read(f,v2);
  .
  .
  .
read(f,vn)
```

Heap variables 1024 words or larger cannot be parameters for read.

Put(f)

The procedure call

```
put(f)
```

will

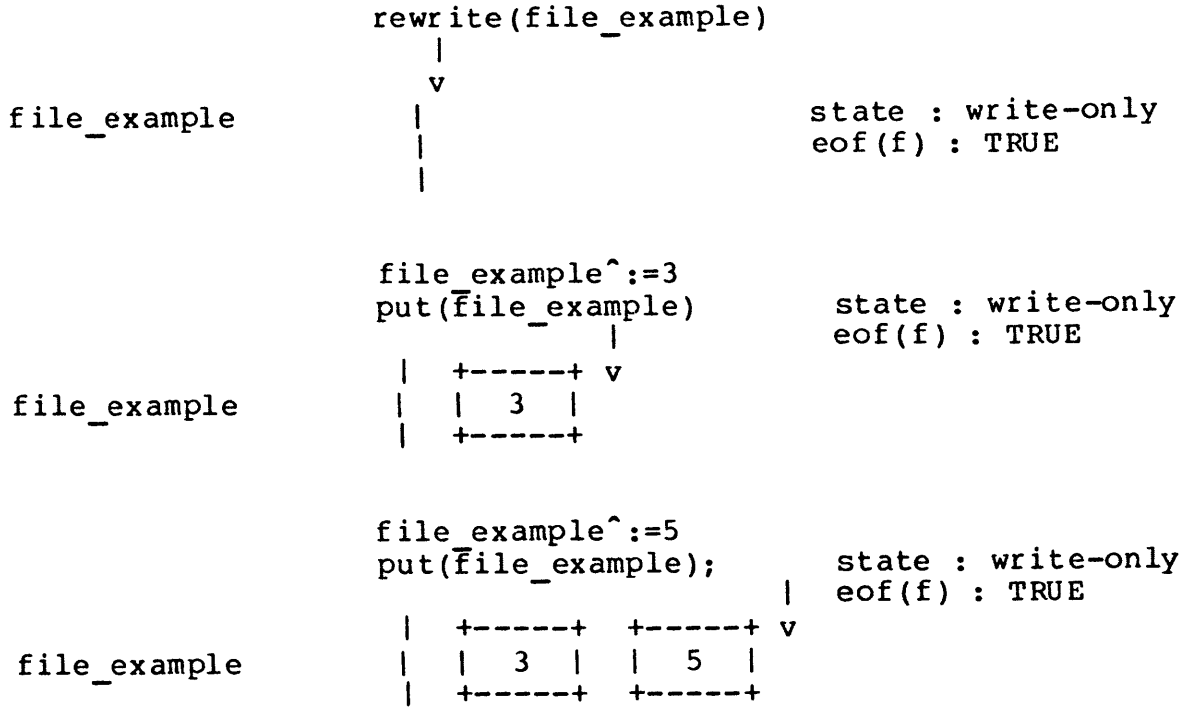
1. assign the contents of the file buffer into the current component of f, and

2. advance the current position pointer to the next component.

Following the procedure call, the contents of the file buffer is undefined. An error occurs if the file was not in a write-only or read/write state prior to the procedure call.

For example:

The file "file_example" is an INTEGER file. The following figures show the results of the procedure call rewrite(file_example) and several calls to the procedure put.



Write (f,v)

For a file f, and a variable v of a type compatible with the components of f, the procedure call

```
write(f,v)
```

will

1) assign the contents of the variable v to the file buffer.

2) assign the contents of the file buffer into the current component of f, and

3) advance the current-position pointer to the next component.

Since steps 2 and 3 perform the same action as the procedure put, the procedure call

```
write(f,v)
```

is equivalent to

```
f^:=v;
put(f);
```

If the file f is not included in the procedure call, then f is assumed to be OUTPUT.

An error will occur if the file was not in a write-only or read/write state prior to the procedure call.

The procedure write(f,v1) may also contain additional parameters, v2,...,vn, of a type assignment compatible with the file's component type. The procedure call

```
write(f,v1,...,vn)
```

is equivalent to

```
write(f,v1);
     .
     .
     .
write(f,vn)
```

Example: The program make_integer_file reads integer values interactively and stores them into integer files.

```
PROGRAM make_integer_file(INPUT,out_file,OUTPUT);

  VAR
    temp_variable:INTEGER;
    out_file:FILE OF INTEGER;

  BEGIN
    rewrite(out_file);
    read(temp_variable);
    WHILE temp_variable > 0 DO
      BEGIN
        write(out_file, temp_variable);
        read(temp_variable);
      END;
    writeln('end condition encountered-file-closed')
  END.
```

The following RU command, and integers when entered from LU 1 will produce a system file 'IDATA' which contains five integer components.

```
RU,MAKE_,1,IDATA,1
8
163
51
4502
3
0
```

Heap variables 1024 words or larger cannot be parameters for write.

Example of sequential file manipulation.

Program compute_mean reads in real inputs from several files. It then computes the mean and puts the results in an answer file.

```
PROGRAM compute_mean(in1,in2,in3,out);

  VAR
    in1,in2,in3,out : FILE OF REAL;
    v1,v2,v3,mean : REAL;

  BEGIN                                    { compute_mean     }
                                           { Open each file.  }
    reset(in1); reset(in2); reset(in3);
    rewrite(out);
                    { Read the first number from each file.}
    read(in1,v1); read(in2,v2); read(in3,v3);
    WHILE NOT eof(in1) AND NOT eof(in2) AND NOT eof(in3) DO
      BEGIN
                                { Compute and write the mean.}
        mean := (v1+v2+v3)/3;
        write(out,mean);
                    { Read the next number from each file.}
        read(in1,v1); read(in2,v2); read(in3,v3);
      END;
  END.                                     { compute_mean     }
```

Text File Operations

Text files may be used as parameters with any of the procedures used to access non-text sequential files: get(f.), put(f.), read(f) and write(f). In addition, several standard procedures can be used exclusively with text files. These are readln, writeln, page, prompt and overprint, as well as several functions, eoln and maxpos. Manipulation of text files by these procedures are described below.

Get(f) & Put(f) With Text Files

The procedures get(f.) and put(f) will perform similar actions on text and non-text files. However, additional caution must be used when using text files. In addition to testing for the eof(f) status, the end-of-line status must be tested to insure the content of the file buffer is meaningful. This can be done by the Boolean function eoln(f) which returns TRUE when the end-of-line is reached.

Read(f,v) With Text Files

Although the text files contain only components of type CHAR, the variable v, in the procedure call read(f,v) may be of type INTEGER, REAL, LONGREAL, strings, a subrange of integer, or CHAR. This is because the procedure does an implicit conversion from the ASCII form which appears in the text file to the actual form stored in the variable.

If v is a variable of type CHAR, the procedure call

```
read(f,v)
```

is equivalent to

```
v :=f^;
get(f);
```

If a read is performed when the end-of-line has been reached, the blank character ' ' will be stored into the character variable.

If variables of types REAL, LONGREAL, INTEGER, or INTEGER subrange, are included as parameters in the procedure read, the file will be searched for characters which satisfy the syntax of these variables. For example, if the variable is of type integer, an attempt will be made to find a sequence of digits which form a number within the range of integers. If the sequence of digits is not found, or if the sequence of digits form a number which is not contained in the range of integers, an error will occur.

Any preceding blanks or end-of-lines are skipped.

After the procedure call, the file buffer will contain the next character immediately following the characters read.

The following table shows the results of the procedure call

```
read(file_one, variable)
```

on several sequences of characters.

sequence of characters contained in file_one following current position pointer	variable type	result stored in variable
(space)(space)1.850	REAL	1.850
(space)(linemarker)(space)1.850	LONGREAL	1.850
10000(space)10	INTEGER	10000
8135(end-of-line)	INTEGER	8135
54(end-of-line)36	INTEGER	54
1.583E7	REAL	1.583x10(7)
1.583E+7	LONGREAL	1.583x10(7)

Table 6-1. Results of the Procedure Read(file_one,variable).

Variables of Type String

If a variable of type string is included as a parameter in the procedure read, the number of characters needed to fill the variable are read.

Any preceding end-of-lines are skipped.

If an end-of-line is encountered before the string is filled, the remaining characters in the string are filled with blanks.

The following table shows the result of the procedure call

```
read(file_two,str_variable)
```

where str_variable has been previously defined to be of type

```
str_variable : PACKED ARRAY [1..5] OF CHAR
```

sequence of characters in file_two following current position pointer	result stored in str_variable
(space)Pascal/1000	' Pasc'
(space)Pas(end-of-line)cal/1000	' Pas '
(end-of-line)Pascal/1000	'Pasca'

Table 6-2. Results of the Procedure Read(file_two,str_variable).

Additional parameters

The procedure read(f) may contain several variable parameters.

The procedure call

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

is equivalent to

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

Write(f,v) With Text Files

When used with text files, the procedure write(f,v) uses the variable identifiers occurring as "write parameters". The program output can be formatted through the use of these write parameters to display program results in a more readable format.

Write parameters have three different forms

1. variable_identifier
2. variable_identifier:m
3. variable_identifier:m:n

where m and n are field-width parameters.

If no formatting is desired, the first form is used. The variable may be a number, character, Boolean value or string. The field-width parameter m in this case will be defaulted depending upon the type of the variable. Table 6-3 shows the default values.

Table 6-3. Field-Width Parameter Default Values

PARAMETER TYPE	DEFAULT FIELD WIDTH "M"
CHAR	1
string	length of string
INTEGER	12
REAL	13
LONGREAL	23
BOOLEAN	5

Examples

The statements

```
int_var:=20;
write('This string contains');
write(int_var);
write(' characters');
```

will produce output of the form,

```
This string contains          20 characters
```

The statement

```
FOR char_var:='a' to 'k' DO
  write(char_var);
```

will produce output of the form,

```
abcdefghijk
```

If formatting is desired, the write parameters can be used to adjust the space in which a variable is written. For variables of type INTEGER, CHAR, or string, only the first field-width parameter can be specified. The second form of the write parameter is used.

The field-width parameter m is an integer specifying the number of characters which will be used to represent the variable in the text file. If m is greater than the number of characters actually needed, the additional characters will be represented as blanks preceding the variable value. If m is less than the number of characters actually needed, the field-width parameter is ignored, and the necessary number of characters is assumed, except for strings and Booleans, where only the first m characters are printed.

Both field-width parameters m and n can be used to format real variables. If the parameter n is present, a fixed-point representation with n digits after the decimal point is obtained. If n is 0 the decimal point will be omitted. Under no circumstances will more significant digits be written than are contained in the internal representation. If n is less than the number of significant digits in the internal representation, the number will be rounded off. When the field-width parameter n is missing, a floating point representation consisting of a coefficient and scale factor will be chosen.

Several adjustments may be made by the compiler if the write parameter is specified to be written in more characters than are left on the output line. If the field width is less than the maximum line length, the field will be moved to a new line, otherwise, the field will be divided. A run-time warning will be issued if either of these adjustments occurs.

When the file name is omitted from the write statement, the previously defined file OUTPUT is assumed.

The write statement may contain more than one write parameter. In this case

```
write(f,p1,...,pn);
```

is equivalent to

```
write(f,p1);
  .
  .
  .
write(f,pn);
```

Example:

The following procedures print the results of a test listing output in a tabular format.

```
PROCEDURE head;
TYPE
  pretty_array = PACKED ARRAY [1..66] OF CHAR;
CONST
  array_of_stars = pretty_array [66 OF '*'];
  header = pretty_array [20 of '*','RESULTS TEST #XR107',27 OF '*'];
BEGIN
  writeln(array_of_stars);
  writeln(header);
  writeln(array_of_stars);
  writeln;
  writeln(array_of_stars);
  write('*   TIME    *  NUMBER OF       ');
  writeln('*  RESULTS WITHIN *  DATA RECEIVED *');
  write('*           *  INTERRUPTS      ');
  writeln('*  RANGE          *  FROM PT # 55  *');
  writeln(array_of_stars);
END;
PROCEDURE out_result(str : string; real_n : REAL;
          int_num : INTEGER; range_test : BOOLEAN);
BEGIN
  write(str:9,'*':4,int_num:7,'*':11,range_test:7);
  writeln('*':11,real_n:10:5);
END;
```

Sample output:

```
**********************************************************************
*********************RESULTS TEST #XR107**************************
**********************************************************************

**********************************************************************
*   TIME    *  NUMBER OF      *  RESULTS WITHIN *  DATA RECEIVED *
*           *  INTERRUPTS     *  RANGE          *  FROM PT # 55  *
**********************************************************************
    12:00   *      1          *  TRUE           *   1.20000
    12:01   *      3          *  TRUE           *   1.25000
    12:02   *      6          *  TRUE           *   1.32000
    12:03   *     12          *  TRUE           *   1.22100
    12:04   *      2          *  TRUE           *   1.21200
    12:05   *      5          *  FALSE          *   1.23300
    12:06   *     12          *  TRUE           *   1.22200
    12:07   *     13          *  FALSE          *   1.21200
    12:08   *      5          *  TRUE           *   1.11100
    12:09   *     13          *  TRUE           *   1.23200
```

Readln(f,v)

The parameter list for procedure readln(f,v.) is similar for that of procedure read. If the file identifier parameter is missing, the predefined file INPUT is assumed. Several variable parameters of the types REAL, LONGREAL, INTEGER (or a subrange of INTEGER), or string can be used.

However, read and readln differ in their use of the end-of-line. After a call to readln, the current position pointer is positioned after the end-of-line.

The procedure fills its variable parameters and then skips to the next line regardless of what remains on the line.

Readln is often used with only a file identifier parameter, or with no parameters at all. This will ignore anything remaining on the current line and skip to the next line of input.

Example:

The logical file 'LWIN2' has already been created and contains five lines of text.

LWIN2 contains:

```
1.555565 HELLO THERE  564  234   56  1
    .345.678 GREETINGS       3
23.789PASCAL/1000   4
2.5 BLAISE  32767      THIS IS NEVER READ
2.3 XXXXXXXXXXX9999NEITHER IS THIS
```

The following program reads the lines contained in the text file but accepts only the values it needs through the use of readln.

```
PROGRAM READLN_EXAMPLE(OUTPUT);

TYPE
  string = PACKED ARRAY [1..12] OF CHAR;
VAR
  real_var : REAL;
  str_var : string;
  int_var,i : INTEGER;
  t_file : TEXT;

BEGIN
  reset(t_file,'LWIN2');
  FOR i := 1 TO 5 DO
    BEGIN
      writeln('new line being read');
      readln(t_file,real_var,str_var,int_var);
      writeln('real_var is now ',real_var);
      writeln('str_var is now ',str_var);
      writeln('int_var is now ',int_var);
      writeln;
    END;
END.
```

The results of this program are:

```
new line being read
real_var is now        1.555565E+00
str_var is now  HELLO THERE
int_var is now          564

new line being read
real_var is now        3.456780E+02
str_var is now  GREETINGS
int_var is now            3

new line being read
real_var is now        2.378900E+01
str_var is now PASCAL/1000
int_var is now            4

new line being read
real_var is now        2.500000E+00
str_var is now  BLAISE  327
int_var is now           67

new line being read
real_var is now        2.300000E+00
str_var is now  XXXXXXXXXXX
int_var is now          999
```

Writeln(f,v)

The procedure writeln is the same as the procedure write except that it places an end-of-line immediately after writing the values of its write parameters. As with the procedure write, parameters in writeln may be of type INTEGER, REAL, LONGREAL, BOOLEAN, string or a subrange of INTEGER.

The procedure is often used without write parameters. This adds a blank line to the output and may be used to separate lines of output.

The following procedure illustrates the use of writeln.

```
PROCEDURE display_result(book_title:string);
  BEGIN
    writeln;
    writeln(star_array);
    writeln;
    write('The book you have requested, ');
    writeln(book_title);
    CASE book_status OF
      checked_out:
        BEGIN
          writeln('is now checked out.');
          write('Please check with the librarian');
          writeln(' at the main desk if you wish');
          writeln('to have it reserved.');
        END;
      on_shelf:
        BEGIN
          write('should be on the shelf. Please');
          writeln(' ask the librarian at the');
          write('information desk to help you ');
          writeln('locate the book if you cannot');
          writeln('find it.');
        END;
      missing:
        BEGIN
          write('is currently missing. ');
          writeln('The librarian at the main');
          write('desk may be able to give you');
          writeln(' more information on its status.');
        END;
      lost:
        BEGIN
          write('has been lost. Please check');
          writeln(' with the librarian in room');
          write('124 to see if it has been');
          writeln(' reordered');
        END;
  END{case};
END;
```

Results from the preceding procedure:

```
**************************************************************

The book you have requested, Programming in Pascal
is now checked out.
Please check with the librarian at the main desk if you wish
to have it reserved.

**************************************************************

The book you have requested, The Life of B. Pascal
should be on the shelf.  Please ask the librarian at the
information desk to help you locate the book if you cannot
find it.

**************************************************************

The book you have requested, Programming With Style
is currently missing.  The librarian at the main
desk may be able to give you more information on its status.

**************************************************************

The book you have requested, Everyman's PL1
has been lost.  Please check with the librarian in room
124 to see if it has been reordered.

**************************************************************
```

The following procedures are similar to writeln but have special effects which help in the production of particular types of output.

Page(f)

The procedure page(f) will cause the next element written to text file f to appear at the top of the next page. Page causes the line printer to skip to the top of form so it will only effect a printed listing. If the file f is not a text file an error will occur. If the procedure call contains no parameters, the predefined file OUTPUT is assumed. If CCTL is not in effect, an error will occur.

Prompt(f,vl,...,vn)

The procedure prompt(f) writes the contents of any file buffer associated with text file f to be written to the file. It can be thought of as a procedure which flushes the buffer. Unlike writeln, no end-of-line is written to the file after the buffer contents. Prompt will only be meaningful if the listing appears on the terminal. If the procedure call contains no file parameter f the predefined file OUTPUT is assumed. The parameters vl,...,vn are interpreted as for writeln. Prompt is a Pascal/1000 extension.

Overprint(f,vl,...,vn)

The procedure overprint(f) will cause a carriage return, but will inhibit the line feed. Thus, if the text file is being printed and the end-of-line is encountered, the line following the end-of-line will be printed over the line preceding it. As with page, overprint will only be meaningful if the listing appears on the line printer. If the procedure call contains no file parameter, the predefined file OUTPUT is assumed. The parameters vl,...,vn are interpreted as for writeln. If CCTL is not in effect, an error will occur.

Overprint is a Pascal/1000 extension.

Linepos(f)

Function linepos(f) returns the number of characters read from, or written to, the file since the last end-of-line. The component currently in the file buffer is not included in this count.

This function is helpful with files whose data is in a fixed format as in the example below.

File ifile contains information of the form

```
line n {<employee name>  <sex>  <age>  <social security number>
         .               (male   .      .
         .                 or    .      .
         .               female.) .      .
         :               :       :      :
        col 0            col 30  col 40 col 45
```

A file is needed whose lines have the form:

```
        <employee name>  <social security number>
         .               .
         .               .
         :               :
         col 0           col 30
```

The following statements will create a physical file LWXXX with the desired form,

```
        .
        .
        .
reset(ifile);
rewrite(ofile,'LWXXX');
read(ifile,read_array);          {transfer the name       }
write(ofile,read_array);
WHILE (linepos(ifile) < 45) DO {position to 45th column}
  get(ifile);
read(ifile,read_array);          {transfer the ss number }
write(ofile,read_array);
```

where "ifile" and "ofile" are text files and read_array is a 30 character string.

Linepos is a Pascal/1000 extension.

Eoln(f)

Function eoln returns a Boolean value indicating whether the end-of-line status for file f is TRUE or FALSE. It is important to remember that the end-of-line status will not become TRUE until an attempt is made to access the file beyond the last component.

Therefore, if the end-of-line status is TRUE before the statement

```
read (file_identifier,variable);
```

and variable is of type char, then the undefined contents of the file buffer will be stored into variable. For information about the effects when the variable is of any other type, refer to Read(f,v) With Text Files in this chapter.

If the parameter f is omitted the file INPUT will be assumed.

Direct Access File Operations

In addition to the procedures already defined for non-text sequential files (get, put, read, write) three additional procedures (seek, readdir, and writedir.) and several additional functions (position and maxpos) can be used to access direct-access files.

The following procedures and functions are all Pascal/1000 extensions.

Seek(f)

An important property of direct-access files is the ability to randomly move the current-position pointer. This means any component of the file can be accessed without sequentially arriving at its position in the file. The current-position pointer is moved through the use of the procedure seek. The call:

```
seek(f,k)
```

positions the current-position pointer at component k. File f must have been previously opened as a direct-access file through the use of the file procedure open.

No error will occur if the parameter k is a number greater than the number of the last available component.

Readdir and Writedir

These procedures give the user the ability to seek to a particular component of a direct-access file and access that component using one procedure statement.

Readdir is a combination of procedures seek and read.

The procedure statement:

```
readdir(f,k,v)
```

is equivalent to:

```
seek(f,k);
read(f,v)
```

Writedir is a combination of procedures seek and write.

The procedure statement:

```
writedir(f,k,v)
```

is equivalent to:

```
seek(f,k);
write(f,v)
```

More information on these procedures is contained in Chapter 7.

Position and Maxpos

These functions can only be used with direct-access files. Position(f) returns the integer position of the component pointed to by the current-position pointer. Immediately after the statement:

```
seek(f,k)
```

function position(f) will return the value of k.

Function maxpos(f) returns the last available component in file f. When the result of function position(f) is greater than the result of function maxpos(f), eof(f) will return TRUE.

These functions are explained in greater detail in Chapter 7.

Closing Files

A file is closed by the procedure close. The file will be in the closed state and any attempt to access it will produce an error.

Opening a file will implicitly close any physical file previously associated with that logical file.

If any physical file which was not a scratch file was associated with a logical file the procedure close will save the physical file unless the file is purged through the string parameter 'PURGE'. The procedure close(file_name, 'PURGE') will purge any physical file associated by the program with the file "file_name" at the time of the call.

If the logical file was associated with a scratch file, that scratch file will be purged unless the string parameter 'SAVE' is included as a second parameter.

The procedure close is a Pascal/1000 extension.

Brief Summary of Procedures and Functions

The procedures and functions described in this chapter are often restricted in the type of logical files they may be used with. Table 6-4 summarizes these restrictions.

Table 6-4. Procedure and Function File Restrictions

	Sequential		
	text	non-text	random access
put	x	x	x
get	x	x	x
reset	x	x	
rewrite	x	x	
append	x	x	
close	x	x	x
open			x
seek			x
eof	x	x	x
eoln	x		
linepos	x		
pos			x
maxpos			x
read	x	x	x
readln	x		
readdir			x
write	x	x	x
writeln	x		
writedir			x
page	x		
prompt	x		
overprint	x		

Chapter 7
Standard Procedures and Functions

File-Handling Procedures

The following procedures are used to manipulate files from a Pascal/1000 program. They are explained in greater detail in Chapter 6.

Append

Usage:
```
append(f)
append(f,s1);
append(f,s1,s2);
```

Description

The procedure opens the file as a sequential file in the write-only state. Any components previously existing in the file remain, and the current-position pointer is positioned directly after the last component of the file. The content of the file buffer is undefined. The function eof(f) will return TRUE.

Parameter Types

The parameter f must be a file which has been previously declared. It need not be closed before the call is made. If the file was open before the procedure call, it is automatically closed and reopened in the write-only state.

The string parameters s1 and s2 are explained in Chapter 6.

Close

Usage: close(f)
close(f,s)

Description

The procedure makes the file unavailable for accessing. Any association with a system file is dropped. The content of the file buffer is undefined. The function eof(f) will return TRUE.

Parameter Types

The parameter f must be a file which has been previously declared. It need not be open before the call is made. If the file was closed before the procedure call, no error will be produced.

The second parameter is optional and is of type string. This second parameter can take on one of two values, SAVE or PURGE. If the string SAVE is used, the file is closed and saved as a permanent file in the system. If the string PURGE is used, the file is destroyed as it is closed.

If the file was associated with a system file through the program header, or through the use of the string parameter s, the string defaults to SAVE. Otherwise, the default is PURGE.

Get

Usage: get(f)

Description

The procedure advances the current file position, and assigns the current component to the file buffer. If the component does not exist, the content of the file buffer is undefined and eof(f) will return TRUE. If eof(f) was true before the call, an error will occur.

Parameter Types

The parameter f must be a file which has previously been opened in the read-only or read-write state.

Open

Usage: open(f)
open(f,s1)
open(f,s1,s2)

Description

The procedure open makes the file available in the read-write state. A file which has been opened through the procedure open is referred to as a direct-access file. The current-position pointer is positioned at the first component of the file.

Parameter Types

The parameter f must be a file which has been previously declared. If the file was open before the procedure call no error will be produced, and the file will be opened as a direct-access file.

The second and third parameters are explained in Chapter 6.

Overprint

Usage: overprint
overprint(f)
overprint(f,v1,...,vn)
overprint(v1,...,vn)

Description

The procedure overprint causes the next line to print over the current one when text file f is printed. NOTE: This will only affect printed copies.

Parameters

The file f, if specified, must have been previously declared as a text file, and opened as a sequential file. The file must have carriage control at the time the procedure call is made.

If the parameter f is omitted, the file OUTPUT is assumed.

The optional parameters v1,...,vn are the same as those for the procedure writeln.

Page

Usage: page
page(f)

Description

The procedure page causes skipping to the top of a new page when the text file f is printed. The number one is placed in the first column of a line and is then recognized by DVR00 and DVR05. NOTE: This procedure will only have effect on printed copies.

Parameters

The file f must have been previously declared as a text file, and opened as a sequential file. The file must have carriage control at the time the procedure call is made.

If the parameter f is omitted, the file OUTPUT will be assumed.

Prompt

Usage: prompt
prompt(f)
prompt(f,v1,...,vn)
prompt(v1,...,vn)

Description

The procedure prompt causes the cursor to remain on the same line as the line that has just been written to the text file f.

Parameters

The file f must have been previously declared as a text file, and opened as a sequential file. The file must have carriage control at the time the procedure call is made.

If the parameter f is omitted, the file OUTPUT is assumed.

The parameters v1,...,vn are the same as those for the procedure writeln.

Put

Usage: put(f)

Description

The procedure put writes the value of the buffer variable f^ to the current component of f and advances to the next component. Following the call, the content of the file buffer is undefined.

Parameter Types

The parameter must be a file which has previously been opened in the write-only or read-write state.

Read

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

Description

The procedure read accepts input from a file which has previously been opened in a read-only state. The input is then assigned to variables specified as parameters in the procedure call.

Parameter Types

The parameter f must be a file which has been previously declared and opened in a read-only state. If file identifier f is not included as a parameter, the file INPUT is assumed.

The procedure call:

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

is equivalent to the procedure calls:

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

where the variables v1 through vn are as explained in Chapter 6.

Heap variables 1024 words or larger cannot be parameters for read, readdir, or readln.

Readdir

Usage: readdir(f,k,v)
readdir(f,k,vl,...,vn)

Description

The procedure readdir is used to perform a read operation on a particular component of a direct-access file.

The procedure statement:

```
readdir(f,k,v)
```

is equivalent to the statements:

```
seek(f,k);
read(f,v)
```

and the procedure statement:

```
readdir(f,k,vl,...,vn)
```

is equivalent to the statements:

```
seek(f,k);
read(f,vl,...,vn)
```

Parameters

The parameter f must be a file which has been previously declared and opened as a direct-access file by the procedure open.

The parameter k must be a positive integer. If it is greater than the position of the last available component of the file, the function eof(f) will return TRUE.

The variable parameters must be of a type which is assignment-compatible with the file's component type.

Readln

Usage:
```
readln(v1)
readln(v1,v2,...vn)
readln(f,v1)
readln(f,v1,v2,...,vn)
```

Description

The procedure readln is used to read and then skip to the next line. It is similar to the procedure read used with text files in that input is received from the file and assigned to the variable parameter(s). However, once this action has been completed, the procedure readln will ignore any remaining characters on the line and the next access to the file will begin on the following line.

Parameters

The parameter f must be a file which has been previously declared as a text file and opened in the read-only state. If file identifier f is not included as a parameter, the file INPUT is assumed.

The variables v1 through vn may be of type CHAR, REAL, LONGREAL, INTEGER (or a subrange of INTEGER) or string. Their values will be assigned in the same way as variable parameters or the procedure read used with text files.

Reset

Usage:
```
reset
reset(f)
reset(f,s1)
reset(f,s1,s2)
```

Description

The procedure opens the file as a sequential file in the read-only state. The current-position pointer is initially positioned at the first component and then a get is performed.

Parameter Types

The parameter f must be a file which has been previously declared. It need not be closed before the call is made. If the file was open before the procedure call, it is automatically closed and reopened in the read-only state. If f is omitted, the file INPUT is assumed.

The second parameter and third parameters are described in Chapter 6.

Rewrite

Usage:
```
rewrite
rewrite(f)
rewrite(f,s1)
rewrite(f,s1,s2)
```

Description

The procedure opens the file as a sequential file in the write-only state. The current-position pointer is positioned at the first component. Any components previously existing in the file are discarded and the file buffer is undefined. The function eof(f) returns TRUE.

Parameter Types

The parameter f must be a file which has been previously declared. It need not be closed before the call is made. If the file was open before the procedure call, it is automatically closed and reopened in the write-only state. If parameter f is not included, the file INPUT is assumed.

The second and third parameters are explained in Chapter 6.

NOTE: If the Pascal file is associated with a system file by the procedure rewrite, all contents of that system file are destroyed.

Seek

Usage: seek(f,k)

Description

The procedure seek positions file f at component k. Component k will be read by the next call to get or written by the next call to put. If k is greater than the position of the last component in file f, then eof(f) becomes TRUE. The content of the file buffer, f^, is undefined following the call to seek.

Parameter Types

The parameter f must be a file which has been previously declared and opened as a direct-access file.

The parameter k must be a positive integer. It need not be less than or equal to the position of the last component of file f. However, if k is greater than the last possible position in the file, the function eof(f) will return TRUE, and any attempt to access the file at that time will produce a run-time error.

Write

Usage: write(p)
write(p1,...,pn)
write(f,p)
write(f,p1,...,pn)

Description

The procedure write places the values of its write parameters into a file previously opened as a direct-access file, or sequential file in the write-only state.

Parameter Types

The parameter f must be a file which has been previously declared and opened in a write-only state. If file identifier f is not included as a parameter, the file OUTPUT is assumed.

Heap variables 1024 words or larger cannot be parameters for write, writedir, or writeln.

Writedir

Usage: writedir(f,k,v)
writedir(f,k,vl,...,vn)

Description

The procedure writedir is used to perform a write operation on a particular component of a direct-access file.

The procedure statement

```
writedir(f,k,v)
```

is equivalent to the statements

```
seek(f,k);
write(f,v)
```

And the procedure statement

```
writedir(f,k,vl,...,vn)
```

is equivalent to the statements

```
seek(f,k);
write(f,vl,...,vn)
```

Parameter Types

The file parameter f must be a file which has been previously declared and opened as a direct-access file by the procedure open.

The parameter k must be a positive integer. If it is greater than the position of the last available component of the file, the function eof(f) will return TRUE.

The variable parameters must be of a type which is assignment-compatible with the file's component type.

Writeln

Usage:
```
writeln(p)
writeln(f,p)
writeln(pl,...,pn)
writeln(f,pl,...,pn)
```

Description

The procedure writeln places the values of its write parameters into the text file f, and appends a line marker to the file immediately following the last character. Writeln will also empty the contents of the file buffer.

The statement:

```
writeln(f,pl,...,pn)
```

is equivalent to the statements:

```
write(f,pl,...,pn);
writeln
```

Parameters

The file identifier f must be a text file which has previously been opened. If the parameter f is not included, the predeclared file OUTPUT is assumed.

The parameters p,pl,...,pn are write parameters as explained in Chapter 6.

Dynamic Allocation And De-allocation Procedures

Overview

Pascal allows variables to be created during program execution. The space allocated to dynamic variables can then be deallocated and later allocated to another variable. Dynamic allocation and deallocation are useful when variables are needed only temporarily, and when a program contains data structures whose maximum size may vary each time the program is run. Examples are temporary buffer areas and dynamic structures such as linked lists or trees. Dynamic variables are not explicitly declared and cannot be referred to directly by identifiers.

The standard procedure NEW is used to create variables dynamically. When a dynamic variable is no longer needed, the area of memory it occupies can be deallocated by using the standard procedure dispose. The area of memory reserved for dynamic variables is called the "heap".

When it is known in advance that a group of dynamic variables may be needed only temporarily, the state of the heap before these variables are allocated can be recorded using the procedure mark. The variables are then allocated as needed using new. When the variables are no longer needed, the procedure release can be used to return the heap to the state previously recorded by the mark procedure. The effect of this is to dispose all variables allocated after the call to mark.

If a program attempts to create a variable when there is not enough space remaining in the heap to do so, the following message is printed:

```
*** PASCAL ERROR: HEAP/STACK COLLISION IN LINE xxxx
```

and the program is aborted.

This section contains descriptions of the standard dynamic allocation and deallocation procedures. Chapter 9 includes information about Pascal/1000's implementation of heap management, and a section describing a set of alternative short versions of heap procedures available to the user.

New

Usage: `new(p);`

where p is a variable of type pointer.

Description:

The pointer variable p can only point to dynamic variables of a particular type. Variable p is then said to be bound to this type. For example, suppose p is to be bound to type T. Then, the following statements could be used:

```
TYPE                                        TYPE
  T     = <type definition>;                  T = <type definition>;
  TPTR  = ^T;                    -OR-
                                            VAR
VAR                                           p : ^T;
  p : TPTR;
```

When new(p) is executed, a section of the heap large enough for a variable of type T is allocated, and the address is returned in the pointer variable p. If the heap area resides in the 32K logical address space (this is the default compiler option $HEAP 1$) then the value of p is a one-word address. If the heap resides in EMA ($HEAP 2$), the value of p is a two-word address.

NOTE: Syntax error 189 will occur if new, dispose, mark or release is used with the $HEAP 0$ compiler option.

The new variable is denoted by dereferencing the pointer using the symbol "^" after the pointer identifier. Thus, p^ is used in the same manner that an identifier for a static variable is used. Pointer dereferencing is discussed further in Chapter 5.

If type T is a record with variants, then the amount of space allocated is the amount required for the fixed part plus the largest variant.

If type T is a record with variants having tag fields t1, t2, ..., tn, then tag values can be specified at the time of allocation by using an alternative form of new:

Usage: new(p,v1,v2,...,vn);

The tag field constant values v1,...,vn must be listed contiguously and in the order of their declaration. The amount of memory allocated to the record is determined by the size of the variants invoked by these tag values. These values must not be changed since no other variants of the record can be invoked as long as the record exists in the heap. The tag field values are used by NEW only to determine the amount of space needed, and are not assigned to the tag fields by this procedure.

Dispose

Usage: dispose(p);

where p is a variable of type pointer.

Description:

When dispose(p) is called, the heap area occupied by the variable pointed to by p is deallocated. The value of p is set to nil.

If the second form of new was used to allocate p, then the alternate form of dispose must be used.

Usage: dispose(p,v1,v2,...,vn);

The tag field values should be identical to those specified when the record was allocated. Dispose(p) will cause a run-time error if variants of p are in effect that cause the size of p to be different from its size when it was allocated. This occurs if a tag field value, specified in the call to new, is changed in such a way that a larger (or smaller) variant of p is referred to. Note that the tag values themselves are not checked, but only the object's size. Care should be taken to ensure that the program is correct.

An error occurs if the value of the pointer parameter of dispose is nil or undefined.

Mark

Usage: mark(p);

where p is a variable of type pointer.

Description:

The state of the heap is maintained by Pascal throughout program execution. The procedure mark uses the pointer variable p to preserve the current heap state information. The variable p may be any pointer type and must not be subsequently altered by assignment.

More information may be found in Chapter 9.

Release

Usage: release(p);

where p is a variable of type pointer.

Description:

This procedure restores the state of the heap to its state when p was marked. This has the effect of disposing all heap variables allocated since p was marked. The parameter p is set to nil. An error will occur if the value of p is already nil or is not the result of a mark(p).

Transfer Procedures

Pack

The standard procedure PACK assigns the values of elements of an unpacked array to a packed array.

Usage: pack(a,i,z)

where a is of type ARRAY [m..n] OF t;

i is of a type compatible with the index type of array a.

z is of type PACKED ARRAY [u..v] OF t;

The length of the unpacked array must be greater than or equal to the length of the packed array.

The procedure successively assigns the values of the elements of array a, starting with a[i], to the elements of array z, starting with z[u]. All elements of array z (z[u]...z[v]) are assigned values of elements from array a.

The example below uses arrays that have index types compatible with integer.

Example:

```
VAR
   a : ARRAY [1..10] OF CHAR;
   z : PACKED ARRAY [1..8] OF CHAR;
   i : INTEGER;
   .
   .
   .
BEGIN
   .
   .
   .
   i := 1;
   pack(a,i,z);
   .
   .
   .
END.
```

After pack(a,i,z) is executed, the array z contains values from the first eight elements of array a. The difference in size between arrays a and z determines the values of i that can be used. In the above example, the value of i must be 1,2, or 3. If the value of i is 3, then the 3rd through 10th elements of a are assigned to z. If the value of i is 4 an error will occur when PACK tries to access a[11] since PACK attempts to assign values to all eight elements of array z. The value of i must also be greater than or equal to the lower bound of the unpacked array.

In general the following condition must be true:

```
lbl <= i <= len1-len2 + 1
```

where len1 is the number of elements in the unpacked array, lbl is the lower bound of the unpacked array, and len2 is the number of elements in the packed array.

The program in the previous example has the same result if the statment "pack(a,i,z)" is replaced by:

```
FOR j := u TO v DO
    z[j] := a[j-u+i]
```

In more general terms, the statement "pack(a,i,z)" can be stated:

```
BEGIN
  k := i;
  FOR j := u TO v DO
    BEGIN
      z[j] := a[k];
      IF j <> v THEN k:= succ(k);
    END;
END;
```

where the iterative variable j is the same type as the index type of array z, the indexing variable k is the same type as the index type of array a, and i is an expression that is compatible with the index type of array a.

Note that the index types of arrays a and z do not have to be compatible.

Unpack

The procedure UNPACK assigns the values of elements of a packed array to an unpacked array.

Usage: unpack(z,a,i)

where z is of type PACKED ARRAY [u..v] OF t;

a is of type ARRAY [m..n] OF t;

i is a type compatible with the index type of array a.

The procedure successively assigns the values of array z, starting with z[u], to the elements of array a, starting with a[i]. All element values of array z are assigned to elements in array a.

Example:

```
VAR
   a : ARRAY[1..10] OF CHAR;
   z : PACKED ARRAY [1..8] OF CHAR;
   i : INTEGER;
   .
   .
   .
BEGIN
   .
   .
   .
   i := 1;
   unpack(z,a,i);
   .
   .
   .
END;
```

After unpack(z,a,i) is executed, the elements a[1] through a[8] contain values from the eight elements of array z. As in the previous example for pack, the value of i must be 1,2, or 3. If i has any other value an error occurs when unpack attempts to index array z beyond the range of its index type. As in the procedure pack, i must be such that:

```
lb1 <= i <= len1 - len2 + 1
```

where len1 is the number of elements in array a (the unpacked array), lb1 is the lower bound of array a, and len2 is the number of elements in array z (the packed array).

In the above program, the statement "unpack(z,a,i)" is equivalent to:

```
FOR j := u TO v DO
  a [j - u + i] := z [j]
```

In general, the statement "unpack(z,a,i)" is equivalent to:

```
BEGIN
  k := i;
  FOR j := u TO v DO
    BEGIN
      a[k] := z[j];
      IF j <> v THEN k := succ(k);
    END;
END;
```

where the iterative variable j is the same type as the index type of array z, the indexing variable k is the same type as the index type of array a, and i is an expression that is compatible with the index type of array a.

The index types of a and z do not have to be compatible.

Additional Procedures

Halt

The procedure HALT terminates the program.

Usage: halt(n)

where n is an integer expression.

When halt(n) is executed, the integer n is displayed at the same place run-time errors are displayed. This is the terminal from which the program was scheduled, or the system console if the program was not scheduled interactively.

For example, if x is equal to zero in the following program fragment:

```
    .
    .
    .
  CONST
     div_by_0 = 99;
  VAR
     x,y = REAL;
    .
    .
    .
    IF x <> 0.0 THEN
       y := y/x
    ELSE
       halt (div_by_0);
    .
    .
    .
```

the message:

PASCAL HALT: 99

will be displayed.

Arithmetic Functions

There are eight predefined arithmetic functions in Pascal/1000. Each of these functions is passed an arithmetic expression as a parameter and returns a numeric value.

The type of the value returned depends on the type of the parameter passed. The functions abs (absolute value) and sqr (square) return integer values if integer values are passed to them. The other arithmetic functions return real values if integer values are passed to them. All of the functions return a real or longreal value when a real or longreal parameter is passed.

To compute the values of the functions, Pascal/1000 uses system routines and compiler-defined algorithms. For each function the main routine used to determine the result is listed in Table 7-1. For more detailed information about the methods used, the user should refer to the assembler code generated.

Table 7-1. System Routines Called to Calculate Function Values.

	INTEGER	REAL	LONGREAL
abs	--	--	--
sqr	.DMP	.FMP	.TMPY
sqrt	SQRT	SQRT	.SQRT
exp	EXP	EXP	.EXP
ln	ALOG	ALOG	.LOG
sin	SIN	SIN	.SIN
cos	COS	COS	.COS
arctan	ATAN	ATAN	.ATAN

Errors which occur within system arithmetic routines cause error messages to be sent to the system log device. Refer to the system routines to find the error messages generated.

Abs

Usage: abs(x)

Compute the absolute value of x.

```
Examples:   abs(-13)        returns  13
            abs(-7.11)      returns  7.110000E+00
```

Sqr

Usage: sqr(x)

Compute the value of x squared, (x^2). If the value to be returned is greater than the maximum value for that type the largest value of the type is returned.

```
Examples:    sqr(3)          returns  9
             sqr(1.198E3)    returns  1.435204E+06
```

Sqrt

Usage: sqrt(x)

Compute the square root of x, $(x^{1/2})$. If x < 0 then an error message is sent to the log device and a value of zero is returned.

```
Examples:    sqrt(64)         returns 8.000000E+00
             sqrt(13.5E12)    returns 3.674235E+06
             sqrt(-5)         returns  0
```

Exp

Usage: exp(x)

Compute e (base of the natural logarithms) to the power of x (e^x). If x < -129 * ln(2) then an underflow occurs and a value of zero is returned without an error message. If x > 128 * ln(2) then an overflow occurs. An error message is sent to the log device and a value of zero is returned.

```
Examples:      exp(3)       returns  2.008554E+01
               exp(8.8E-3)  returns  1.008839E+00
```

Ln

Usage: ln(x)

Compute the natural logarithm of x. If x < 0 then a value of zero is returned and an error message is sent to the log device.

```
Examples:      ln(43)       returns  3.761200E+00
               ln(2.121)    returns  7.518877E-01
```

Sin,Cos

```
Usage:   sin(x)
         cos(x)
```

Compute the sine and cosine of x, where x is interpreted to be in radians. If x is outside of the range -8192 * pi..8192 * pi then an error message is sent to the log device and the value zero is returned.

```
Examples:     sin(0.024)      returns  2.399769E-02
              cos(1.62)       returns  -4.918370E+00
```

Arctan

```
Usage:   arctan(x)
```

Compute the arctangent of x. The result is in radians within the range -pi/2..pi/2.

```
Examples:     arctan(2)         returns  1.107149E+00
              arctan(-4.002)    returns  -1.325935E+00
```

Predicates

The following three procedures return Boolean results.

Odd

Usage: odd(x)

where x is an INTEGER (or a subrange of INTEGER).

Description:

The procedure odd returns TRUE if x is odd, and FALSE otherwise.

Examples:

Procedure Statement	Result
odd(6)	FALSE
odd(-32767)	TRUE
odd(32768)	FALSE
odd(0)	FALSE

Eof

Usage: eof
eof(f)

where f is file which has previously been declared and opened.

Description:

The function eof(f) returns TRUE if the file f is not in a read-only state, if a direct-access file is positioned beyond maxpos(f), or if no component remains for sequential input.

If the parameter f is omitted, the file INPUT is assumed.

Eoln

Usage: eoln
eoln(f)

where f is a file which has previously been declared and opened.

Description:

The function eoln(f) returns TRUE if the text file f is positioned at the end of a line.

If the parameter f is omitted, the file INPUT is assumed.

Transfer Functions

Trunc

Usage: trunc(x)

Function trunc returns an integer result which is the integral part of the real or longreal expression x. The absolute value of the result is not greater than the absolute value of x. An error will occur if the result is not within the integer range.

```
Example:    trunc(5.61)      returns  5
            trunc(-3.38)     returns  -3
            trunc(18.999)    returns  18
```

Round

Usage: round(x)

Function round returns the integer value of the real or longreal expression x rounded to the nearest integer. If x is positive or zero then round(x) is equivalent to trunc(x + 0.5); otherwise, round(x) is equivalent to trunc(x - 0.5). An error will occur if the result is not in the integer range.

```
Example:    round(3.1)       returns  3
            round(-6.4)      returns  -6
            round(-4.6)      returns  -5
```

Ordinal Functions

Ord

Usage: ord(x)

where x is an expression of ordinal type.

The function ord returns the ordinal number associated with the value of x. If the result can be contained in one word, a one-word result is returned, otherwise, a two-word result is returned. If the parameter evaluates to an integer value, then this value is returned as the result. If x is of type char, then the result is an integer value between 0 and 255 determined by the ASCII ordering. If x is of any other ordinal type (i.e., a predefined or user-defined enumeration type) then the result is the ordinal number determined by mapping the values of the type onto consecutive non-negative integers starting at zero.

The predeclared type Boolean, for example, is defined:

```
TYPE BOOLEAN = (false,true)
```

thus,

```
ord(false)  returns  0
ord(true)   returns  1
```

The same method is used to determine the ordinality of an element in a user-defined enumeration type. For example, given the declaration:

```
TYPE color = (red,blue,yellow);

ord(red)    returns  0
ord(blue)   returns  1
ord(yellow) returns  2
```

Additional Examples:

Value of x	Value of ord(x)
'a'	97
'A'	65
-1	-1
1000	1000

Chr

Usage: chr(x)

where x is an integer expression.

The function chr returns the character value whose ordinal number is equal to the value of the integer expression x. No range checking on the value of x is performed. If the value of x is not within the range 0..255 then chr(x) yields x as its result. A type compatibility error occurs if this result is assigned to a variable of type char.

For any character ch, the following is true:

```
chr(ord (ch)) = ch
```

Examples:

Value of x	Value of chr(x)
63	'?'
100	
13	(carriage return)

Succ

Usage: succ(x)

where x is an expression of ordinal type.

The function succ returns as its result a value whose ordinal number is one greater than that of the expression x. The result is of a type identical to that of x. If no such value exists, no error is reported at the function call, but a run-time error will occur if the value is assigned to a variable of the ordinal type.

For Example, given the declaration:

```
TYPE color = (red, blue, yellow);
```

the following is true:

```
succ(red) = blue
succ(yellow) returns a value that is not of type color.
```

Additional Examples:

Value of x	Value of succ(x)
1	2
-5	-4
'a'	'b'
false	true

Pred

Usage: pred(x)

where x is an expression of ordinal type.

The function returns as its result a value whose ordinal number is one less than that of expression x. If no such value exists, no error is reported at the function call, but a run-time error will occur if the value is assigned to a variable of the ordinal type.

Given the declaration:

```
TYPE day = (monday,tuesday,wednesday);
```

the following is true:

```
pred(tuesday) = monday
pred(monday)    returns a value that is not of type day.
```

Additional Examples:

Value of x	Value of pred(x)
1	0
-5	-6
'B'	'A'
true	false

File Handling Functions

The functions linepos, position, and maxpos, enable the program to determine the current position in a file relative to the end-of-file and end-of-line.

Linepos

Usage: linepos(f)

where f if a previously opened text file.

Description:

The function linepos returns the integer number of characters read from or written to the text file f since the last EOLN. This does not include the character in f^.

Position

Usage: position(f)

where f is a previously opened file.

Description:

The function position returns the integer index of the current component of the file f, starting with 1. This component is the next to be accessed by read or write. The file f may not be a text file. If the buffer f^ is full, the result if the index of that component.

Maxpos

Usage: maxpos(f)

where f is a previously opened direct-access file.

Description:

This function returns the integer index of the last component of the file f. This index equals the maximum possible number of components which can be contained in file f. The number varies for different files according to component size.

Chapter 8
Implementation Considerations

A practical knowledge of the implementation of Pascal/1000 is useful for efficient programming. This chapter describes data allocation, memory configuration, data and stack managment, heap management, and efficient programming.

Data Allocation

The Pascal/1000 compiler converts source code into assembly language instructions and data definitions. The space reserved by the data definitions is used to represent structured constants, and variables. Type definitions are used only by the compiler and do not result in data allocation.

The size of the data allocation is determined by the type of the variable or structured constant. A variable or structured constant of a PACKED type (refer to Chapter 4) is given data allocations that optimize space utilization. An unpacked data type is given an allocation that allows faster data access.

This section describes the size in bits and words of the data allocation for a variable or structured constant of a particular type and the boundary alignment conventions for that allocation.

Allocations for structured constants are identical to the allocations for variables of the same type as the structured constant.

Allocation for Scalar Variables

Table 8-1 shows the allocations for variables of scalar, subrange, and pointer types. All allocations begin on word boundaries.

Table 8-1. Allocations for Scalar Variables

Type	Size	Notes
BOOLEAN	1 word	FALSE is represented as 0 TRUE is represented as 1
INTEGER	2 words	Bit 15 of the first word is the sign bit.
Subrange of INTEGER	1 or 2 words	Subranges contained in -32768..32767 require 1 word to represent variables of that type. All other subranges require 2 words to represent variables of that type. Examples: Subrange — Allocation 0..8 — 1 word -32768..32767 — 1 word 10..40000 — 2 words -70000..-1 — 2 words
Enumeration	1 word	The values are represented internally as 1-word integers in the subrange 0..(cardinality of the enumeration type - 1)
Subrange of Enumeration	1 word	Represented by their enumerated value.
REAL	2 words	Floating point format.
LONGREAL	4 words	Floating point format.
CHAR	1 word	The character is represented in the right byte (the left byte contains 0).
Pointer	1 or 2 words	1 word if $HEAP 1$ compiler option used. 2 words if $HEAP 2$ compiler option used.

Allocation for Structured Variables

Table 8-2 shows the allocations for variables of array, record, file, and set types. All allocations begin on word boundaries.

Table 8-2. Allocations for Structured Variables

Unpacked Type	Size
ARRAY	The size of an array allocation is the sum of the allocations of its elements: (product of cardinalities of index types) * (allocation of one element) The elements are stored in row major order.
RECORD	The size of a record allocation is the sum of the allocation of the fixed part and, if any, the allocations of the tag field and the largest variant.
FILE	Let b be the number of buffers allocated for the file DCB (as specified by the BUFFER compiler option, default is 1) s be the I/O line size (as specified by the LINESIZE compiler option, default is 80) Allocation size for a text file (words): 30 + 128*b + (s+1 DIV 2) Allocation size for a non-text file (words): 29 + 128*b + allocation size of base type
SET	Let n be the cardinality of the base type. n: <= 16 — Allocation: 1 word n: > 16, <= 32767 — Allocation: 1 word to represent n plus the number of words required to represent n bits, i.e., 1 + (n+15 DIV 16) If the compiler cannot discern the base type, 17 words (1 word + 16 words to represent 256 elements) are allocated. Each element of the base type is represented by its corresponding bit of the set variable, with the first element represented by the most significant bit. If the element is not in the set, the bit is 0, and if the element is in the set, the bit is 1.

Allocation for Elements of Packed Structures

Arrays, records, sets, and files may be packed by prefixing the type definition with "PACKED". This indicates to the compiler that the non-structured elements of the type are to be packed.

In general, packed variables are allocated as small a space as is possible, with the following guidelines used in the interest of accessibility:

a. Any item requiring a word or less of storage will not cross a word boundary.

b. Any item requiring a word or more of storage will be aligned on a word boundary.

The packed attribute of a structured type does not distribute to the structured elements of the type. For example, the elements of an array within a packed record are not packed. (They may be packed, however, by prefixing the array definition with "PACKED").

Table 8-3. Allocations For Elements of Packed Structures.

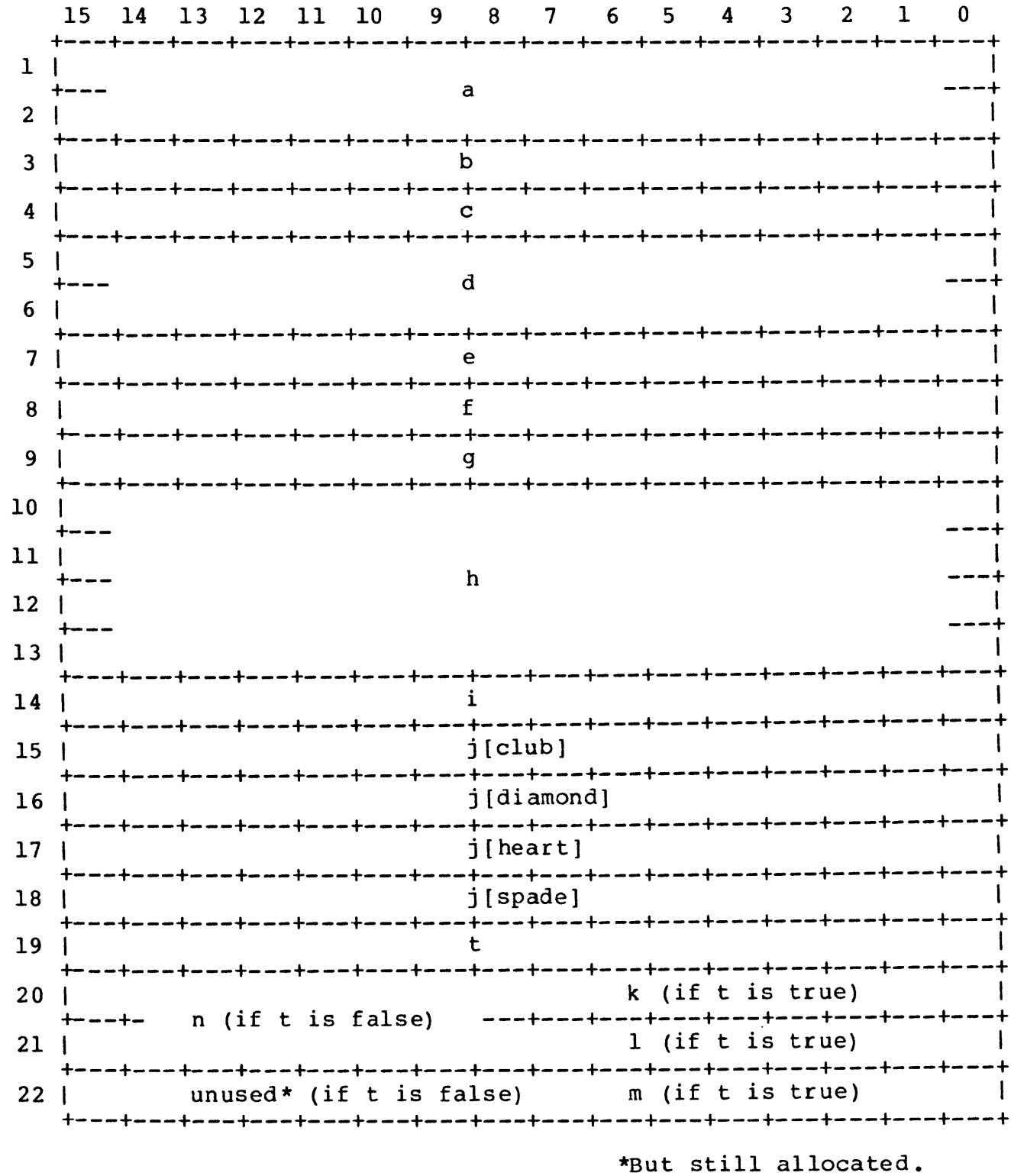
Type	Allocation	
BOOLEAN	Size:	1 bit
	Alignment:	Bit boundary
INTEGER	Size:	2 words
	Alignment:	Word boundary
Subrange of INTEGER	Size:	Minimum number of bits necessary to represent each value of the subrange
	Alignment:	Bit boundary
Enumeration	Size:	Minimum number of bits necessary to represent the value (cardinality of type - 1)
	Alignment:	Bit boundary
Subrange of Enumeration	Size:	Minimum number of bits necessary to represent the value (cardinality of subrange - 1)
	Alignment:	Bit boundary
REAL	Size:	2 words
	Alignment:	Word boundary
LONGREAL	Size:	4 words
	Alignment:	Word boundary
CHAR	Size:	1 byte (8 bits)
	Alignment:	Bit boundary
Pointer	Size:	1 word if $HEAP 1$ 2 words if $HEAP 2$
	Alignment:	Word boundary
SET	Let n be the cardinality of the base type.	
	n <= 16:	
	Size:	n bits
	Alignment:	Bit boundary
	16 < n <= 32767:	
	Size:	1 word + m words where m is the number of words needed to hold n bits.
	Alignment:	Word boundary

Examples of Packed and Unpacked Structures

Example 1: Assume the following:

```
TYPE
   SUIT = (club,diamond,heart,spade);

VAR
   r : RECORD
          a : INTEGER;
          b : 1..13;
          c : SUIT;
          d : REAL;
          e : CHAR;
          f : 'A'..'Z';
          g : BOOLEAN;
          h : LONGREAL;
          i : SET OF SUIT;
          j : ARRAY [SUIT] OF 1..13;
          CASE t : BOOLEAN OF
             true:  (k,l,m : CHAR);
             false: (n     : INTEGER)
       END;
```

Variable r is allocated as follows:

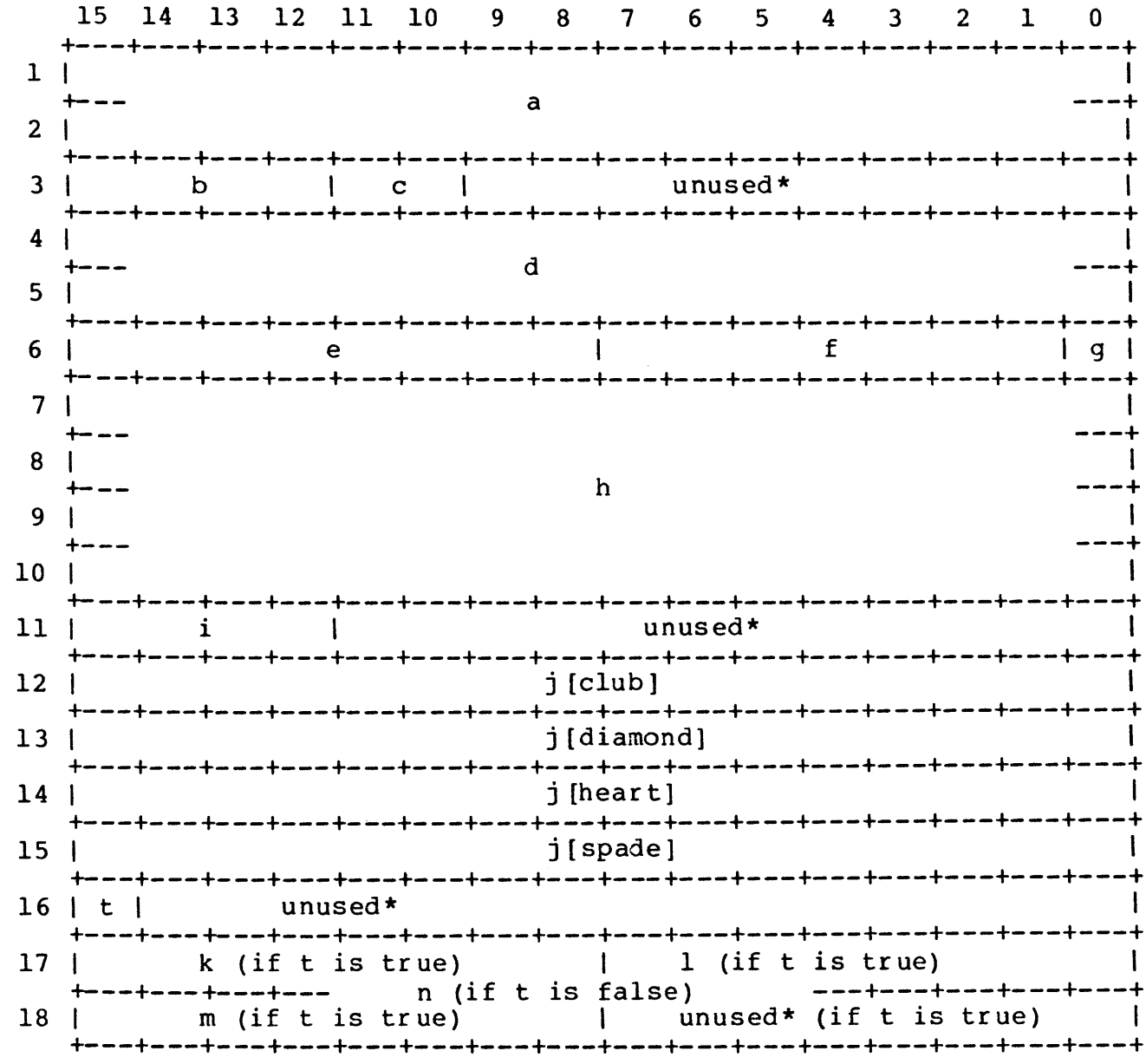


*But still allocated.

Example 2: This is the same structured variable as in the previous example, but now r is packed. Note that field j is not packed.

```
TYPE
   SUIT = (club,diamond,heart,spade);

VAR
   r : PACKED RECORD
          a : INTEGER;
          b : 1..13;
          c : SUIT;
          d : REAL;
          e : CHAR;
          f : 'A'..'Z';
          g : BOOLEAN;
          h : LONGREAL;
          i : SET OF SUIT;
          j : ARRAY [SUIT] OF 1..13;
          CASE t : BOOLEAN OF
             true:  (k,l,m : CHAR);
             false: (n     : INTEGER)
       END;
```

Packed variable r is allocated as follows:



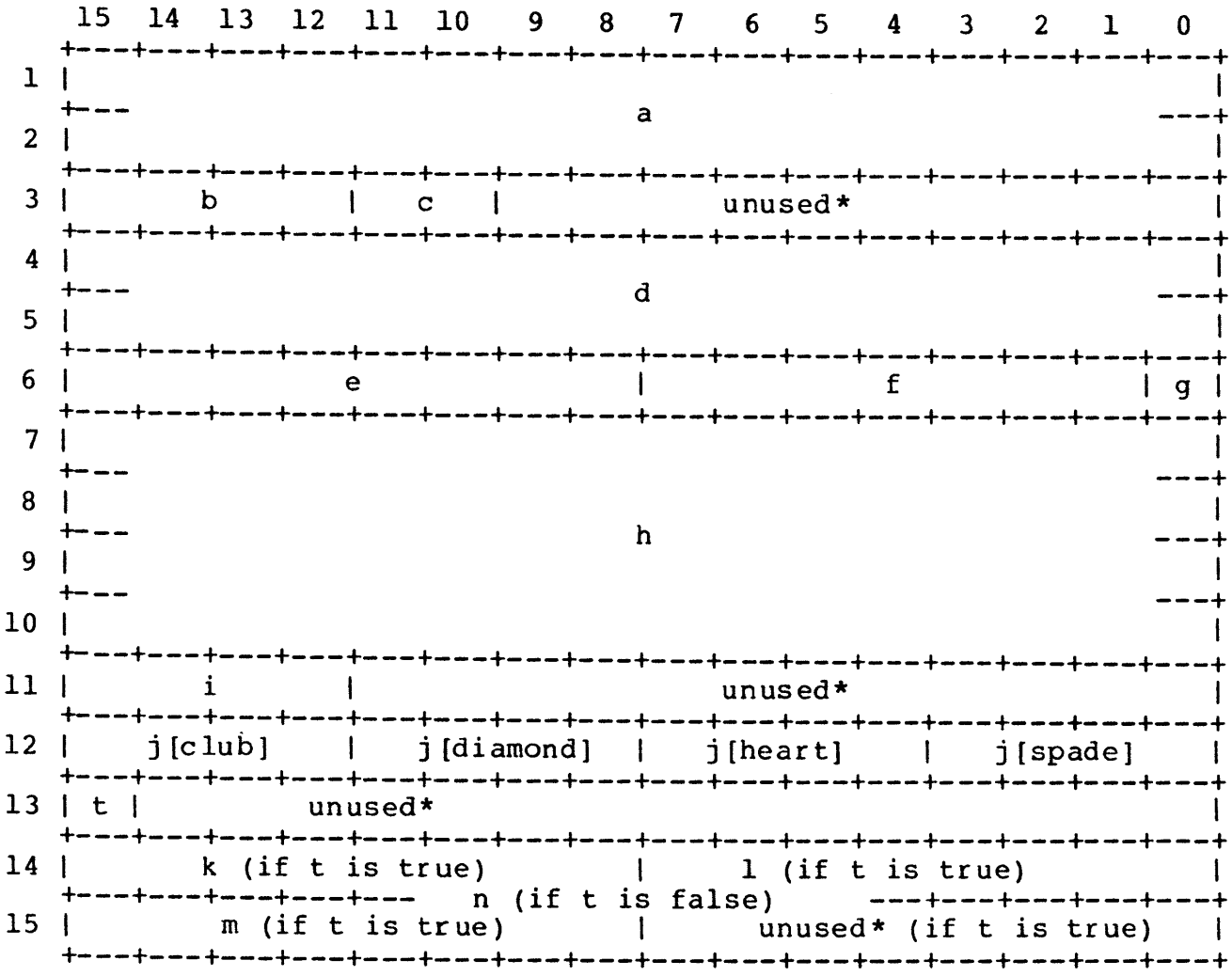
*But still allocated.

Note that the elements at the array j are not packed, but the array as a whole is treated as a field of the packed record.

Example 3: This is similar to the previous example, but field j is now packed.

```
TYPE
   SUIT = (club,diamond,heart,spade);

VAR
   r : RECORD
          a : INTEGER;
          b : 1..13;
          c : SUIT;
          d : REAL;
          e : CHAR;
          f : 'A'..'Z';
          g : BOOLEAN;
          h : LONGREAL;
          i : SET OF SUIT;
          j : PACKED ARRAY [SUIT] OF 1..13;
          CASE t : BOOLEAN OF
             true:  (k,l,m : CHAR);
             false: (n     : INTEGER)
       END;
```

Variable r is allocated as follows:



*But still allocated.

Memory Configuration

Memory configuration, as discussed in this section, is the configuration of a partition in which a Pascal/1000 program is running. Figure 8-1 illustrates the allocation of memory in a partition.

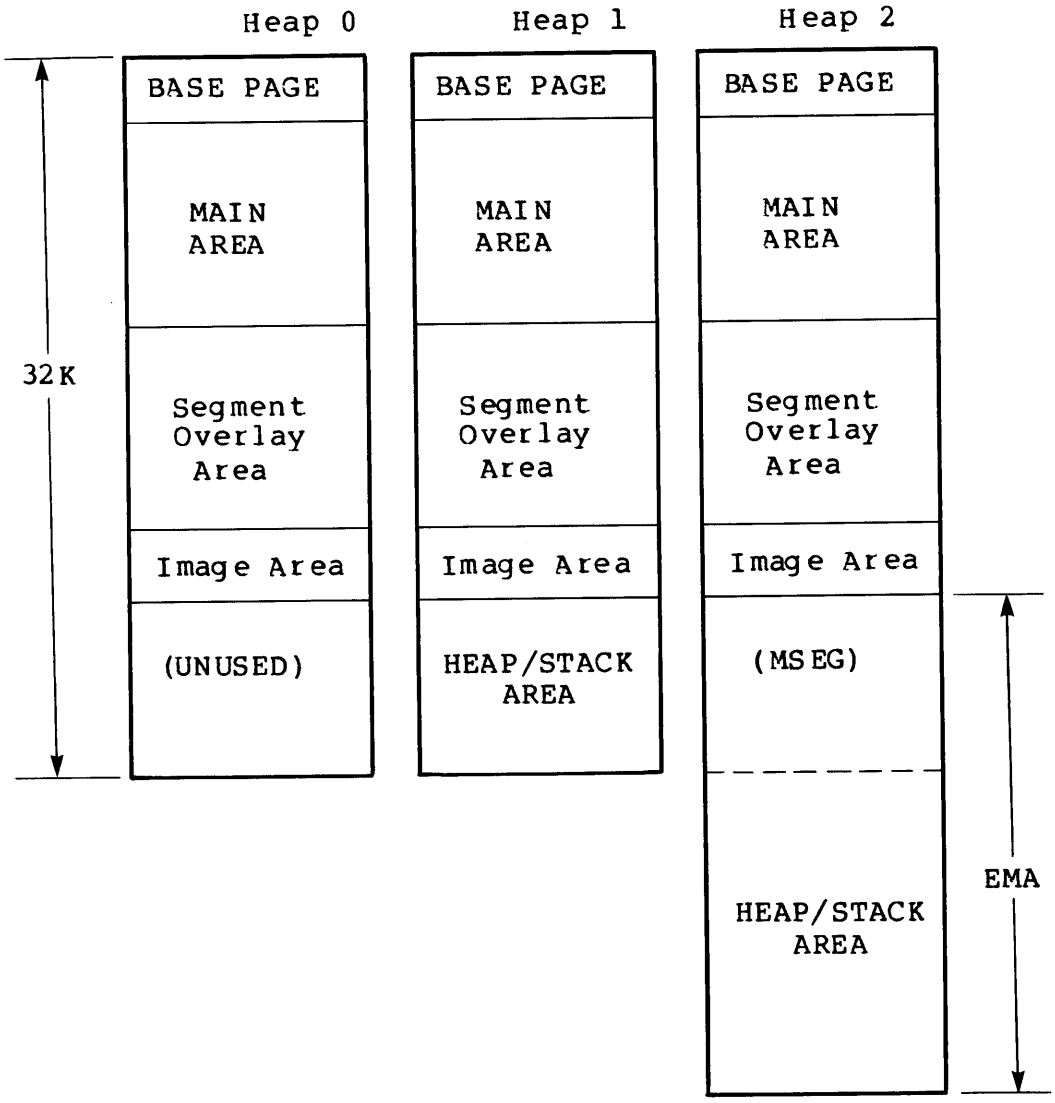


Figure 8-1. Pascal/1000 Memory Configuration

The areas labeled in upper case are always allocated; those labeled in lower case are optional.

In the heap/stack area, the stack begins at the low end of the heap/stack area and grows toward increasing addresses, and the heap begins at the high end of the heap/stack area and grows toward decreasing addresses.

In the Heap 2 configuration, the heap/stack area is allocated in EMA.

Base Page

The base page is the first logical page of the partition. It contains the communication area of the operating system, driver links, trap cells for interrupt processing, and operating system and user program links.

Main Area

The main area of a Pascal/1000 partition contains the main program unit, subprogram units that have been combined with the main program unit, routines not written in Pascal/1000, and library routines. Figure 8-2 illustrates the configuration of the main area for the sample program "main". The main program unit is divided into separate sections of code and data for each routine and for the main program.

The code section defines the actions as described in the body of the routine or program.

The data section for a routine contains the routine's local variables, temporaries used in parsing expressions, formal parameters, return addresses, dynamic link, and entry count. The dynamic link points to the data section of the previous activation of a recursively called routine. The entry count stores the current level of recursion of a recursively called routine. The main data section contains the global and temporary variables.

The main program unit's routines are allocated first. If routine B is declared in routine A (as with proc3 and proc2 in Figure 8-2), the code and data for B are located before the code and data for A. The main code and data follow the routines. The remainder of the area is allocated as the units are relocated. In Figure 8-2 the subprogram is relocated after the main program. Note that there is no subprogram main code as subprograms have no body, and there is no subprogram main data as the only variables declared in a subprogram are the global variables which are part of the main data section. The non-Pascal routine, fortn, was relocated next. The libraries were then searched to supply the routines used in the main area.

```
PROGRAM main;
  {main data declarations}

  PROCEDURE procl;
    {procl local declarations}
    BEGIN
      {procl code}
    END;

  PROCEDURE proc2;
    {proc2 data declarations}
    PROCEDURE proc3;
       {proc3 data declarations}
       BEGIN
         {proc3 code}
       END;
    BEGIN
      {proc2 code}
    END;

  PROCEDURE fortn;
    EXTERNAL;
    {fortn is a FORTRAN routine}

BEGIN
  {main code}
END.


$SUBPROGRAM$
PROGRAM subprogram;
  {redeclaration of global
     declarations}
  PROCEDURE subl;
    {subl data declarations}
    BEGIN
      {subl code}
    END;
  PROCEDURE sub2;
    {sub2 data declarations}
    BEGIN
      {sub2 code}
    END;

.  {End of subprogram}
```

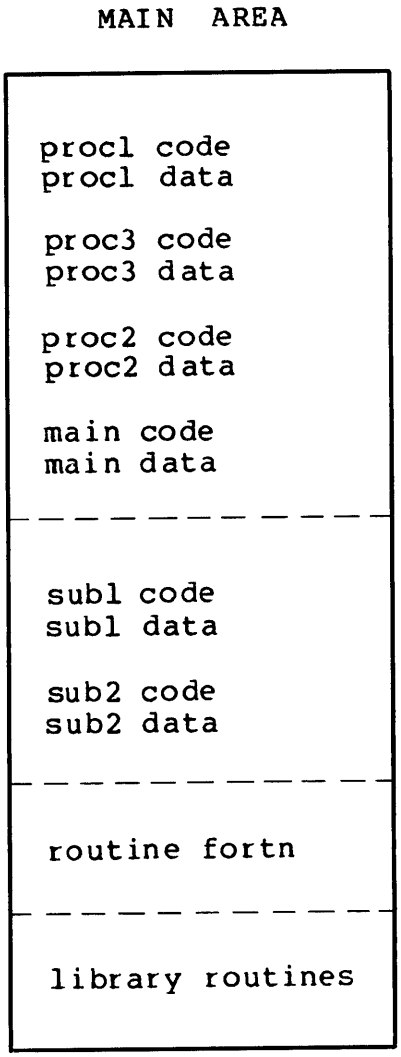


Figure 8-2. MAIN AREA

Segment Overlay Area

The segment overlay area is allocated only for segmented Pascal programs. The size of this area is the size required for the largest segment overlay. It is configured similarly to the main area except there is no main code or data because a segment unit does not have a body and the variables in the declaration section are redeclarations of the global variables. Subprogram units and non-Pascal routines are present only if they have been combined with the particular overlay. The loader combines with a segment overlay those library routines that are referenced by the overlay but have not already been combined in the main area. If a segment overlay is smaller than the segment overlay area, the remaining space at the end of the area is unused.

Image Area

If the Pascal/1000 program interfaces with the IMAGE/1000 subsystem then an area of the partition must be set aside for use by IMAGE. The compiler option $IMAGE n$ allocates n words of IMAGE area before the heap/stack area.

Heap/Stack

The heap/stack area contains the run-time heap and stack of the program. The size of this area is determined by the $HEAP n$ compiler option,

where:

n = 0: no heap/stack area (not allocated)

n = 1: heap/stack area in logical memory; allocated at the end of the partition (default value)

n = 2: heap/stack area in EMA;

These three conditions are illustrated in Figure 8-1.

The heap is used for dynamic data and is described in detail in the Heap Management section in this chapter. The stack is used for stacking data of recursive routines and is described in detail in the Stack Management section in this chapter.

The stack and heap "grow" toward each other during run time, the stack toward increasing addresses and the heap toward decreasing addresses. Each time data is placed on the stack or heap, a collision condition is checked. If they ever meet or attempt to cross each other, the run-time error:

*** PASCAL ERROR: HEAP/STACK COLLISION IN LINE XXXX

is displayed and the program will abort.

Data Management

There are three classes of data in Pascal/1000 programs:

1) global data
2) local data
3) dynamic data

Global data is declared in the declaration section of the main program block and is redeclared in each subprogram and segment unit (see Chapter 7). Global data is static; storage for the data is allocated once and remains accessible throughout the execution of the program. The scope of global data is the entire program.

The local data of a routine includes the routine's parameters and data declared in the declaration section of the routine block (see Chapter 4). Local data is also static since storage for the data is allocated once and remains accessible within the routine during each invocation. The scope of local data is the routine in which the data is declared. The initial values of local data are unspecified when the routine is invoked; i.e., no assumptions should be made regarding values "left over" from the previous invocation.

Dynamic data is allocated and de-allocated in the heap, and is accessed via pointer variables. Dynamic data is described in detail below (see HEAP MANAGEMENT).

Stack Management

The run-time stack is used by Pascal/1000 programs to save copies of a routines's local data during recursive calls. This is the only use of the stack; it is not used if no recursive calls are made.

The local data of a recursive routine is allocated contiguously in the code space of the routine. This data, along with the return address of the routine and a copy of the top of stack pointer, constitute the "activation record" of the routine.

If a routine is invoked recursively, either directly or indirectly, then the local data values of the previous invocation must be saved temporarily, since execution of the previous invocation is yet to be completed. Therefore, at the beginning of each recursive invocation, a copy of the activation record is first pushed (copied) onto the stack, thus preserving the data values, the return address, and the top of stack pointer.

When a routine that was recursively called completes, the copy of its activation record at the top of the stack is popped off and copied back into the activation record in the code space. Thus, the state of the previous invocation is restored.

During the execution of a routine, references to its local data are made to the current activation record in the code space. The data on the stack is referenced only indirectly via VAR parameters. When an activation record is copied onto the stack, any VAR parameters pointing to its data are adjusted to point to the copy on the stack. Likewise, when the activation record is copied back into the code space, these VAR parameters are re-adjusted.

Library routines @INH1,@INH2, @GHS1, @GHS2, @SHS1, and @SHS2 allow a user to initialize the stack and to retrieve and set information about the stack. See HEAP MANAGEMENT, below.

Heap Management

In order to make the most efficient use of the heap area, it is helpful to be familiar with the organization of the heap and with the specific effects of the heap management routines described briefly in Chapter 7.

Unless otherwise noted, the explanations and diagrams below assume that the heap resides in the 32K logical user address space. The differences between $HEAP 1$ and $HEAP 2$ are described at the end of this section.

Overview of Heap Organization

The heap consists of "data spaces" and "free spaces". Each data space is preceded by a "data block", and each free space is preceded by a "free block". These blocks contain header information described below.

A data space and its data block are allocated in the heap by a call to new. Each data space represents a dynamically-allocated variable. The pointer variable in the call is set to point to the beginning of the data space. The data block contains the size (in words) of its data space. This information is used when the data space is deallocated using dispose.

A free space and its free block are created from a data space when a pointer variable pointing to the space is used in a call to dispose. The variable is set to nil.

Every free space in the heap is linked into a "free space list" which is circularly-linked. This list is used to enable free spaces to be reallocated as data spaces. Whenever new is called, the free space list is first searched for a space large enough to be reallocated. Each free block contains the size (in words) of its free space and a link (pointer) to the next free block. The Pascal-managed "current free list pointer" curr_free always points to the current free list. A free space list is initialized with a dummy free block containing zero for the size and a link to itself.

No garbage collection or coalescing of adjacent free spaces is performed.

If, during a call to new, the current free space list does not contain a free space of sufficient size, the new data space is allocated at the current top of the heap.

Calls to mark divide the heap into "mark regions", each of which is preceded by a "mark block". Each mark region is independent of the others and has its own data spaces and free list. Each mark block contains a pointer to the free list of the previous mark region and a pointer to the previous mark block.

When new or mark is called for the first time, a dummy mark block containing two nil pointers is initially allocated at the base of the heap. Subsequently, whenever mark is called, a new mark block is allocated at the current top of the heap. The value of curr_free is saved in the block, along with the pointer to the previous mark block. Thus, the state of the heap up to and including the last mark region is preserved. The pointer variable in the mark call is set to point to the new mark block. The Pascal-managed "current mark block pointer" curr_mark is also set point to the new block. A dummy free block is allocated at the bottom of the new region, and curr_free is set to point to it.

When mark is called to begin a new mark region on the heap, previous mark regions below it are still "active" in the sense that their data spaces and free spaces are accessible as before. When a data space is disposed, the resulting free space is always linked into the free list of its mark region, i.e., the region that contained the disposed data space. However, new data spaces are allocated by calls to new only in the current mark region, i.e., the region at the top of the heap. The user can change the current mark region only by a call to mark or release.

When release is called with a pointer variable that has previously been set with a call to mark, the heap is reset to a previous state. The variable points to a mark block on the heap. From this mark block the pointer to the free list of the previous mark region is recovered, to which curr_free is reset. The pointer to the previous mark block is also recovered, to which curr_mark is reset. The pointer variable of the release call is set to nil. Thus, releasing a mark region also releases all the mark regions existing above it on the heap, except that the pointer variables pointing to the mark blocks above are not set to nil.

After a mark region has been released, none of its data spaces are accessible. However, pointer variables pointing to them are not set to nil. Therefore, a subsequent attempt to access one of these data spaces via a pointer variable will result in a run-time error.

The heap management routines utilize three Pascal/1000 library routines to initialize, retrieve, and change heap status information. These routines are:

```
@INH1  (initialize_heap)
@GHS1  (get_heap_stack_info)
@SHS1  (set_heap_stack_info)
```

Note: @INH2, @GHS2, and @SHS2 are used with the compiler option $Heap 2$.

The parameter for these routines must be of the record type INFO_REC in Figure 8-3 below. The heap management routines will be described in terms of the data structures and identifiers in Figure 8-3. The three library routines above can be made available in the user's program by including the code from Figure 8-3 in the program's declaration section.

The following description of heap management assumes the default compiler option $HEAP 1$.

```
CONST
   bsize    = 2;             {header block size for $Heap 1$      }
   minalloc = bsize DIV 2;   {minimum allocation size for $Heap 1$}

TYPE
   SIZE = 0..32767           {data and free space size}

   ADDR = 0..32767;          {one-word logical address}

   BLOCK_TYPE = (marc, free, data);

   BLOCK =
      RECORD
         CASE BLOCK_TYPE OF
            marc: (pptr  : ^BLOCK   {ptr to prev free list   }
                   mptr  : ^BLOCK); {ptr to prev mark block  }
            free: (fsize : SIZE;    {size of free space      }
                   fptr  : ^BLOCK); {ptr to next free block  }
            data: (dsize : SIZE)    {size of data space      }
      END;

   INFO_REC =
      RECORD
         tos,                  {top of stack         }
         toh,                  {top of heap          }
         init_tos,             {initial top of stack }
         init_toh,             {initial top of heap  }
         high_tos,             {highest top of stack }
         high_toh,             {highest top of heap  }
         curr_free,            {^ to curr. free list }
         curr_mark : ADDR;     {^ to curr. mark block}
      END;

PROCEDURE get_heap_stack_info
   $ALIAS '@GHS1'$
   (VAR heap_info : INFO_REC);
   EXTERNAL;

PROCEDURE set_heap_stack_info
   $ALIAS '@SHS1'$
   (heap_info : INFO_REC);
   EXTERNAL;

PROCEDURE initialize_heap
   $ALIAS '@INH1'$;
   EXTERNAL;
```

Figure 8-3. Heap Management Declarations and Routines

Heap Initialization

The heap is automatically initialized at the time of the first call to new or mark.' A new mark region is created by allocating a dummy mark block and a dummy free block. The free space list contains only the dummy free block.

The state of the heap after initialization is shown in Figure 8-4. Heap diagrams have higher addresses at the bottom, lower addresses at the top. The heap "grows" toward lower addresses.

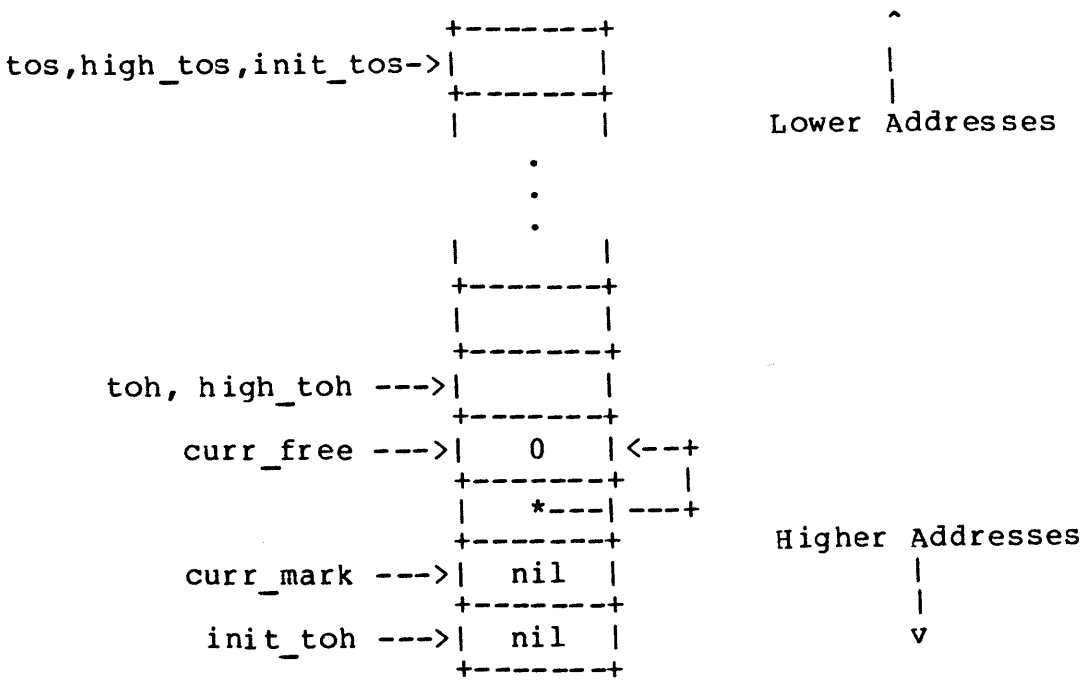


Figure 8-4. Heap/Stack Area After Initialization

The value of init_toh and init_tos do not change once they have been set during heap initialization.

The nil values in the mark block indicate that no previous mark region or free list exists. The size field of a dummy free block is always 0. The pointer in the free block points to itself, indicating that the free list in the current mark region is empty.

The actions taken by new, dispose, mark, and release are stated below and accompanied by examples. In each example, the heap is in the state that resulted from the preceding example.

The examples use the following variables:

```
TYPE
   INT = -32768..32767;
   A2  = ARRAY [1..2] OF INT;
   A3  = ARRAY [1..3] OF INT;
   A4  = ARRAY [1..4] OF INT;

VAR
   p1 : ^INT;
   p2 : ^A2;
   p3 : ^A3;
   p4 : ^A4;
   mark_ptr : ^INT;
```

New

See Figures 8-5 and 8-6.

For new (p):

1. Initialize the heap if necessary.

2. Search the free list in the current mark region for the first free space that is large enough to hold the new variable (data space).

   If found: Allocate the variable at the end of the free space and adjust the length of the free space.

   Else: Allocate the variable at the top of the heap if there is room. If not, report:

   *** PASCAL ERROR: HEAP/STACK COLLISION IN LINE xxxx

   and abort the program.

3. Set p to point to the variable. Set the word preceding the variable to the length of the variable in words.

4. Adjust toh and high_toh if necessary.

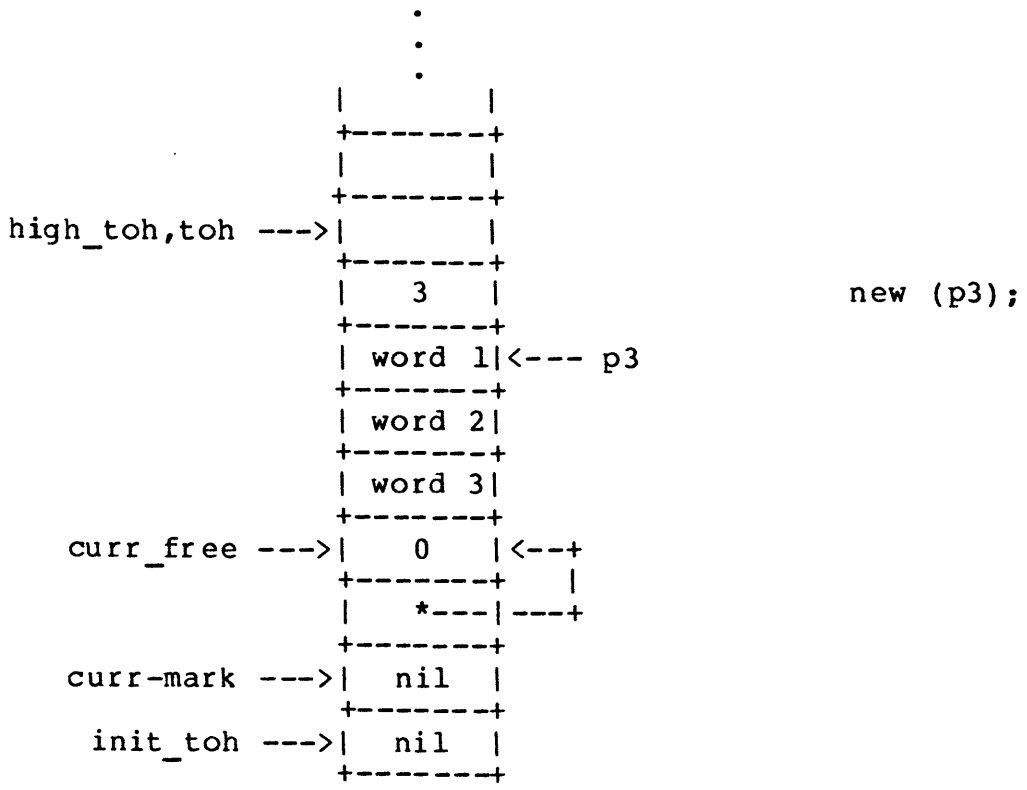


Figure 8-5. Allocation of a 3-word Variable

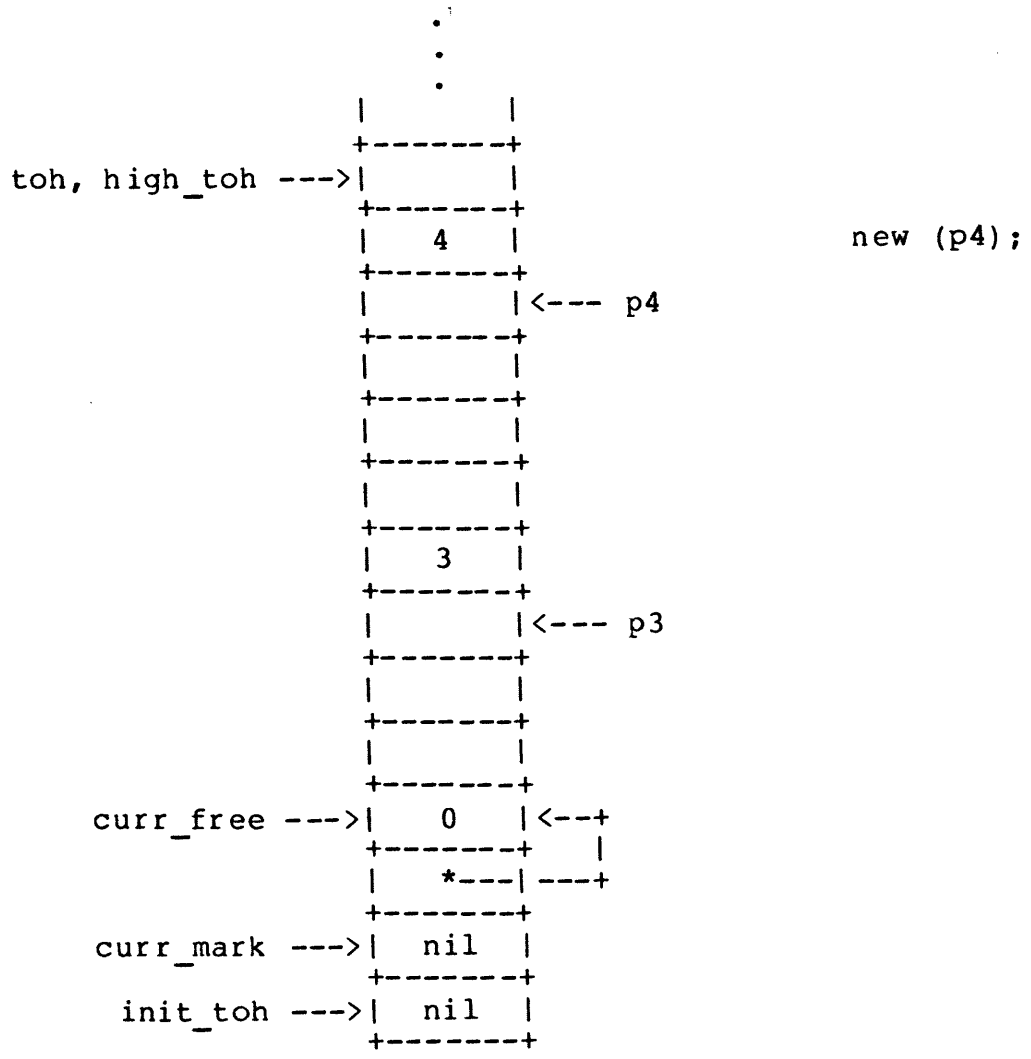


Figure 8-6. Allocation of a 4-word Variable

Dispose

See Figure 8-7.

For dispose (p):

1. If p is nil, report:

   *** PASCAL ERROR: DISPOSE CALLED WITH A NIL PTR IN LINE xxxx

   and abort the program.

2. Check if p is between init_toh and toh. If not, report:

   *** PASCAL ERROR: DISPOSE CALLED WITH A BAD PTR IN LINE xxxx

   and abort the program.

3. Check if the size of variable pointed to by p is equal to the size of the variable to be disposed. The size can be different if a record with variants was allocated with tag fields specified in the call to new, and one or more of the tag field values is different than the original values. In this case, report:

   *** PASCAL ERROR: DISPOSE AN INVALID VARIANT IN LINE xxxx

   and abort the program.

4. Insert the data space to be disposed into the free list of the mark region that contained the variable.

5. Set p to nil.

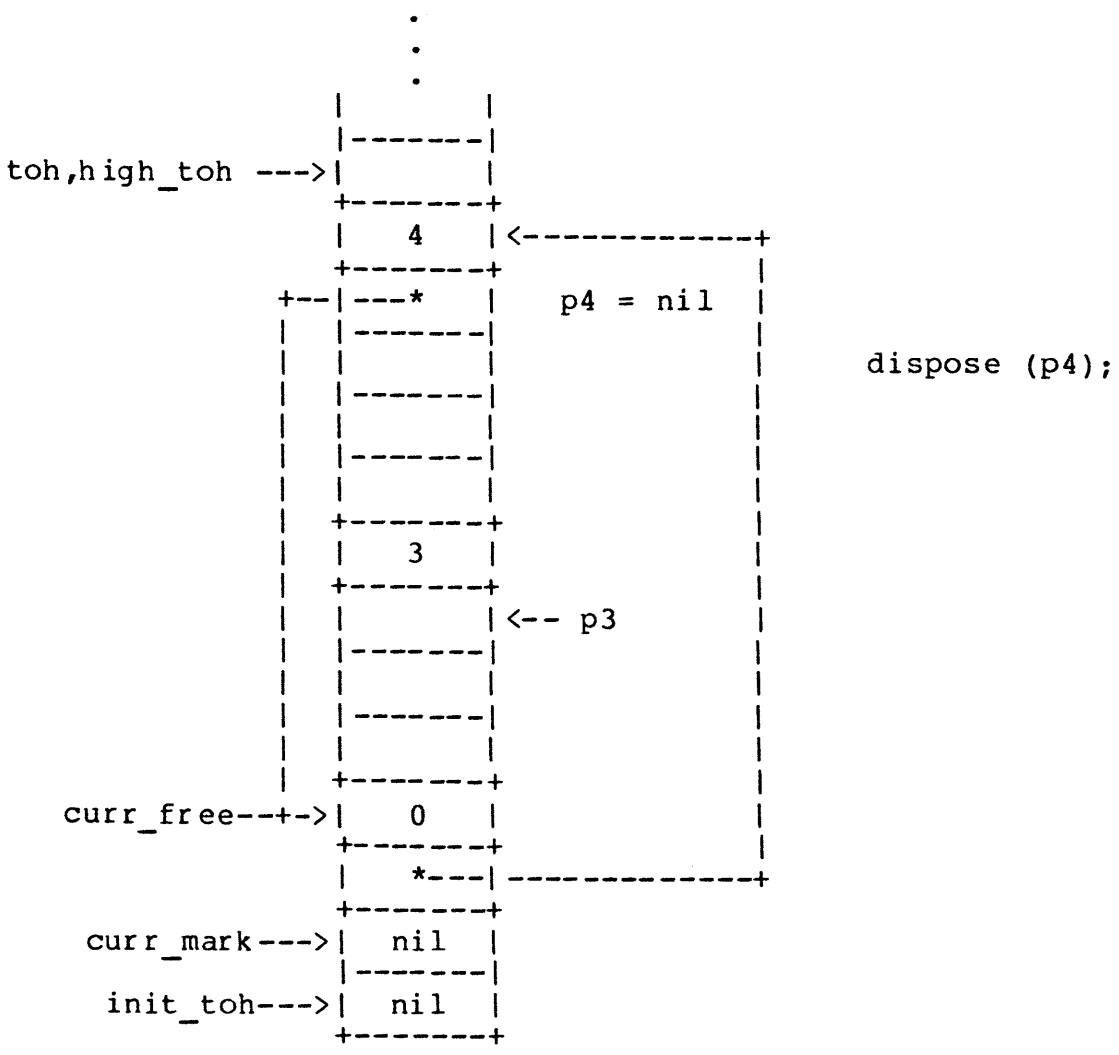


Figure 8-7. Disposing a 4-word Variable

Each free space begins with a "free block" consisting of the size of the block and a pointer to the next free block in the free list. The free list is circularly linked.

The top-of-heap pointer is not affected by dispose.

A variable is allocated in the free list as shown in Figure 8-8.

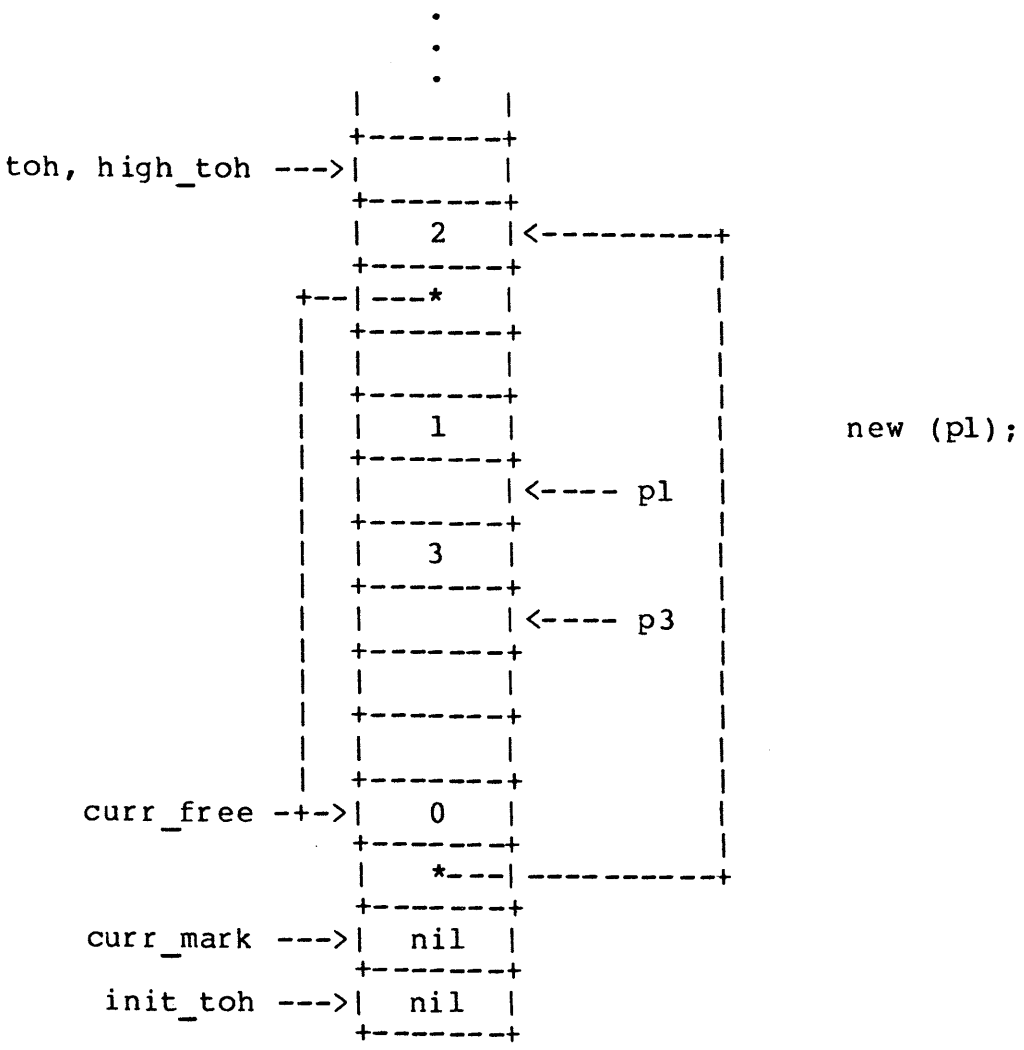


Figure 8-8. Allocation of a 1-word Variable

The remainder of the free space is left in the free list and its size is changed to the new size. If a variable the same size as a free block is allocated (e.g., if the statement above were "new (p4)") then the variable is allocated and the free space and free block are removed from the list, returning the heap to the state shown in Figure 8-6.

In general, if the remaining space in the free block is two words or longer, it is left in the free list. This is because two words are required for a free block. Thus, if a new variable of size n is allocated in a free space of size n+1 (e.g., the statement in Figure 8-8 were "new (p3)"), then the remaining word is not used and cannot be allocated.

Mark

See Figure 8-9.

For mark (p):

1. Initialize the heap if necessary.

2. Check if there is room at the top of the heap for a mark block and a dummy free block (two words each). If not, report:

   *** PASCAL ERROR: HEAP/STACK COLLISION IN LINE xxxx

   and abort the program.

3. Allocate the new mark block at the top of the heap to begin a new mark region. Set the first word to the value of the free list pointer of the previous mark region. Set the second word to point to the mark block of the previous mark region.

4. Set p to point to the new mark block.

5. Set curr_mark to point to the new mark block.

6. Allocate a dummy free block.

7. Set curr_free to point to the new dummy free block.

8. Adjust toh and, if necessary, high_toh.

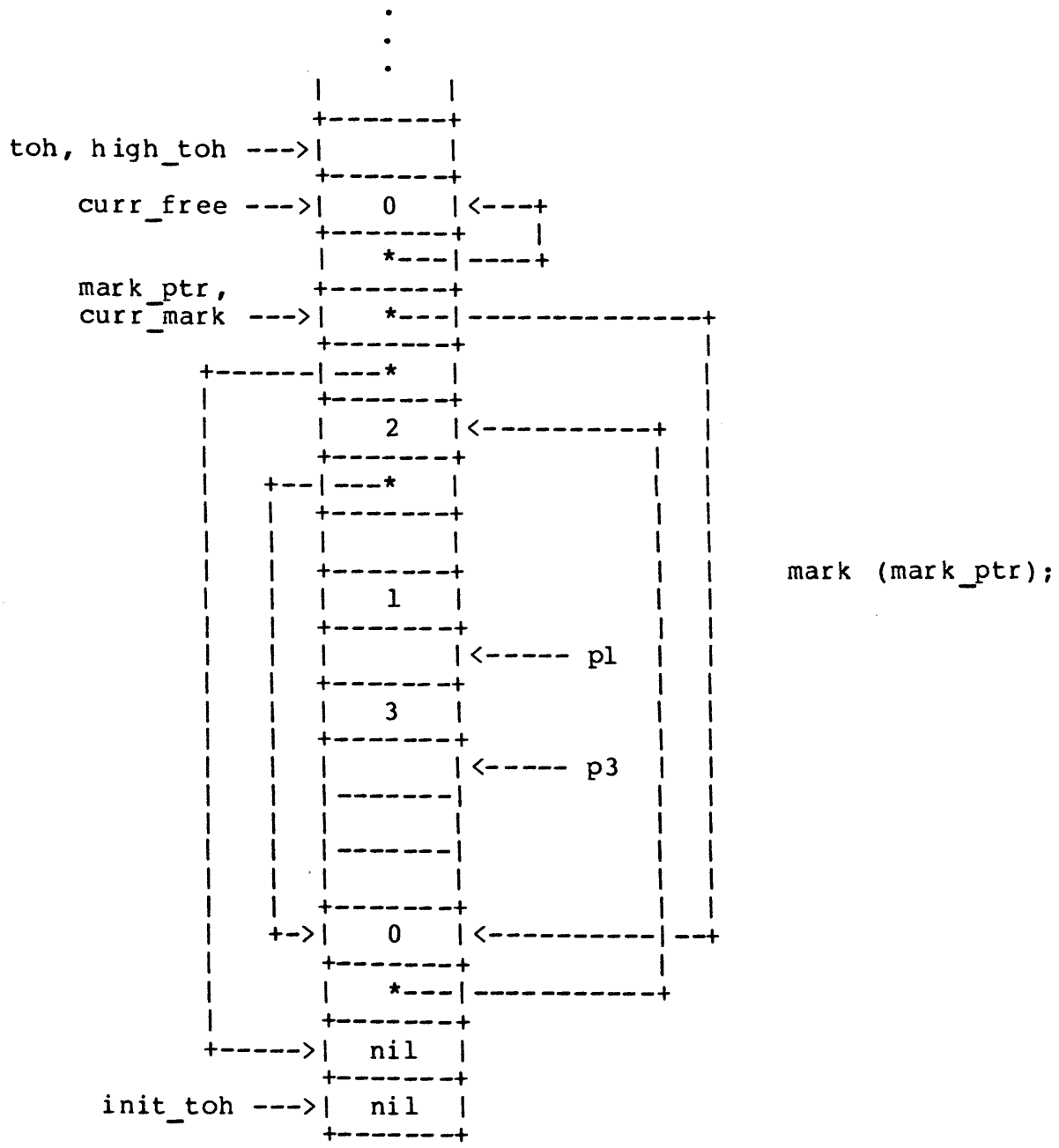


Figure 8-9. Creating a New Mark Region

Remember that new will only search the free list in the current mark region. The mark block's pointer to the free list in the previous mark region is used only by dispose and release.

Calls to new work in the same manner as before except that allocation takes place only in the new mark region.

Release

See Figure 8-10.

For release (p):

1. If p is nil, report:

   *** PASCAL ERROR: RELEASE CALLED WITH A NIL PTR IN LINE xxxx

   and abort the program.

2. Check if p is between toh and init_toh. If not, report:

   *** PASCAL ERROR: RELEASE CALLED WITH A BAD PTR IN LINE xxxx

   and abort the program.

3. Check if p points to a mark region. If not, report:

   *** PASCAL ERROR: RELEASE CALLED WITH A BAD PTR IN LINE xxxx

   and abort the program.

4. Set toh to point to the beginning of the mark region being released.

5. Set the released mark block's previous mark block pointer to curr_mark.

6. Set curr_free to point to the value of the previous free list pointer in the released mark block.

7. Set p to nil.

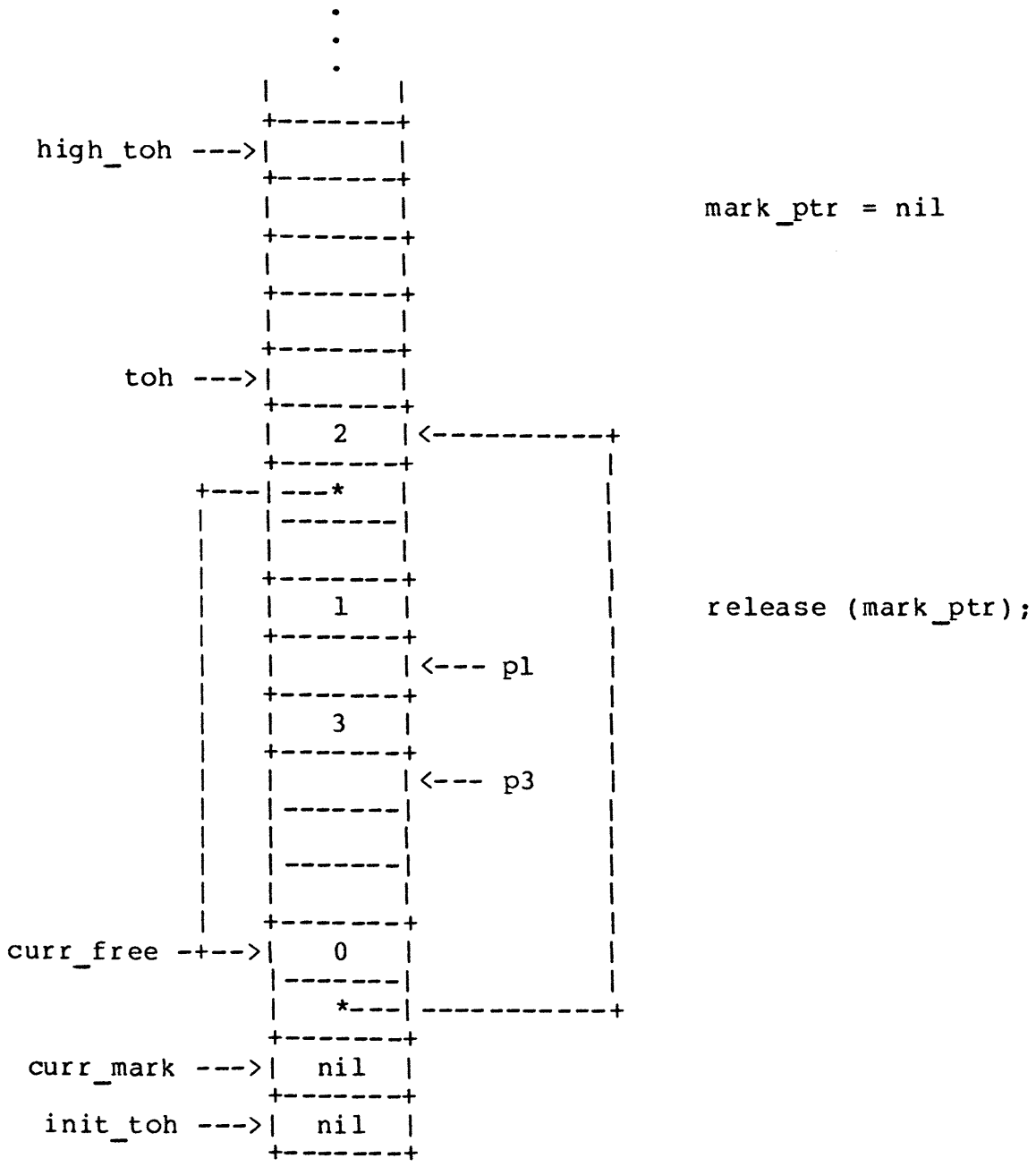


Figure 8-10. Releasing a Mark Region

Although the affect of release is to dispose all variables allocated in the mark region being released, the pointers to these variables are not set to nil as is done by dispose.

EMA Heap Management — $Heap 2$

The following changes to the heap management definitions of Figure 8-3 comprise the differences between $HEAP 1$ and $HEAP 2$ heap management routines.

```
CONST
   bsize   = 4;              {block size for $Heap 2$}
   minsize = bsize DIV 2;    {minimum allocation size}
                             {for $Heap 2$           }

TYPE
   SIZE = 0..maxint;
   ADDR = 0..maxint;

PROCEDURE get_heap_stack_info
   $ALIAS '@GHS2'$
   (VAR heap-info : INFO_REC);
   EXTERNAL;

PROCEDURE set_heap_stack_info
   $ALIAS '@SHS2'$
   (heap_info : INFO_REC);
   EXTERNAL;

PROCEDURE initialize_heap
   $ALIAS '@INH2'$;
   EXTERNAL;
```

Pointers are represented as double word integers. Thus, the block size is four words, and the minimum allocation size is 2 words. The minimum allocation size is the size of the smallest data block that, when disposed, can be inserted into the free list.

The preceding descriptions of heap management procedures apply for $HEAP 2$ if in the figures and text, the sizes of pointers, blocks, and minimum allocation are doubled.

Short Versions of Heap Management Routines

An alternate set of heap management routines is provided. These routines are shorter than the standard procedures and are for use in applications where the memory reclamation features of new and dispose are not needed. No free lists are maintained by the short routines, and thus no data block is allocated for each heap variable. These routines can be used with either the $HEAP 1$ or $HEAP 2$ compiler option.

To use the short heap management routines, the library %SHSLB must be searched at load time before the standard Pascal/1000 library $PLIB is searched.

Example:

```
RE,%PROG
SE,%SHSLB
SE,$PLIB
EN
```

Figure 8-11 shows the definition of type INFO_REC as used by the short routines in calls to get_heap_stack_info, set_heap_stack_info, and initialize_heap_stack_info.

```
TYPE
   ADDR = 0..32767;              {for Heap 2, ADDR = 0..maxint}

   INFO_REC =
      RECORD
         tos,                    {top of stack       }
         toh,                    {top of heap        }
         init_tos,               {initial tos        }
         init_toh,               {initial toh        }
         high_tos,               {high tos           }
         high_toh,               {high toh           }
         dummy,                  { not used          }
         curr_mark: ADDR;        { ^curr.mark block  }
      END;
```

Figure 8-11. Definitions Used by Short Heap Management Routines

Heap initialization is not necessary and is not performed by these procedures.

For the short version of NEW (p):

1. If toh - (size of new variable) <= tos then report:

   *** PASCAL ERROR: HEAP/STACK COLLISION IN LINE xxxx

   and abort the program. Otherwise, set toh to the value of

   toh - (size of new variable)

   and set p to the value of

   toh + 1

For the short version of DISPOSE (p):

1. If p is nil then report:

   *** PASCAL ERROR: DISPOSE CALLED WITH A NIL PTR IN LINE xxxx

   and abort the program.'

2. If p < toh then report:

   *** PASCAL ERROR: DISPOSE CALLED WITH A BAD PTR IN LINE xxxx

   and abort the program.

3. Otherwise, set p to nil.

For the short version of MARK (p):

1. Set p to the value of toh.

For the short version of RELEASE (p):

1. If p is nil then report:

   *** PASCAL ERROR: RELEASE CALLED WITH NIL PTR IN LINE xxxx

   and abort the program.'

2. If p < toh then report:

   *** PASCAL ERROR: RELEASE CALLED WITH A BAD PTR IN LINE xxxx

   and abort the program.

3. Otherwise, set toh to the value of p, and set p to nil.

Efficiency Considerations

This section lists some considerations about the effects of Pascal/1000 language constructs on program space and execution time.

The programmer usually has many implementation decisions to make, some of which have to do with programming style, and deciding which is the clearest and most logical way to do something. These considerations may be helpful when two methods appear equally appropriate, or efficiency is particularly important. Also, in some cases (structured constants for example) there is a trade-off between efficiency and transportability.

Data Access

This section describes the efficiency of various methods of accessing data.

Accessing Variables and Parameters

Variables and parameters may be accessed directly, indirectly through a one-word address, or indirectly through a two-word address. Direct addressing of data is the most efficient in time and space, while indirect accessing through a two-word address is least efficient.

Table 8-4 shows the ways in which variables and parameters are accessed.

Table 8-4. Pascal/1000 Variable and Parameter Access.

CLASS OF DATA	ACCESS	EXAMPLE
Static variables		
Global	direct	LDA g
Local	direct	LDA l
Non-local	direct	LDA nl
Dynamic variables		
Heap (1)	short,I	LDA p,I
Heap (2)	long,I	JSB .LBPR
		DEF p
		LDA B,I
Parameters		
Value	direct	LDA v
Reference (Heap 1)	short,I	LDA r,I
Reference (Heap 2)	long,I	JSB .LBPR
		DEF r
		LDA B,I

Legend:

direct - variable is accessed directly.
short,I - variable is accessed indirectly through a one-word address.
long,I - variable is accessed indirectly through a two-word address.

Passing Parameters

Table 8-5 illustrates parameter passing and accessing in more detail.

Some considerations:

- Value parameters can use a costly amount of space since copies are made. This is especially important when passing large amounts of data.

- Accessing variable parameters may take more time than accessing value parameters.

- In a HEAP 2 program where no actual parameter corresponds to a VAR formal parameter, turning HEAPPARMS OFF will save access time and code space.

- If an actual string parameter and its corresponding formal parameter differ in length, or if one is packed and the other unpacked, then the actual parameter is converted into a variable of the type of the formal parameter. This is less efficient than passing a string whose type matches the formal parameter.

Table 8-5. Pascal/1000 Parameter Passing and Access

Parameter Type:		Calling Action:	Passed As:	Routine Action:	Routine Access:	Exit Action:
Heap 1	Value	cvtl	short	ctl,rc	direct	none
	Var:N	none	short	none	short,I	none
	:R	none	short	al	short,I	rad
	:M	none	short	none	short,I	none
	Func result	none	short	none	short,I	none
Heap 2	Value:1	cvtl	short	ctl,rc	direct	none
	:2	ctl	short	ctl,rc	direct	none
	Var:1N	none	stlm	c2wa	long,I	none
	:1R	none	stlm	c2wa,a2	long,I	rad
	:2N	none	stl	c2wa	long,I	none
	:2R	none	stl	c2wa,a2	long,I	rad
	:MN	flwa	short	none	short,I	none
	:MR	flwa	short	m2wa,a2	long,I	rad
	Func Result:N	none	short	none	short,I	none
	:R	none	short	m2wa,a2	long,I	rad

Legend:

Heap 1 - program is compiled with the $HEAP 1$ option
Heap 2 - program is compiled with the $HEAP 2$ option

Value - formal parameter is a value parameter
Var - formal parameter is a VAR parameter
Func - Called routine is a function whose result is >2 words

short - 1-word address of actual
long - 2-word address of actual
stl - 1-word address of a 2-word address of actual
stlm - 1-word address of a 2-word address whose 1st word is -1
direct - variable is accessed directly
short,I - access to actual is indirect thru a 1-word address
long,I - indirect thru the 1-word address result of .MAPR
flwa - find 1-word address (using .MAPR if necessary)
cvtl - convert to a local w/ formal's type if necessary;pass local

Conversion is done into a temporary variable when the

actual is:	and formal is:
single int	double int
packed string	unpacked string
unpacked string	packed string
short string	long string
bit or byte field of a packed structure	corresponding type

ctl - copy actual to local
rc - range check actual
c2wa - copy 2-word address of actual to a local
m2wa - make 2-word address from 1-word address (first word is -1)
al - adjust 1-word address to point into stack if self-pointing
a2 - adjust 2-word address to point into stack if self-pointing
rad - re-adjust address if self-pointing
:1 - actual is accessible with a 1-word address
:2 - actual is accessible with a 2-word address
:M - formal is a non-heap var parm ($HEAPPARMS OFF$)
:N - called routine is non-recursive
:R - called routine is recursive

Packed vs. Unpacked Data

In general, packing a Pascal array or record results in:

- less space allocated for the structure
- slower access of some or all elements of the structure
- more code to access some or all elements of the structure

The space and time differences vary according to the size and alignment conditions of the elements, and whether the elements are accessed with constant or variable offsets. For example:

- compared to a 16-bit integer element, the savings in space of an element of type 0..32767 (15 bits) is not as great as the savings for an element of type 0..1 (1 bit).
- the access time for an element of a packed structure can be significantly greater if accessed with a variable offset, that is, an offset that must be computed at run time. A typical example is an array subscripted with an expression.
- the access time for an element of a packed structure may only be slightly greater if it is accessed with a constant offset. Record fields and arrays subscripted with constant expressions are examples of constant-offset access.

Table 8-6 summarizes the accessing of packed and unpacked data.

Table 8-6. Packed and Unpacked Data Access

	Variable Offset	Constant Offset
Unpacked	w-inline	direct
Packed		
word	w-inline	direct
byte	by-inline	by-inline
bit	wa-inline and library	wa-inline and b-inline

Legend:

word	- element of a packed structure that is word-aligned and fills an entire word (or double-word).
byte	- element of a packed structure that is byte-aligned and occupies 8 bits.
bit	- element of a packed structure occupying n bits that can not be classified as either a "word" or a "byte".
direct	- element is accessed directly. No address computation is necessary.
w-inline	- inline code to access the element.
wa-inline	- inline code to calculate the word address of the element.
by-inline	- inline code to access the byte field using byte instructions.
b-inline	- inline code to extract or deposit the bit field.
library	- a call to a library routine which extracts or deposits a bit field from a particular word (or double-word).

Example of packed and unpacked data access:

```
 1 00000 PROGRAM pack;
 2 00000 TYPE
 3 00000    RANGE = 0..3;
 4 00000
 5 00000    { Declare Packed and Unpacked Array types }
 6 00000    INDEX = 1..16;
 7 00000    ARR = ARRAY [INDEX] OF RANGE;  {16 1-word elements}
 8 00000    PACKED_ARR = PACKED ARRAY [INDEX] OF RANGE;
 9 00000                                        {16 2-bit elements }
10 00000    { Declare Packed and Unpacked Record types }
11 00000    REC = RECORD
12 00000       first, {8 1-word fields}
13 00000       second,
14 00000       third: RANGE;
15 00000       ar: ARRAY [1..5] OF RANGE;
16 00000    END;
17 00000
18 00000    PACKED_REC = PACKED RECORD
19 00000       first,
20 00000       second, {8 2-bit fields}
21 00000       third: RANGE;
22 00000       ar: PACKED ARRAY [1..5] OF RANGE;
23 00000    END;
24 00000
25 00000 VAR                        {    Allocations:    }
26 00000    a:  ARR;                { 16 words           }
27 00020    pa: PACKED_ARR;         {  2 words           }
28 00022    r:  REC;                {  8 words           }
29 00032    pr: PACKED_REC;         {  1 word            }
30 00033    i:  INDEX;
31 00034    x:  RANGE;
32 00035
33 00035 BEGIN    $LIST_CODE, RANGE OFF$
34 00003    {Variable offsets for unpacked structures}
35 00003    x := a [i];
                00003      LDA .6+0
                00004      ADA @1+27
                00005      LDA A.,I
                00006      STA @1+28
36 00007
```

```
37 00007    {Variable offsets for packed structures}
38 00007    x := pa [i];
                00007       CCA
                00010       ADA @1+27
                00011       STA @1+29
                00012       JSB @XTR1
                00013       DEF *+6
                00014       DEF .5+0
                00015       DEF @1+16
                00016       DEF @1+29
                00017       DEF .5+1
                00020       DEF .5+1
                00021       STA @1+28
39 00022
40 00022
41 00022    {Constant offsets for unpacked structures}
42 00022    x := a [7];
                00022       LDA @1+6
                00023       STA @1+28
43 00024    x := r.second;
                00024       LDA @1+19
                00025       STA @1+28
44 00026    x := r.ar [5];
                00026       LDA @1+25
                00027       STA @1+28
45 00030
46 00030    {Constant offsets for packed structures}
47 00030    x := pa [7];
                00030       LDA @1+16
                00031       LSR 2
                00032       AND =D3
                00033       STA @1+28
48 00034    x := pa [8];
                00034       LDA @1+16
                00035       AND =D3
                00036       STA @1+28
49 00037    x := pr.second;
                00037       LDA @1+26
                00040       LSR 12
                00041       AND =D3
                00042       STA @1+28
50 00043    x := pr.ar [5];
                00043       LDA @1+26
                00044       AND =D3
                00045       STA @1+28
51 00046 $LIST_CODE OFF$
52 00046 END.


          0 Errors detected.
         52 Source lines read.
         79 Words of program generated.
```

Heap 1 vs. Heap 2

Under Heap 2 the access times are slower and more code is emitted to access:

- dynamic variables.
- VAR parameters, except for non-resursive routines with $HEAPPARMS OFF$.
- "Large" function results for recursive functions (large meaning >2 words).

All address calculations for heap items are done using double integer arithmetic routines, which may be in firmware or software. The software routines are slower and occupy space in the loaded program.

EMA-addressing routines are loaded with the program, which may be either firmware or software. The software routines are slower and occupy space in the loaded program.

Heap 2 Heap/Stack routines are larger than the Heap 1 versions.

Recursion is slower for Heap 2 since the stack resides in EMA. Stack access is done with double integer arithmetic and EMA addressing routines.

There can be much more memory available for heap and stack with Heap 2.

Heap 2 - amount of EMA space in EMA partition (up to 1.8MB).

Heap 1 - amount of memory left between the end of the program and the end of the partition (64KB - op sys - program size).

EMA partitions are generally scarcer than normal partitions. The swap time on EMA programs increases with the size of the partition, and several programs may be swapped out when an EMA program is swapped in.

Heap 0 (with RECURSIVE OFF) suppresses the invocation (and loading) of the Heap/Stack initialization routine.

Expressions

This section describes the efficiency of certain expression evaluations.

Partial Evaluation

- More efficient in space and time than full Boolean evaluation.

Common Subexpressions

- The compiler does not eliminate common subexpressions. The programmer can often do so by using his own temporary variables to save intermediate results that are used in several places.

- Some common subexpressions, those involving the re-calculation of record addresses, can be easily eliminated by using the WITH statement.

Numeric Data Types

- Using single-word instead of double-word integers where possible can greatly improve the performance of your program.

- Using exact subranges instead of the full 1-word subrange can save time in various operations. Single-word comparisons, for example, do not need to check for overflow when comparing two variables of type 0..10. Overflow must be checked when comparing two variables of type -32768..32767. Subranges can also be represented in less space when they are used inside packed structures.

- Using REAL variables is usually faster than using LONGREAL variables. LONGREALs should be used when a precision greater than that provided by the type REAL is needed.

Range Checking

- Range checking (compiler option RANGE) is extremely useful in debugging a program. It also adds a significant amount of overhead that you may eventually want to eliminate when your program is more stable.

- Expressions of the following types are range checked:

  enumeration types
  subrange types
  CHAR
  BOOLEAN
  pointer types

- Expressions are range checked during the following operations:

  assignments
  array indexing
  pointer dereferencing
  copying of a value parameter into a local variable
  FOR statement (initial value and each successive value)

Sets

- Sets that fit in one word (16 elements or less) are much more efficient than multi-word sets. Set operations are performed with inline AND's, IOR's, etc. instead of by library routines.

- Members of sets and packed Boolean arrays are accessed almost identically, so effectively the only difference in efficiency between the two is the fact that multi-word sets have an extra "length" word allocated.

Statements

WITH

- The WITH statement can be used to avoid the repeated calculation of a record's address when more than one field of the record is to be accessed. The resulting amount of savings can be quite significant depending on the number of calculations that were avoided.

- Sometimes the WITH statement does not save time or space (although it still saves typing the record's name for each field). This is the case when there is no address calculation to be done for the record. For example the WITH statements below do not provide any greater efficiency to the program:

```
WITH r DO <statement>;
WITH p^ DO <statement>;
```

  These do:

```
WITH p^.r DO <statement>;
WITH p^.a[i] DO <statement>;
```

FOR

- FOR loops using single-integer control variables are much faster than double-integer for loops. Range checking is done for both kinds of loops not only for the initial value of the control variable, but for each iteration of the loop. Thus there is a great difference between the speed of a range-checked FOR loop and one that does not check.

CASE

- At the end of the CASE statement, the compiler emits a section of code that is used in deciding which case is to be selected and executed. This section of code consists of a combination of the following constructs, depending on which are the most efficient for the particular cases being considered:

    - Element-by-element comparisons
    - Jump table
    - Interval test

  The CASE statement is an unordered selection process, i.e. the order in which the comparisons is done is unspecified. Thus you should never rely on the comparisons being done in any particular

order. If a specific order is desired, IF statements are more appropriate.

Procedures and Functions

This section describes ways to make procedures and functions more efficient.

Recursion

Recursive routines are more costly in terms of space and execution time than non-recursive routines.

Space Considerations

- Recursive routines are allocated a few more words of space used to save the environment of a current activation.

- Extra code from the run-time library is loaded with the program to handle recursion. (This code is always the same size, regardless of the number of resursive routines in the program.)

- If recursion is off and the heap option is set to 0 for the entire program, then the Heap/Stack initialization routine is not called, and thus not loaded with the program.

- Recursive activations of routines cause the stack to grow, thus reducing the amount of space available in the heap.

Time Considerations

- Routine activation methods can be divided into four categories according to the overhead for each method. In order of decreasing speed, they are:

    A) Direct activation of a non-recursive routine
    B) .ENTR activation of a non-recursive routine
    C) Non-recursive activation of a recursive routine
    D) Recursive activation of recursive routine

A) The amount of overhead time required for a routine with a direct calling sequence is less than its .ENTR equivalent when the number of parameters is small (see below for more on Direct calling sequences). This is because parameter address resolution is done as inline code rather than in microcode (.ENTR may also be in software, in which case the difference is even greater). Recursive routines cannot have direct calling sequences.

B) The amount of overhead time required for a non-recursive routine which uses .ENTR can be less than a non-recursive activation of a recursive routine (first time into the routine) because, even though they perform roughly the same job, .ENTR is often microcoded.

C&D) A non-recursive activation of a recursive routine is faster than a recursive activation of the same routine because its activation record need not be stacked upon routine entry and unstacked upon routine exit.

Direct Calling Sequence

The time overhead for routines that are called frequently can be significantly reduced by employing the DIRECT compiler option. This option, which is only allowed on non-recursive routines, causes parameter addresses to be resolved by inline code rather than by a call to the routine .ENTR.

The savings is substantial for routines having a small number of parameters, but above a certain number of parameters, it is faster to use .ENTR. The cross over point, for microcoded .ENTR, usually occurs when the number of parameters is small, as shown in the table below.

Table 8-7. Overhead Times for Routines with .ENTR vs. $DIRECT$ Calling Sequences (microseconds)

Number of Parameters	.ENTR	$DIRECT$			
		Indirect levels:			Words:
		0:	1:	2:	
0	10.1	1.5	1.5	1.5	1
1	12.6	5.6	9.0	12.4	6
2	15.1	9.7	16.5	23.3	11
3	17.6	13.8	24.0	34.2	16
4	20.1	17.9	31.5	45.1	21
5	22.6	22.0	39.0	56.0	26
6	25.1	26.1	46.5	66.9	31
7	27.6	30.2	54.0	77.8	36

.ENTR overhead =
    10.2 + (number of parameters * 2.5) microseconds
    code space = 2 words

$DIRECT$ overhead =
    1.5 + (number of parameters * (4.1 + indirect levels * 3.4))
    code space = 1 + (number of parameters * 5) words

FMP vs. Pascal/1000 I/O

- Roughly equivalent in speed.

- Pascal I/O is less space-efficient because of the library routines that are loaded with the program.

- The BUFFERS option can be used to increase I/O performance.

- The LINESIZE option can be used with text files to decrease the size of the buffer(s) allocated for the file.

NOTE: If FMP calls are used to access a Pascal file, the following should be taken into consideration:

  - For the DCB parameter of the FMP call, specify the Pascal file variable.

  - Pascal-managed file status information is not updated when using FMP calls directly. This status information is used by Pascal file-handling routines.

Reducing the Size of a Loaded Program

Short Versions of Library Routines

- The program can be loaded with the short version of the run-time error reporter. This will not print out the long error messages, but will instead print only error numbers. The name of this module is %PRERS and it must be relocated in the program before the library $PLIB is searched.

- A short version of the heap management routines, in the file $SHSLB, can be used to save space in your loaded program. It is smaller because it does not do any linking of free lists or mark blocks. A call to the procedure dispose serves only to set that pointer to nil, and does not free up the space associated with it. Thus if you do not use dispose, or you use it but do not rely on reclaimed space, then the short Heap/Stack library can be used. (Also, a certain amount of savings of space in the heap can be realized since a header is not allocated for each individual heap object). Refer to Data Management in this Chapter for more information about Heap/Stack routine implementation.

Using Segmentation to Save Space

- Initialization of globals and the heap can be done in a segment. The price paid for the segment load is often negligible compared to the savings that can be gained in code space.

- If all files are opened in a segment, then the Pascal and FMP routines required to open files are loaded only in that segment. Remember, however, that files INPUT and OUTPUT (if specified in the program heading) are automatically opened in the main. Thus, in order to realize this savings, the program must use other file names.

- The run-time error routine can be loaded in a segment. A segment load is then required in order to report a run-time error. Refer to Run-Time Errors (Appendix C) for details.

Putting Globals in the Heap

- Large or infrequently-used globals in Heap 2 programs can be put in the heap. This makes more room for code in the logical address space.

Structured Constants

Using structured constants can save time and space in several ways:

- No execution time is spent setting up a structured constant as is the case for structured variables which are initialized at run time. The amount of space required for a structured constant is the same as that required for a structured variable. In addition, structured constants declared in a recursive routine are not copied to and from the stack for recursive activations of the routine, saving both stack space and execution time.

- Rather than initializing element by element as required by Standard Pascal, structured constants can be used to quickly initialize a structured variable. If the structure is large, it is often best to declare the constant and perform the initialization inside of a segment, so that you do not pay a large space penalty for the existence of both a variable and a constant structure.

- Structured constants can improve on CASE statements which serve to map one data type onto another. For example, the functions below are equivalent:

```
TYPE
  COLOR = (red, blue);
  SHADES = (red,blue,purple);
```

```
FUNCTION shade
  (color1,
   color2: COLOR): SHADES;
BEGIN
  CASE color1 OF
    red:
      CASE color2 OF
        red:  shade := red;
        blue: shade := purple;
      END;

    blue:
      CASE color2 OF
        red:  shade := purple;
        blue: shade := blue;
      END;
  END;
END;
```

```
FUNCTION shade
  (color1,
   color2: COLOR): SHADES;

TYPE
  ROW = ARRAY [COLOR] OF SHADES;

  TRANS_TABLE =
    ARRAY [COLOR] OF
      ROW;

CONST
  table = TRANS_TABLE
    [ROW [red, purple],
     ROW [purple, blue]];

BEGIN
  shade := table [color1, color2];
END;
```

Chapter 9
How To Use Pascal/1000

The first sections of this chapter describe how to compile, load, and run Pascal/1000 programs. The next sections contain information on error analysis and some of the debugging tools available to the programmer. The final sections describe the use of various features of Pascal/1000, including interaction of Pascal with non-Pascal routines and the IMAGE/1000 Data Base Management System, and using EXEC calls in a Pascal program.

Compiling A Program

The Monitor

The program PASCL serves as the monitor for a Pascal compilation. Its functions are to:

1) Parse and verify the correctness of the parameters in the run string.

2) Schedule the Pascal compiler PCL.

3) Schedule the Pascal cross-referencer PXREF if the compiler option XREF is specified.

4) Schedule the assembler ASMB to generate relocatable code from the assembly code created by the compiler if the CODE compiler option is ON and there are no compilation errors.

5) Save or purge the assembly file, depending on the following:

   - If the KEEPASMB compiler option is not specified and there are no assembly errors, the assembly file is purged.

   - Otherwise, the assembly file is saved.

   Note that the assembly file is not created by the compiler if the CODE compiler option is OFF.

PASCL is run with the following command:

```
:RU,PASCL,[<source>],[<list>],[<relocatable>],[<option>]
```

where:	is:
<source>	Namr of the source file. If it is not specified, LU 1 is used.
<list>	Namr of the list file. If it is not specified or is 0, the listing is suppressed, and only the source lines containing errors will be output to LU 1. The initial heading is also printed on LU 1.
<relocatable>	Namr of the relocatable file. If it is not specified or is 0, code generation is suppressed.
<option>	Namr of the option file which contains the number of pages of EMA to be used for the compiler workspace. If it is not specified or is 0, a default number (15) sufficient to compile most small programs is used. If it is 1, the following prompt is issued at LU 1 for the number: /PASCL: Enter options: Type in the number of pages of EMA needed.

The namr follows the RTE file naming format. A namr may consist of a file name, security code, cartridge name, etc., or it may be a logical unit number. (Refer to Terminal User's Reference Manual.)

The run strings

```
:RU,PASCL,&TEST

:RU,PASCL,&TEST,0

:RU,PASCL,&TEST,0,0
```

are equivalent. They specify that the source file name is &TEST, that the listing is to be suppressed (equivalent to $LIST OFF$), and that code generation is to be suppressed (equivalent to $CODE OFF$). In other words, the source is to be scanned for compile-time errors only, and source lines with errors are to be output to LU 1. This run string is useful for fast detection of compilation errors, as the compiler runs much faster when it is not generating code.

The run strings

```
:RU,PASCL

:RU,PASCL,1

:RU,PASCL,1,0

:RU,PASCL,1,0,0
```

are equivalent. They specify that the source is to be accepted a line at a time from LU 1. Each line is scanned for compile-time errors only. Neither a listing nor code is generated.

If COMPL or CLOAD is used to compile and/or load a Pascal program, the first line of the source must begin with '$PASCAL'.

It is recommended that the file names begin with the following standard key characters:

File	Key	Example
source	&	&TEST
list	'	'TEST
relocatable	%	%TEST
option	*	*TEST

The dash ("-") may be used in place of the list, relocatable, and/or option namr in the run string. PASCL will construct the appropriate name based on the source file name. The construction rules are:

1) If the source file name begins with the standard key character "&", then the name constructed will be the source file name with the "&" replaced by the appropriate standard key character.

   Example:

   ```
   :RU,PASCL,&MERGE::CR,-,-,-
   ```

   is equivalent to:

   ```
   :RU,PASCL,&MERGE::CR,'MERGE::CR,%MERGE::CR,*MERGE::CR
   ```

   Note:

   It is not necessarily equivalent to

   ```
   :RU,PASCL,$MERGE::CR,'MERGE,%MERGE,*MERGE
   ```

   since CR may not be the first available cartridge.

2) If the source file name does not begin with the standard key character "&", then the name constructed will be the source file name with the appropriate standard key character appended at the left.

Example:

```
:RU,PASCL,PROGRM::RM,LIST,-,-
```

is equivalent to:

```
:RU,PASCL,PROGRM::RM,LIST,%PROGR::RM,*PROGR::RM
```

The name of the assembly file is constructed from the relocatable namr (not the source file namr) using the above construction rules, with "^" as the standard key character.

Examples:

Relocatable file name	Assembly file name
%TEST	^TEST
RELOCA	^RELOC

The run string:

```
:RU,PASCL,&FCOPY,-,-
```

produces the assembly file ^FCOPY. This file is saved or purged after the assembly, as described above.

File Verification

The monitor PASCL verifies the correctness of the files specified in the run string before scheduling the compiler PCL. The following criteria are used:

Source file

- FMP must be able to open the file, otherwise the following message is output to LU 1, and PASCL is terminated.

  /PASCL: Source file: FMP error code = <code>

- The file must be of type 3 or 4, otherwise the following message is output to LU 1 and PASCL is terminated.

  /PASCL: Invalid source file: <name>

  where <name> is the source namr as it appeared in the run string.

- If the file name does not include the cartridge reference, the two-character or integer reference will be added to the name.

List file

- If the file already exists, it must be of type 3 or 4, and it must not be the same file as the source file, otherwise the following message is output to LU 1, and PASCL is terminated.

  /PASCL: Invalid list file: <name>

  where <name> is the list namr as it appeared in the run string.

- If the file already exists, it will be overlaid by the new listing.

Relocatable file

- If the file already exists, it must be of type 5, and it must not be the same file as the source file, otherwise the following message is output to LU 1, and PASCL is terminated.

  /PASCL: Invalid relocatable file: <name>

  where <name> is the relocatable namr as it appeared in the run string.

- If the file already exists and its name begins with the standard key character "%", the file will be overlaid by the new relocatable output. If the file already exists but its name does not begin with "%", the file will be purged and a new file (of type 5) will be created with the same name.

Option file

- The file must already exist and be of type 3 or 4, and it must not be the same file as the source file, otherwise the following message is output to LU 1, and PASCL is terminated.

  /PASCL: Invalid option file: <name>

  where <name> is the option namr as it appeared in the run string.

Assembly file

- If the file already exists, it must be of type 3 or 4, and it must not be the same file as the source file, otherwise the following message is output to LU 1, and PASCL is terminated.

  /PASCL: Invalid assembly file: <name>

  where <name> is created from the relocatable namr.

- The following message is output to LU 1 if the assembly file is kept.

  /PASCL: Assembly file kept: <name>

  where <name>, including cartridge reference is created from the relocatable namr.

Scheduling Messages

The following message is output to LU 1 if the cross-referencer is scheduled.

/PASCL: PXREF scheduled.

The following message is output to LU 1 if the assembler is scheduled.

/PASCL: ASMB scheduled.

If an error occurs when PASCL attempts to schedule PCL, PXREF, or ASMB, or if the scheduled program is aborted, then the appropriate one of the following messages is output to LU 1:

```
/PASCL: PCL schedule error code = <code>

/PASCL: PXREF schedule error code = <code>

/PASCL: ASMB schedule error code = <code>
```

After one of the above messages, one of the following messages is output to LU 1, depending on the value of <code>:

CODE	MESSAGE
1	/PASCL: Duplicate program name.
2	/PASCL: No ID segments available.
3	/PASCL: Program not found.
4	/PASCL: Program file open error.
5	/PASCL: Program file close error.
6	/PASCL: RP error.
7	/PASCL: Program busy.
8	/PASCL: Program aborted.
9	/PASCL: Insufficient system memory.

Example:

The following is an example of the output to LU 1 when PASCL is run. The source program in file &TOWER contains the compiler option $KEEPASMB$.

```
:RU,PASCL,&TOWER,-,-

                0 Errors detected.
               29 Source lines read.
              667 Words of program generated.
/PASCL: ASMB scheduled.
/ASMB:   NO ERRORS TOTAL
/ASMB: $END
/PASCL: Assembly source kept: ^TOWER::RM
```

Note that the cartridge reference RM was added to the assembly source file name.

Insufficient Workspace

The compiler will abort if it runs out of workspace, issuing the message:

```
/PASCL: Insufficient Workspace.
```

and either the message

```
/PASCL: Increase EMA pages in option file to at least n.
```

where n is the minimum number of pages which will change the amount of workspace allocated (the program may need even more in order to compile),

OR the messages

```
/PASCL: Maximum of n EMA pages are already allocated for workspace.
/PASCL: Partition not large enough to compile this program.
```

If the last two messages appear, the compiler is running in a partition that is too small , for one of two reasons:

1. The SZ command (see the RTE-IVB Terminal User's Reference Manual) was used to make the compiler size smaller than that of the partition in which it is actually running. In this case, the SZ command can be used to increase the compiler size, allowing it to run in a partition of the new size or larger.

2. The compiler is already the size of the largest partition. In this case, the system must be reconfigured with larger partitions and the compiler sized up appropriately.

   For more information, see the STATS option (Appendix D) and the Configuration Guide.

Listing

The listing produced by the Pascal/1000 compiler for the source program below appears on the next page. (The program contains errors.)

```
PROGRAM tower;

TYPE
   INT = 0..100;

VAR
   n: INT;
   f1, f2: TEXT;

PROCEDURE hanoi (n:INT; s, d, i: CHAR);
  BEGIN {hanoi}
     IF n > 0 THEN BEGIN
        hanoi(n - 1, s, i, d);
        writeln(f2, ' Move disk', n:3, ' from ', a, ' to ', b);
        hanoi(n - 1, i, d, s);
     END;
  END END; {hanoi}

BEGIN {tower}
  reset(f1, '1');
  rewrite(f2, '1');
  writeln(f2, ' Number of disks? ');
  read(f1, m);

  writeln(f2, ' Tower of Hanoi solution for', m:3, 'disks');
  writeln(f2);

  hanoi(n, 'A', 'C', 'B');
END. {tower}
```

```
   Pascal/1000                                  Wed Apr  2, 1980   2:43 pm
   Ver. 1/2015                                  &TOWER::JA         Page 1

    1 00000 PROGRAM tower;
    2 00000
    3 00000 TYPE
    4 00000    INT = 0..100;
    5 00000
    6 00000 VAR
    7 00000    n: INT;
    8 00001    f1, f2: TEXT;
    9 00677
   10 00677 PROCEDURE hanoi (n:INT; s, d, i: CHAR);
   11 00000   BEGIN {hanoi}
   12 00002      IF n > 0 THEN BEGIN
   13 00006         hanoi (n - 1, s, i, d);
   14 00017         writeln(f2,'Move disk',n:3,' from ',a,' to ',b);
0    ****                                                ^104     ^104
   15 00017         hanoi(n - 1, i, d, s);
   16 00017      END;
   17 00017   END END; {hanoi}
14   ****          ^14,6
   18 00017
   19 00017 BEGIN {tower}
   20 00017   reset(f1, '1');
   21 00017   rewrite(f2, '1');
   22 00017   writeln(f2, ' Number of disks? ');
   23 00017   read(f1, m);
17   ****             ^104
   24 00017
   25 00017   writeln(f2,'Tower of Hanoi solution for', m:3,'disks');
23   ****                                               ^104
   26 00017   writeln(f2);
   27 00017
   28 00017   hanoi(n, 'A', 'C', 'B');
   29 00017 END. {tower}


            6 Errors detected (first 14/last 25).
           29 Source lines read.
           15 Words of program generated.


    Errors in this compilation:

      6: illegal symbol
     14: ';' expected
    104: identifier not declared
```

In the listing a two-line heading appears at the top of each page. Beneath 'Pascal/1000' is the version of the Pascal/1000 compiler which produced the listing. The date and time at which the listing was created appears in the first line. Beneath the date is the name of the source file entered in the run string. The cartridge reference will be printed whether or not it appeared in the run string. The security code will always be omitted. Beneath the time is the page number of the listing.

Each line of the source file is echoed on the list file. The numbers in the leftmost column are the source file line numbers. The numbers to their right are dependent on whether the source line is part of a declaration section or body. In the declaration section of a program (or routine), the number is the octal number of words allocated for variables since the beginning of the program (or routine). In the body of the program (or routine), the number is the total number of words of code and data allocated thus far in the compilation unit. Data space is not added into this total until the end of the corresponding procedure (or end of the main block in the case of globals). In any case, the number represents the relative offset from the start of the compilation unit.

Lines containing errors are followed by error lines. Lines 14, 17, 23, and 25 in the previous example, contained errors. For a detailed explanation of error lines refer to the section Errors in this Chapter.

The last three lines after the source give the number of errors detected, the numbers of the lines where the first and last errors were found, the number of source lines read, and the number of words of code generated. When the first error is encountered, code generation stops, resulting in faster compilation of the rest of the program.

At the end of the listing, the numbers of the errors encountered during compilation are listed with descriptions of the errors they represent.

The compiler options STATS and TABLES can be used to send additional information to the list file (refer to Appendix D).

Loading A Program

As described above, the monitor PASCL schedules the assembler to assemble the code generated by the compiler. The assembler produces a relocatable code file which must be loaded before it can be executed. The loader links together the various compilation units, non-Pascal routines, and library routines, and resolves all external references to produce an absolute load module, which can then be executed with the RUN command.

This section describes the loader commands necessary to load Pascal/1000 programs of various configurations. For a full description of the loader and its operation, refer to the RTE-IVB Terminal User's Reference Manual.

If the loader is run with the command

```
:RU,LOADR
```

it expects the commands to be entered interactively. It will prompt for the commands, one line at a time, with

```
/LOADR:
```

The loader can also be run with the command

```
:RU,LOADR,<command file namr>
```

in which case it expects the commands to be in the file named by <command file namr>.

If the Pascal run-time library ($PLIB) has not been generated into your system, the library (LI) command or the search (SE) command must be used when loading a Pascal program. In the examples below, the LI commands are unnecessary if the library is generated into your system.

Example 1: Load a Pascal/1000 program consisting only of a main program unit named %MAIN. There are no subprogram units and no non-Pascal routines.

```
LI,$PLIB
RE,%MAIN
EN
```

Example 2: Same as Example: 1, except that subprogram units %SM1 and %SM2 are to be combined with the main program unit.

```
LI,$PLIB
RE,%MAIN
RE,%SM1
RE,%SM2
EN
```

Example 3: Same as Example: 2, except that external references in %MAIN, %SM1, and %SM2 are satisfied in non-Pascal routines %ALPHA and %BETA.

```
LI,$PLIB
RE,%MAIN
RE,%SM1
RE,%SM2
RE,%ALPHA
RE,%BETA
EN
```

Example 4: Same as Example: 3, except that routines %ALPHA and %BETA are kept in a library named $NPRTN.

```
LI,$PLIB
RE,%MAIN
RE,%SM1
RE,%SM2
SEA,$NPRTN
EN
```

Example 5: Load a segmented Pascal/1000 program consisting of a main area and three segment overlays. The main area contains the main program unit %MAIN and subprogram unit %SUB01. Segment overlay 1 contains segment unit %SEG1 and subprogram unit %SUB11. Segment overlay 2 contains segment unit %SEG2 and subprogram units %SUB21 and %SUB22. Segment overlay 3 contains segment unit %SEG3. Refer to Figure 3-1 for a diagram of this program's structure.

```
LI,$PLIB
** Load main program unit
RE,%MAIN
RE,%SUB01
** Load segment unit 1
RE,%SEG1
RE,%SUB11
** Load segment unit 2
RE,%SEG2
RE,%SUB21
RE,%SUB22
** Load segment unit 3
RE,%SEG3
EN
```

It is often desirable to have two versions of a progrm loaded at the same time. If the program is segmented, however, the loader will create segments that have their first two characters replaced by '..', making them inaccessible from the program. To correct for this problem, the file %..GER can be relocated in the main. When a call to @SGLD is made to overlay the segment ABCDE, for example, the segment ..CDE will be brought in.

Running A Program

When a program has been successfully relocated by the RTE loader, it is in executable form. To determine the program run string, refer to the program heading in the source code file. It contains the program name and parameter list. The parameter list gives the Pascal file names which will be associated with FMGR file names or LU's in the run string. To run the program, enter the FMGR command RU followed by the program name, then the FMGR namrs of the files or LU's which are to correspond to the files in the parameter list.

Example:

If the program heading is
```
PROGRAM test (input, output);
```
then the run command can be
```
RU,TEST,1,1
```
in which case input is read from and output written to LU 1, or
```
RU,TEST,INFIL,OUTFIL
```
in which case input is read from the file INFIL and output written to the file OUTFIL.

Errors

Errors may be flagged during several stages of program development. Errors resulting from illegal Pascal grammar and logic are compile-time errors and run-time errors, respectively. Other errors, such as FMGR errors and RTE loader errors, are not covered in this manual.

Compile-Time Errors

A compile-time error is one detected by the compiler. When the Pascal compiler detects an invalid construct in the source code, it prints an error line following the line containing the error.

An error line contains, in this order:

- The NUMBER of the line on which the previous error occurred (in the case of the first error, this is zero).

  Note: To locate every error in the program, start at the last one (its line number is printed at the end of the listing) and work towards the beginning of the program.

- Four ASTERISKS (****) which identify a compile-time error line.

- A CARET (^) pointing to each invalid construct in the line above.

- A LIST OF ERROR NUMBERS following each caret. The numbers correspond to the errors associated with the construct above the caret.

The error numbers encountered during compilation are listed, along with descriptions of the errors they represent, at the end of the listing (refer to the section The Listing in this Chapter for an example). Appendix C contains a complete listing of Pascal/1000 compile-time errors. It can also be obtained by listing the file "PERRS.

Run-Time Errors

Run-time errors are those which occur during program execution. The Pascal/1000 run-time library detects five types of run-time errors:

Program errors
EMA errors
Input/output errors
FMP errors
Segmentation errors

Pascal/1000 run-time errors are listed at LU 1. Each consists of three asterisks (***) followed by a brief error description. (Refer to Appendix B for a list of the possible Pascal/1000 run-time errors.)

Pascal/1000 run-time errors cause the program to abort. If this is undesirable, you may substitute your own run-time error-handling routines for those provided (see Appendix B).

Debugging Tools

Four tools available for debugging Pascal programs are:

Range-checking
Procedure and function tracing
Mixed listing
System debugger

Range Checking

When the RANGE compiler option is ON (the default), array subscripts, subrange assignments, and the subranges of routine parameters are checked at run time. The run-time error,

*** PASCAL ERROR: VALUE OUT OF RANGE IN LINE XXXX

is issued when an array subscript is outside its specified bounds, when a variable is assigned a value outside its specified range of values, or when an actual parameter value is outside the specified subrange of its corresponding formal value parameter. Without range-checking, an error such as an out-of-bounds array subscript will go unflagged, and resulting run-time errors or erroneous solutions may not reflect the error.

Procedure and Function Tracing

In debugging a program it may be useful to know which routines are called and in what order they are called. The compiler option TRACE and one of the Pascal libraries, %TRACA, %TRACB, or %TRACC can be used to provide routine entry and exit information when the program is executed. If $TRACE <lu>$ appears in the source code, the Pascal/1000 compiler will generate the code necessary to trace routines until it encounters $TRACE 0$ or the end of the program. Trace information will be printed at <lu> (no trace is printed if <lu> is 0, the default).

Special library routines are used to generate the trace information. These routines are part of the Pascal libraries. The command sequence to the RTE loader must include a command to search one of the Pascal trace libraries before the command to search the Pascal run-time library. If the command to search a trace library is absent or follows the command to search $PLIB, no trace information will be generated, even if $TRACE <lu>$ (where <lu> is not zero) is in the program.

Note: Tracing can be turned on and off as needed. Only routines compiled with a non-zero TRACE option will appear in the TRACE output.

Example: This program is used in the following examples to illustrate the trace information generated by the different libraries.

```
PROGRAM sample_trace;
$TRACE 4$

TYPE
   pos_int = 0..32767;

VAR
   value : pos_int;

PROCEDURE one;
   BEGIN
   END;

PROCEDURE two(value : pos_int);
   PROCEDURE inside_two(value : pos_int);
      BEGIN
         one;
         value := -value;  {causes value to go out of range}
      END;
   BEGIN
      inside_two(value);
   END;

BEGIN
   value := 23;
   one;
   two(value);
END.
```

Trace Library A

When trace library %TRACA is used, entry and exit information for each routine is generated. The following loader command sequence is used to load program sample_trace with trace library A.

```
LI,%TRACA
LI,$PLIB
RE,%SAMPL
EN
```

The trace information sent to LU 4 when this program executes consists of:

```
>     1 Enter: SAMPL
>     2 Enter: ONE
>     2  Exit: ONE
>     2 Enter: TWO
>     3 Enter: INSIDE_TWO
>     4 Enter: ONE
>     4  Exit: ONE
>     3  Exit: SAMPL
```

The number at the left of each line of the trace is the length of the dynamic call chain at that point; that is, the depth into which a routine is being entered, or from which a routine is being exited. As illustrated by the abbreviation of the program name, 'sample_trace,' to 'SAMPL', only the first five characters of the program name appear when it is listed in the trace. Up to 50 characters of a procedure or function name can appear. Identifiers are printed in capital letters in the trace regardless of how they appear in the source.

Trace Libraries B and C

In the following descriptions, n=20 for trace library %TRACB, and n=100 for trace library %TRACC.

Under the heading:

```
** TRACE tree **
```

is a traceback of the first n of the currently-active routines. The traceback is only output when the program terminates with a Pascal run-time error or by executing the halt procedure in the program. When the number of routines in the traceback exceeds n, the message:

```
trace overflow...
```

is listed after the first n routines in the traceback. If this happens with trace library B, the program can be reloaded and run with trace library C. If the program terminates normally, the traceback will contain only the name of the main program.

After the traceback, under the heading

```
** TRACE calls **
```

is a list of the last n-1 routine entries and exits prior to program termination.

The following loader command sequence will load program sample_trace with trace library B:

```
LI,%TRACB
LI,$PLIB
RE,%SAMPL
EN
```

The trace information sent to LU 4 when sample_trace executes consists of:

```
** TRACE tree **
     SAMPL      1
     TWO        2
     INSIDE     3

** TRACE calls **
--->>>SAMPL     1
--->>>ONE       2
<<<---ONE       2
--->>>TWO       2
--->>>INSIDE    3
--->>>ONE       4
<<<---ONE       4
```

An arrow pointing to the right indicates routine entry. An arrow pointing to the left indicates routine exit. The first five characters of the program name and the first six characters of the routine names are listed. The program and routine names are output in capital letters regardless of how they appear in the source. The number on the right is the length of the dynamic call chain as described above for trace library A. If the program terminates with a MP (memory protect) or DM (dynamic mapping) instruction, or any error that causes an immediate abort by the operating system, no trace information will be listed when trace libraries B and C are used.

Mixed Listing

The LIST_CODE compiler option is used to get mixed listings of a Pascal/1000 program or just part of a program. If LIST_CODE is ON (default is OFF), the assembly code generated by the compiler is sent to the list file. LIST_CODE may be turned ON and OFF around Pascal constructs so that only the assembly code generated for those constructs is listed, or it may be left ON throughout the program so that the assembly code for the whole program is listed. The source line is always listed before the assembly code that is generated for it.

Example: The program list_digits is followed by the mixed listing created during compilation.

```
PROGRAM list_digits (OUTPUT);

PROCEDURE write_digits;

VAR
   i: 0..9;

BEGIN {write_digits}

   {turn LIST_CODE on around FOR loop}

   $LIST_CODE ON$
   FOR i := 0 TO 9 DO BEGIN
      write(i:2);
   END;
   $LIST_CODE OFF$
END;  {write_digits}

BEGIN {list_digits}
   write_digits;
   writeln;
END.  {list_digits}
```

```
Pascal/1000                                  Thu Feb 28, 1980   2:53 pm
Ver. 1/2015                                  &SAMPL::HP         Page 1

    1 00000 PROGRAM list_digits (OUTPUT);
    2 00306
    3 00306 PROCEDURE write_digits;
    4 00000
    5 00000 VAR
    6 00000    i: 0..9;
    7 00001
    8 00001 BEGIN {write_digits}
    9 00003
   10 00003    {turn LIST_CODE on around FOR loop}
   11 00003
   12 00003    $LIST_CODE ON$
   13 00003    FOR i := 0 TO 9 DO BEGIN
                   00003.8    LDA =D-10
                   00004      STA .10+1
                   00005      CLA
                   00006      STA .10+0
   14 00007       write(i:2);
                   00007.18   JSB @WRI
                   00010      DEF *+4
                   00011      DEF @1+0
                   00012      DEF .10+0
                   00013      DEF .5+0
   15 00014    END;
                   00014      ISZ .10+1
                   00015      JMP *+2
                   00016      JMP .19
                   00017      LDA .10+0
                   00020      INA
                   00021      JSB @CKI
                   00022      DEF .5+1
                   00023      DEC 15
                   00024      STA .10+0
                   00025      JMP .18
   16 00026    $LIST_CODE OFF$
   17 00026 END;  {write_digits}
   18 00054
   19 00054 BEGIN {list_digits}
   20 00074   write_digits;
   21 00076   writeln;
   22 00101 END.  {list_digits}


            0 Errors detected.
           22 Source lines read.
          285 Words of program generated.
```

Interactive Debugging

The RTE interactive debugger DBUGR can be used to check programs for logical errors during execution. Using DBUGR, the user may examine and modify memory and registers, set breakpoints, and trace instruction execution. A complete mixed listing (including symbol table information) and a load map are recommended for reference when using DBUGR. Symbol table information will be sent to the list file if compiler option TABLES is ON (default is OFF). Refer to the RTE-IVB Terminal User's Reference Manual for DBUGR instructions.

Interfacing Pascal With Non-Pascal Routines

Pascal programs may call routines written in FORTRAN or assembly language and vice-versa. The .ENTR calling sequence is used unless the $DIRECT$ option is specified. The use of .ENTR is transparent to the programmer in Pascal and FORTRAN because those compilers generate the necessary code needed for using .ENTR. An Assembly language routine must refer to .ENTR or perform the equivalent operation itself. When passing parameters between routines of different languages, be aware that the parameter values may be represented differently in different languages.

Calling Non-Pascal Routines From Pascal Routines

A Pascal program, subprogram, or segment which calls a non-Pascal routine must declare it to be external before calling it. When the RTE loader is run, the non-Pascal routine must be relocated with the main area or segment overlay which uses it. If the non-Pascal routine is in a library, this can be done by searching that library.

Calling Pascal Routines From Non-Pascal Routines

A non-Pascal routine that has been combined with a Pascal/1000 main program unit can access any level-1 routine in the main area. If the program is segmented, a non-Pascal routine in the main area can access any level-1 routine in a segment overlay that has been loaded into memory. A non-Pascal routine in a segment overlay that is already loaded into memory can access any level-1 routine in the main area or in the segment overlay. If a Pascal routine is called, the .ENTR calling sequence must be used, unless the $DIRECT$ compiler option was used for the Pascal routine.

A non-Pascal routine can access global data only through parameters passed to the routine.

Calling Pascal Routines From Non-Pascal Programs

Non-Pascal programs can call Pascal routines as long as:

1. The Pascal routine to be called is declared in a subprogram or segment unit, not in a main program unit. This subprogram or segment unit must be relocated with the non-Pascal program at load time.

2. In each subprogram or segment unit the compiler option:

   ```
   $HEAP 0, RECURSIVE OFF$
   ```

   must be included to suppress any use of the Pascal heap and stack.

3. The Pascal routine can only use its local objects (including routine parameters). No global objects or program parameters in the subprogram or segment unit can be used by the Pascal routine.

Pascal And FORTRAN

The programmer should be aware of the following differences between Pascal and FORTRAN when interfacing between them.

Booleans:
In Pascal, the value of true is 1 and the value of false is 0. In FORTRAN, the value of true is any negative value and the value of false is any nonnegative value.

Arrays:
In Pascal, there is no limit to the number of array dimensions allowed. Arrays are stored in row-major order. In FORTRAN, the number of array dimensions is limited, and arrays are stored in column-major order.

Files:
A file used by both Pascal and FORTRAN routines must be declared in Pascal. If a file is opened in a Pascal routine (or main), both FORTRAN and Pascal routines can write to it or close it. If a file is opened in a FORTRAN routine (or main), it can only be written to or closed from FORTRAN. For more information, refer to the section FMP vs. Pascal/1000 I/O in Chapter 8.

Pascal Segments and Subprograms:
If the main program is a FORTRAN program, the Pascal segments and subprograms cannot have any global variables (this includes file parameters in their headings). Global types and constants (except structured and string constants) may be declared in the Pascal compilation units.

COMMON:

Pascal cannot directly use variables stored in FORTRAN COMMON. A Pascal routine within a subprogram, however, can access common variables in a FORTRAN main. This is shown in the following example:

```
FTN4,L
      PROGRAM FTN
      COMMON FIRST,SECOND,THIRD
      REAL FIRST,SECOND,THIRD
      FIRST = 1.0
      SECOND = 2.0
      THIRD = 3.0
      WRITE(1,10) FIRST,SECOND,THIRD
      CALL PAS(FIRST)
      WRITE(1,20) FIRST,SECOND,THIRD
      STOP
   10 FORMAT('STEP 1: System Common initialized to:', 3F10.2)
   20 FORMAT('STEP 2: Changed in Pascal to:         ', 3F10.2)
      END
      END$



  $SUBPROGRAM$
  PROGRAM pasc;

  TYPE
     SYSTEM_COMMON = RECORD
        first,
        second,
        third : REAL;
     END;

  PROCEDURE pas
     (VAR common: SYSTEM_COMMON);

  BEGIN
     WITH common DO BEGIN
        first  := 10.0;
        second := 20.0;
        third  := 30.0;
     END;
  END;
  .
```

EMA parameters:

If a Pascal HEAP 2 (EMA) VAR parameter is passed to a FORTRAN routine, the corresponding FORTRAN parameter must be a call-by-reference EMA parameter. Pascal passes the variable's double-word address. The order of these words must be reversed because FORTRAN expects them in reverse order. The program below demonstrates a method of reversing pointers using a variant record structure.

```
$HEAP 2$

PROGRAM demonstrate_pointer_swapping;

TYPE
   WORD = -32768..32767;
   BUF  = ARRAY [1..100] OF WORD;
   BUF_PTR = ^BUF;

VAR
   pas_ptr : BUF_PTR;

PROCEDURE ftn_routine ( parm : BUF_PTR );
   EXTERNAL;

FUNCTION swap ( EMA_ptr : BUF_PTR ) : BUF_PTR;

   TYPE
      FIELD_TYPE = (pointer, words);
   VAR
      buf_rec : RECORD
         CASE FIELD_TYPE OF
            pointer : ( ptr  : BUF_PTR );
            words   : ( w1,
                        w2,   : WORD   );
         END;
      temp : word;
   BEGIN
      WITH buf_rec DO BEGIN
         ptr  := EMA_ptr
         temp := w1;
         w1   := w2;
         w2   := temp;
         swap := ptr;
       END;
   END; { swap }

BEGIN { main }

   new (pas_ptr);
   ftn_routine ( swap(pas_ptr) );

END.  { main }
```

Number formats:

Although real number format is the same in Pascal and FORTRAN (for instance, 123E45), Pascal uses 'L' with longreal while FORTRAN uses 'D', and Pascal accepts lower case 'e' and 'l' while FORTRAN accepts only capital 'E' and 'D'.

FORTRAN output can contain the following real values, of which two are not valid Pascal input:

FORTRAN output:

0.0	Valid Pascal input
.0	Not valid Pascal input
-.0	Not valid Pascal input

Value parameters:

FORTRAN treats Pascal value parameters as though they were VAR parameters; that is, as though they were passed by reference.

Data Storage:

Data type memory representations may be different between Pascal and FORTRAN. For more details refer to section Data Representation of Chapter 8, and the appropriate FORTRAN reference manual.

Pascal And IMAGE

A Pascal program can use the IMAGE/1000 Data Base Management package. If the program uses the heap/stack area, space must be reserved for IMAGE using the compiler option $IMAGE n$, where n is the number of words to be reserved. The size of n is determined by the complexity of the data base being accessed. In general, n = 2000 will allow most data bases to be accessed. The most that will ever be required is 10,240 words. If the program uses the &HEAP 0$ compiler option and does not use recursion, then it is not necessary to set aside the space explicitly. Refer to the IMAGE/1000 Reference Manual for more information.

The following example illustrates the use of the two IMAGE subroutines dbopn and dbget. They appear in Pascal procedures which check the status of the calls and proceed accordingly.

```
      .
      .
      .
   $IMAGE 5000$

   CONST
      term_lu = 1;
   TYPE
      SINGLE_INTEGER = -32768..32767;
      BASE_TYPE = PACKED ARRAY [1..16] OF CHAR;
      LEVEL_TYPE = PACKED ARRAY [1..6] OF CHAR;
      STATUS_TYPE = ARRAY [1..10] OF SINGLE_INTEGER;
      DS_NAME_TYPE = PACKED ARRAY [1..6] OF CHAR;
      LIST_TYPE = PACKED ARRAY [1..250] OF CHAR;
      BUFFER_TYPE = PACKED ARRAY [1..500] OF CHAR;
      KEY_TYPE = PACKED ARRAY [1..40] OF CHAR;

   {external declaration of IMAGE procedure dbopn}

   PROCEDURE dbopn (VAR ibase: BASE_TYPE;
                    VAR ilevl: LEVEL_TYPE;
                    VAR imode: SINGLE_INTEGER;
                    VAR istat: STATUS_TYPE); EXTERNAL;

   {Pascal procedure which uses dbopn}

   PROCEDURE dbopen (VAR data_base: BASE_TYPE;
                     VAR level: LEVEL_TYPE);
   CONST
      mode = 1;
   VAR
      status: STATUS_TYPE;
   BEGIN {dbopen}
      dbopn(data_base, level, mode, status);
      IF status[1] <> 0 THEN BEGIN
         writeln(term_lu, 'IMAGE ERROR ', status[1], ' ON OPEN');
         halt(1); {to stop program}
      END;
   END; {dbopen}

   {external declaration of IMAGE procedure dbget}

   PROCEDURE dbget (VAR ibase: BASE_TYPE;
                    VAR id: DS_NAME_TYPE;
                    VAR imode: SINGLE_INTEGER;
                    VAR istat: STATUS_TYPE;
                    VAR list: LIST_TYPE;
                    VAR ibuf: BUFFER_TYPE;
                    VAR iarg: KEY_TYPE); EXTERNAL;
```

```
{Pascal procedure which uses dbget}

PROCEDURE master_get (VAR master_ds_name: DS_NAME_TYPE;
                      VAR key: KEY_TYPE;
                      VAR list: LIST_TYPE;
                      VAR buffer: BUFFER_TYPE;
                      VAR found: BOOLEAN);

CONST
   mode = 7; {keyed read}

VAR
   status: STATUS_TYPE;

BEGIN {master_get}
   dbget(cr_base, master_ds_name, mode, status, list, buffer,
         key);
   CASE status [1] OF
        0: found := true;
      107: found := false;
      OTHERWISE
         writeln(term_lu, 'ERROR ', status[1], ' ON DBGET');
         halt(2);
      END;
   END;
END; {master_get}
```

Exec Calls

An executing Pascal/1000 program may request various system services via a call to the EXEC processor. The specific service that is requested is encoded in the calling parameters. The use of EXEC calls provide the following services:

- standard I/O
- disk track management
- program management
- system status return
- class I/O

EXEC calls are described fully in the RTE-IVB Programmer's Reference Manual. This section describes the interface to EXEC calls that is provided by Pascal/1000.

Encoding the Calls

An EXEC call may be coded either as a procedure or as a function. If it is coded as a function, the return value type must be a one-word type to return the value of the A register only, or a two-word type to return the values of both the A and B registers.

The Pascal/1000 compiler does not treat an EXEC call in any special manner. Therefore, it is possible to call EXEC directly if an external declaration has been made with a set of formal parameters.

If the $HEAP 2$ compiler option is used, then the HEAPPARMS option must be OFF for EXEC external declarations with VAR parameters.

Example:

```
PROGRAM example;
CONST
   execll = 11;

TYPE
   INT  = -32768..32767;
   TIME = ARRAY [1..5] OF INT;

VAR
   time_buffer : TIME;

PROCEDURE exec
   (    icode : INT;
    VAR itime : TIME); EXTERNAL;

BEGIN
    :
   exec (execll, time_buffer);
    :
END.
```

It is usually either necessary or desirable to use aliases for each EXEC service used in a program for the following reasons:

- The name EXEC represents an entire class of services. A program using EXEC calls will be more readable if a descriptive Pascal/1000 name is given to each service.

- Each EXEC service requires a different set of parameters. Some services (e.g., EXEC 11) have optional parameters. Since each Pascal/1000 routine must have a specific set of parameters (with respect to order, number, and type) and a specific return type for functions, a separate external declaration, each aliased to EXEC, must be used for each set of parameters.

Example:

```
PROGRAM exec_example;

CONST
   exec7  = 7;
   exec11 = 11;

TYPE
   INT  = -32768..32767;
   TIME = ARRAY [1..5] OF INT;

VAR
   time_buffer  : TIME;
   current_year : INT;

PROCEDURE suspend $ALIAS 'EXEC'$
   (icode : INT);
   EXTERNAL;

PROCEDURE get_time $ALIAS 'EXEC'$
   (    icode : INT;
    VAR itime : TIME);
   EXTERNAL;

PROCEDURE get_time_and_year $ALIAS 'EXEC'$
   (    icode : INT;
    VAR itime : TIME;
    VAR iyear : INT);
   EXTERNAL;

BEGIN
     :
     :
   get_time (exec11, time_buffer);
     :
     :
   get_time_and_year (exec11, time_buffer, current_year);
     :
     :
   suspend (exec7);
     :
     :
END.
```

No-Abort Bit and Error Return

Normally, if an EXEC call is successful, control will return to the first instruction after the call, as with any other Pascal/1000 procedure or function call. If an error is encountered during the call, the program will abort.

If this abort is not desired, then bit 15 (the no-abort bit) of the first parameter (service request code) of the EXEC call can be set to 1. When the no-abort bit is set for an EXEC call encoded as a procedure call and an error is encountered during the call, control will return to the first instruction after the call. This is known as the "error return". Otherwise, if the call succeeded, control will return to the second instruction after the call. This is known as the "normal return".

In order for error returns and normal returns to be meaningful in a Pascal/1000 program, the first statement after a no-abort EXEC procedure call must be one instruction that is one word in length. There are two statements in this category:

- GOTO statement
- Procedure statement of a parameterless procedure that was declared with the $DIRECT$ compiler option.

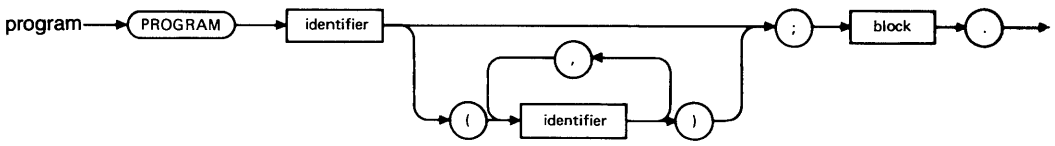
The no-abort bit must not be set for EXEC calls encoded as function calls, since the concept of error returns is meaningless for functions.

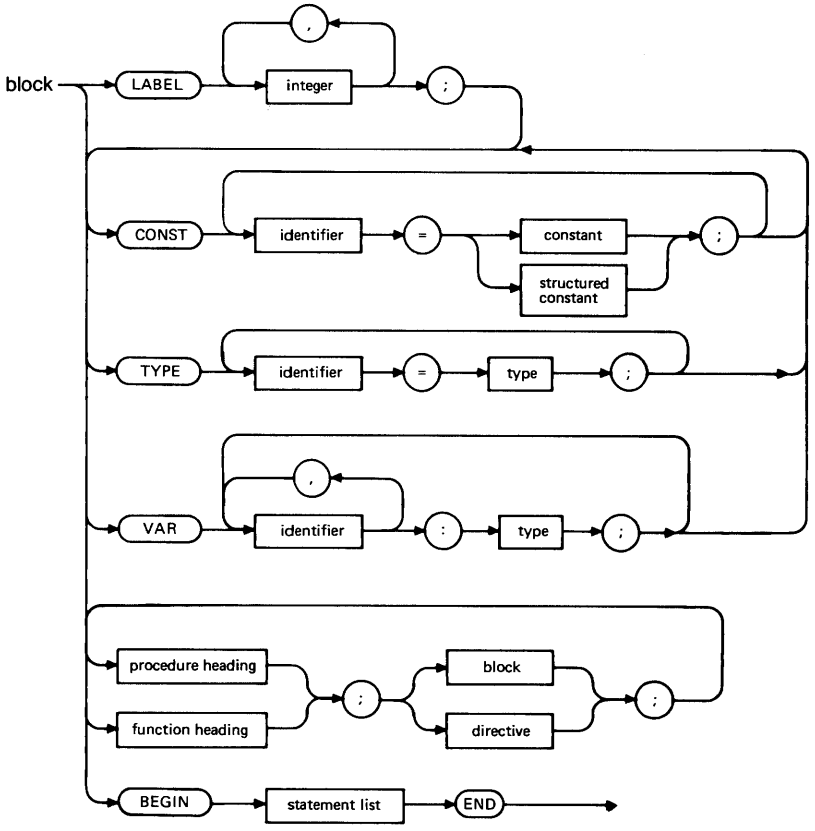
Example:

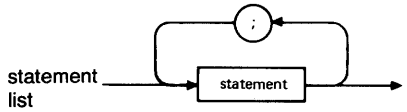
```
PROGRAM exec_program(OUTPUT);

CONST
   no_abort_bit = -32768;  {octal 100000}
   exec7        = 7 + no_abort_bit;

TYPE
   INT = -32768..32767;
   ASCII_WORD = PACKED ARRAY [1..2] OF CHAR;

PROCEDURE abreg(VAR a_reg, b_reg : ASCII_WORD);
   EXTERNAL;


PROCEDURE exec_error;
   $RECURSIVE OFF, DIRECT$
   VAR
      a_reg, b_reg : ASCII_WORD;
   BEGIN
      abreg (a_reg, b_reg);
      writeln ('**ERROR IN EXEC CALL - ERROR CODE IS:',a_reg,b_reg);
      halt(1);
   END;
   $RECURSIVE ON$

PROCEDURE suspend
   (icode : INT);
   EXTERNAL;

BEGIN
    :
    :
   suspend (exec7);
   exec_error;
    :
    :
END.
```

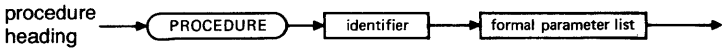
Appendix A
Syntax Diagrams

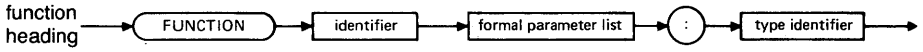
Syntax Diagrams

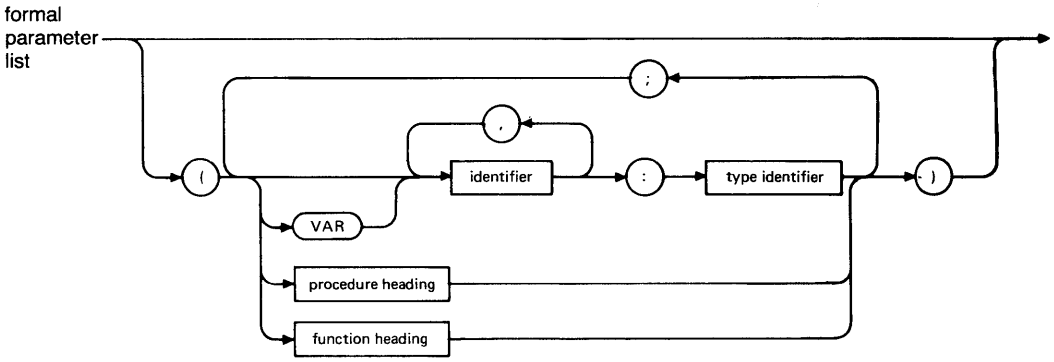


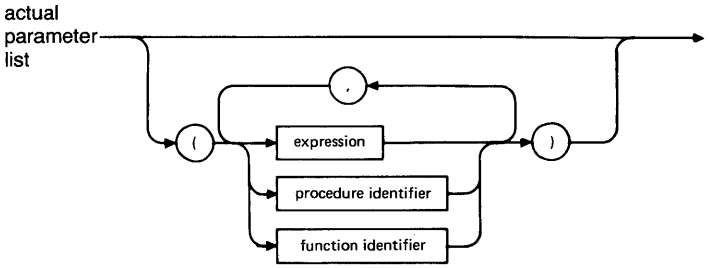


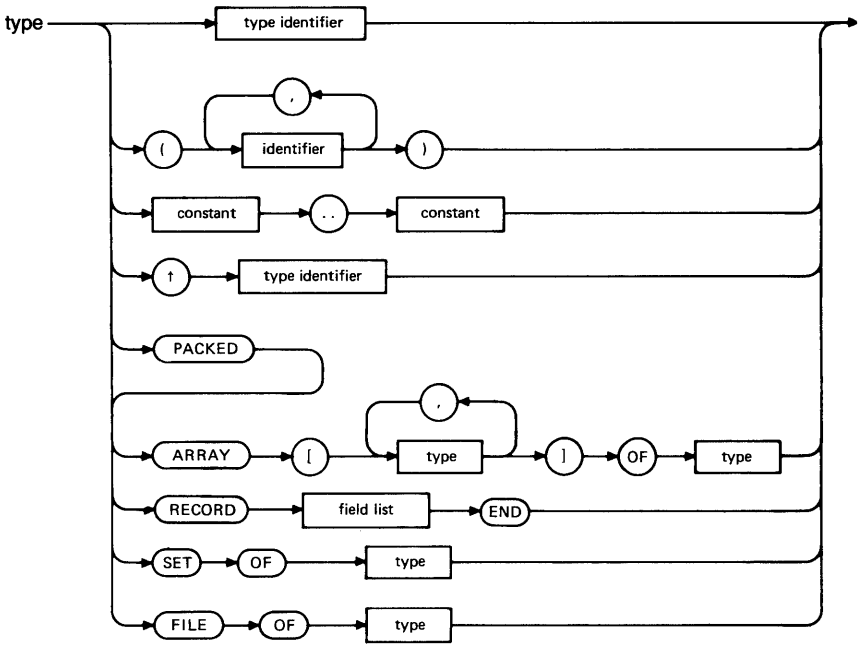


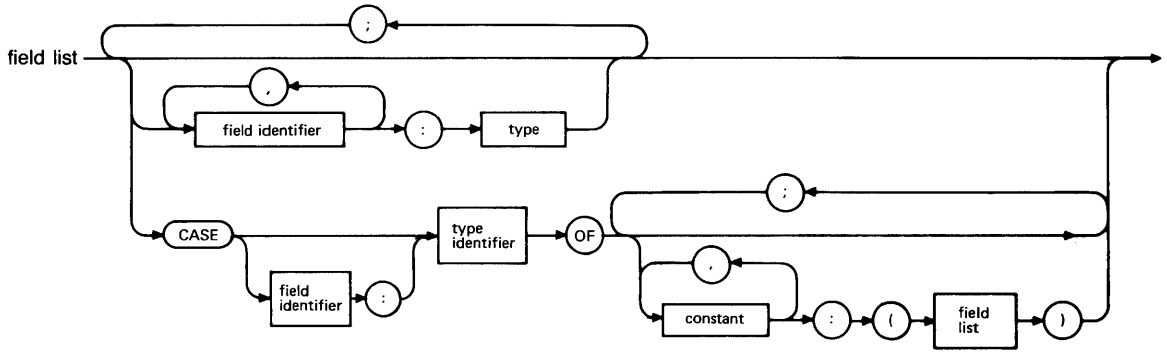

procedure heading
PROCEDURE
identifier
formal parameter list

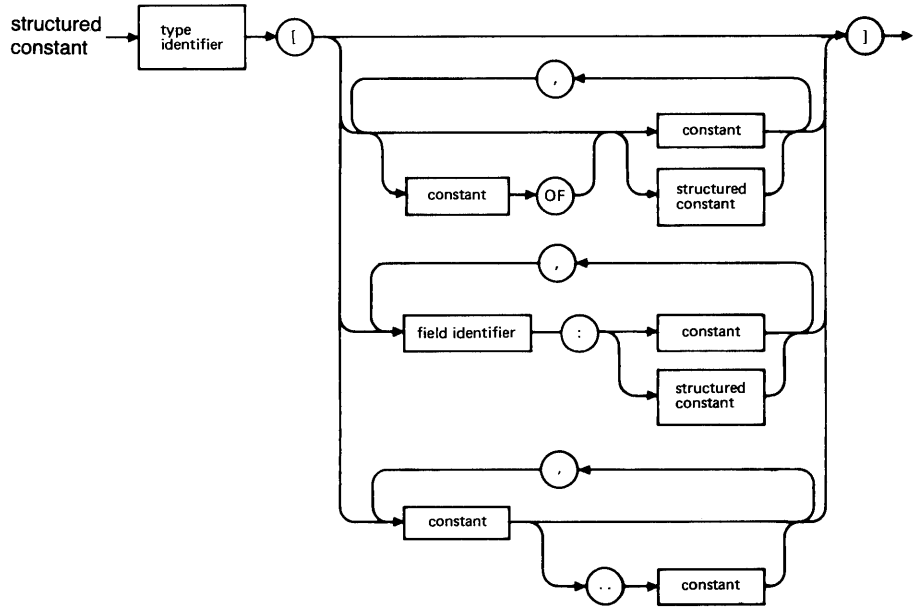

function heading
FUNCTION
identifier
formal parameter list
:
type identifier

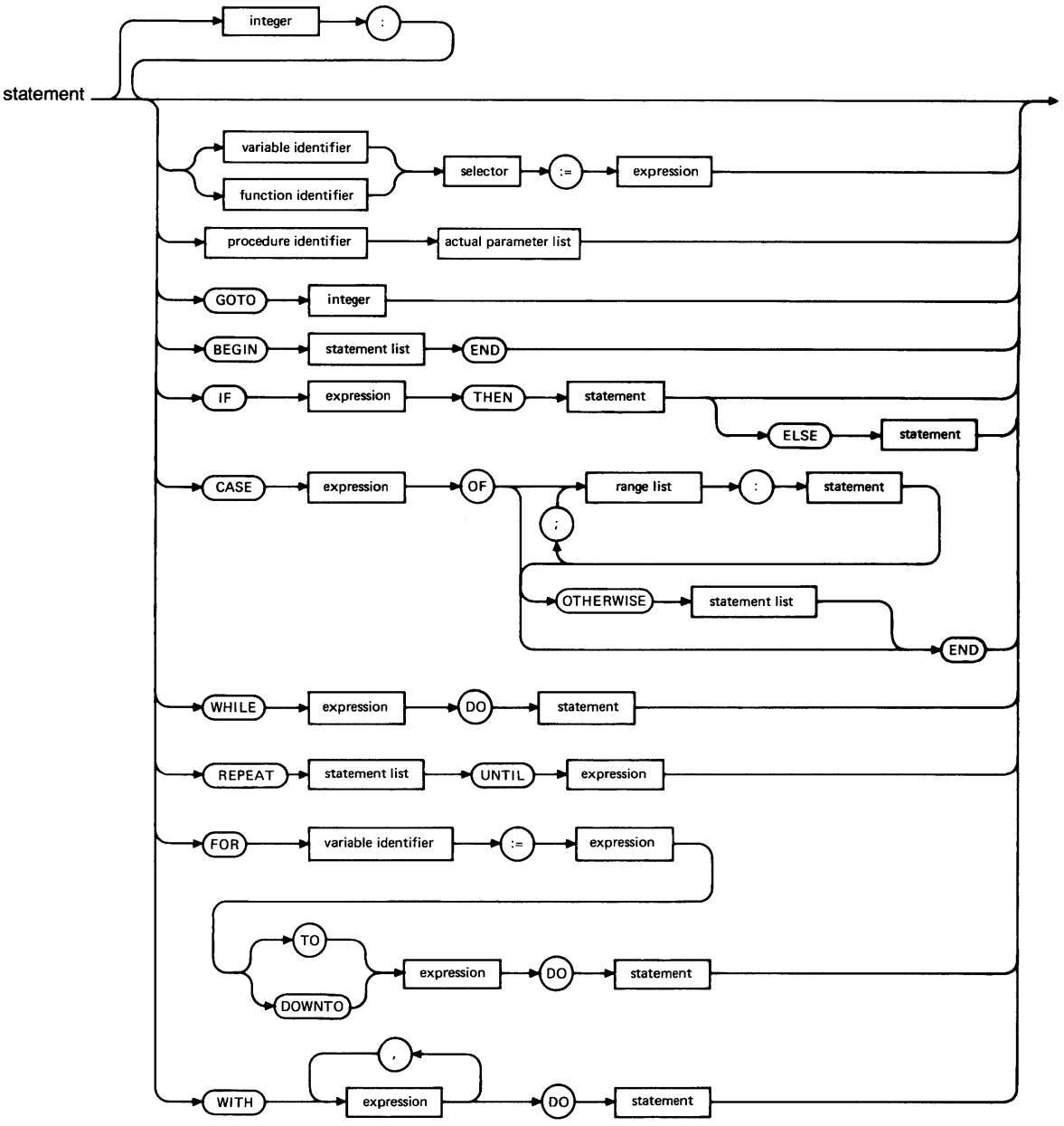

formal parameter list
;
,
(
identifier
:
type identifier
)
VAR
procedure heading
function heading

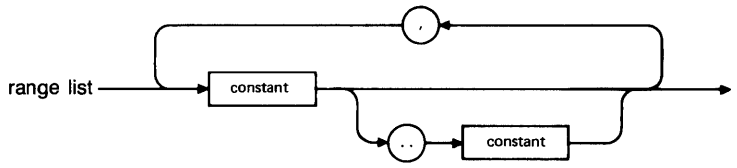

actual parameter list
,
(
expression
)
procedure identifier
function identifier

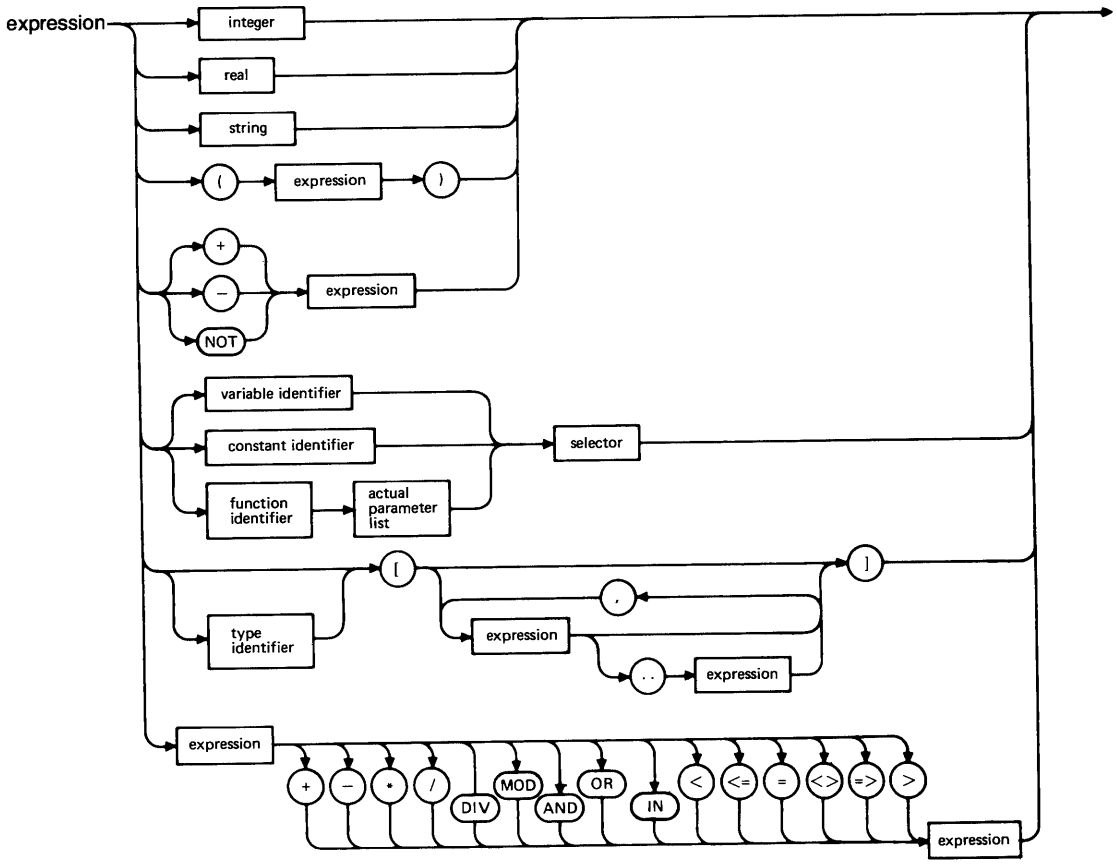

type
type identifier
,
(
identifier
)
constant
..
constant
↑
type identifier
PACKED
ARRAY
[
,
type
]
OF
type
RECORD
field list
END
SET
OF
type
FILE
OF
type

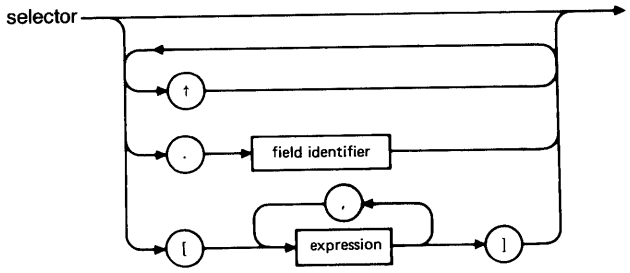

field list
;
,
field identifier
:
type
CASE
field identifier
:
type identifier
OF
;
,
constant
:
(
field list
)

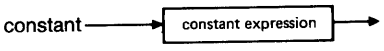

structured constant
type identifier
[
]
,
constant
constant
OF
structured constant
,
field identifier
:
constant
structured constant
,
constant
..
constant

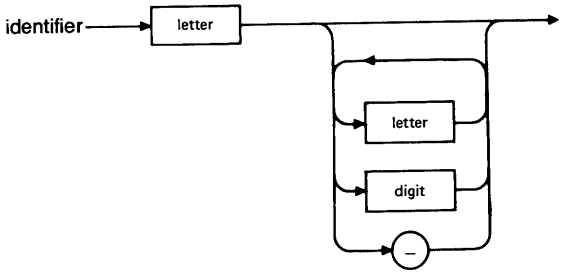

statement
integer
:
variable identifier
function identifier
selector
:=
expression
procedure identifier
actual parameter list
GOTO
integer
BEGIN
statement list
END
IF
expression
THEN
statement
ELSE
statement
CASE
expression
OF
range list
:
statement
;
OTHERWISE
statement list
END
WHILE
expression
DO
statement
REPEAT
statement list
UNTIL
expression
FOR
variable identifier
:=
expression
TO
DOWNTO
expression
DO
statement
WITH
,
expression
DO
statement

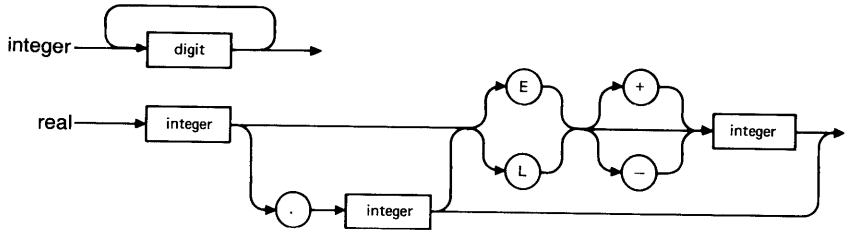

range list
,
constant
..
constant

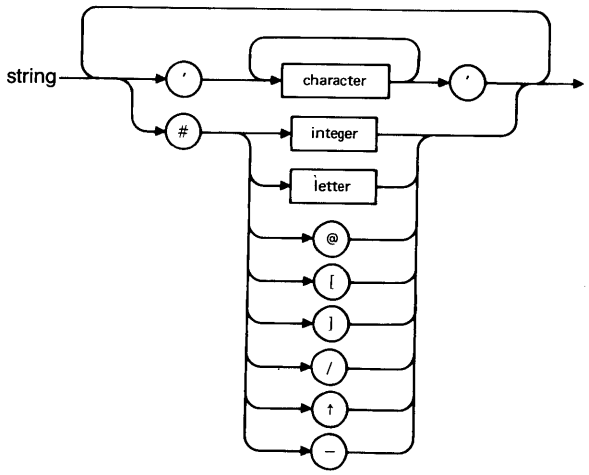

expression
integer
real
string
(
expression
)
+
–
NOT
expression
variable identifier
constant identifier
function identifier
actual parameter list
selector
[
type identifier
expression
,
..
expression
]
expression
+
–
*
/
DIV
MOD
AND
OR
IN
<
<=
=
<>
=>
>
expression

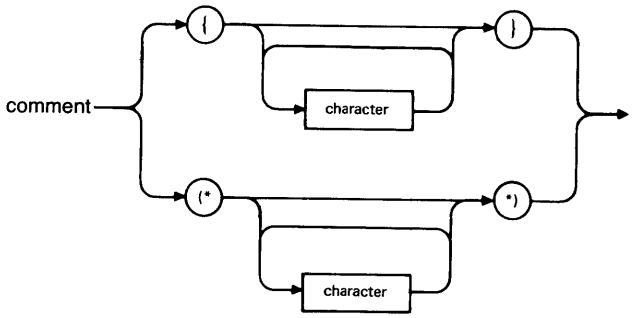

selector
↑
.
field identifier
[
,
expression
]

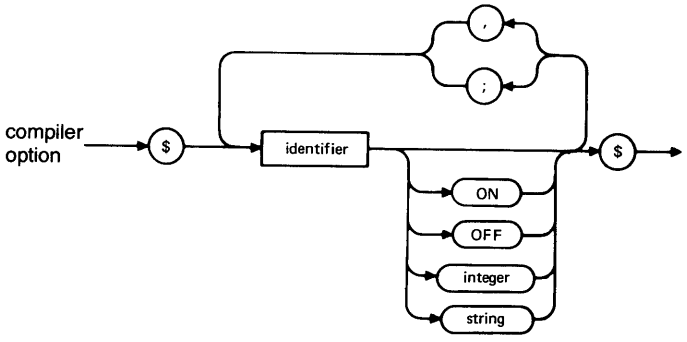

constant
constant expression


identifier
letter
letter
digit
_


integer
digit
real
integer
E
L
+
–
integer
.
integer


string
'
character
'
#
integer
letter
@
[
]
/
↑
_


comment
{
}
character
(*
*)
character


compiler
option
$
identifier
,
;
$
ON
OFF
integer
string

Appendix B
Run-Time Errors and Warnings

There are five types of errors which can be detected during the execution of a Pascal/1000 program. These are Program errors, I/O errors, FMP errors, EMA errors, and Segmentation errors.

Program Errors

These errors occur when the program detects a value that it cannot process or a problem is detected in the management of the heap/stack area. The error message is of the form:

```
*** PASCAL ERROR: "ERROR MESSAGE"
```

Number	Error Message
1	UNDEFINED CASE IN LINE xxxx
2	HEAP/STACK COLLISION IN LINE xxxx
3	NIL POINTER DEREFERENCED IN LINE xxxx
4	VALUE OUT OF RANGE IN LINE xxxx
91	DISPOSE CALLED WITH A NIL PTR IN LINE xxxx
92	DISPOSED OF AN INVALID VARIANT IN LINE xxxx
93	RELEASE CALLED WITH A NIL PTR IN LINE xxxx
94	DISPOSE CALLED WITH A BAD PTR IN LINE xxxx
95	RELEASE CALLED WITH A BAD PTR IN LINE xxxx
99	INSUFFICIENT IMAGE SPACE

Undefined CASE

The line number is that of the line of the case statement on which the final END appears. A case selector had a value which did not correspond with any case label, and no OTHERWISE clause was specified.

Heap/Stack Collision

The line number (if present) is that of the line on which a call to NEW was made that exhausted the heap area. If no line number appears, a call to a recursive routine has exhausted the stack area.

Nil Pointer Dereferenced

An attempt was made to use a dynamic variable that did not exist (i.e. the pointer to it was nil).

Value Out of Range

A value to be assigned, used as a pointer or array subscript, or passed as a value parameter, is out of the range of valid values. The line number is that of the end of the assignment statement, the end of the pointer or subscript expression, or the end of the routine to which the argument is being passed.

Dispose Called With a Nil Pointer

Dispose was called with a pointer that did not point to any dynamic variable (i.e. the pointer was nil).

Disposed of Invalid Variant

The alternate form of dispose was called with tag values that specified a record with a different size than that specified when the record was created (with either form of NEW). Pascal/1000 only detects size mismatch; if the tags specify a variant other than that specified when NEW was called, but the size of the variant is the same as that specified by the NEW call, no error will occur.

Release Called With a Nil Pointer

Release was called with a pointer that did not contain the MARKED state of the HEAP. Release should only be called with a pointer that was passed to MARK (and not modified since). If release is called with a pointer created by NEW, no immediate error will occur; however, the results will be unpredicatable.

Dispose Called With a Bad Pointer

Dispose was called with a pointer that did not point into the heap area (i.e. the pointer was not initialized with NEW or has been corrupted).

Release Called With a Bad Pointer

Release was called with a pointer that did not point into the heap area (i.e. the pointer was not initialized with MARK or has been corrupted).

Insufficient Image Space

After the program was loaded into a partition, there was not enough free space remaining between the end of the program and the end of the partition to meet the IMAGE specification. The program can be made to run in a larger partition with the SZ command, or the amount of IMAGE area can be reduced.

I/O Errors and Warnings

A I/O error occurs when an attempt is made to incorrectly access a Pascal/1000 logical file, or a physical file is not compatible with a Pascal/1000 logical file. The error message is of the form:

```
*** PASCAL I/O ERROR ON FILE xxxxx
    "Error Message"
```

Where "xxxxx" is the name of the Pascal/1000 logical file.

Number	Error Message
1	UNEXPECTED EOF
2	FILE MUST BE TEXT
3	FILE MUST BE DIRECT
4	BAD RECORD LENGTH
5	MUST RESET OR OPEN FILE
6	MUST REWRITE OR OPEN FILE
7	DIRECT ACCESS READ ERROR
8	SEQUENTIAL ACCESS READ ERROR
9	INVALID INTEGER READ
10	LINE READ WAS TOO LONG
11	INVALID REAL NUMBER READ
12	FILE IS NOT CCTL
13	NO SCRATCH FILE AVAILABLE
14	MINUS FIELD WIDTH NOT ALLOWED
15	FILE CANNOT BE TYPE 1 OR 2
16	FILE MUST BE TYPE 1 OR 2
17	CANNOT OPEN LU 0 FOR READ ONLY
18	MISSING FILE NAMR

The warning message is of the form:

```
*** PASCAL I/O WARNING ON FILE xxxxx
    "Error Message"
```

Where "xxxxx" is the name of the Pascal/1000 logical file.

Number	Error Message
1	OUTPUT LINE MOVED TO NEXT LINE
2	OUTPUT LINE SPLIT

FMP Errors

A FMP error occurs when an attempt is made to incorrectly access a physical file. See the appropriate documentation for an explanation of the error codes. The error message has the form:

```
*** FMP ERROR nnnn ON FILE xxxxx
```

Where "nnnn" is the FMP error code and "xxxxx" is the Pascal/1000 logical file name.

EMA Errors

This error occurs when an invalid two-word pointer is dereferenced. It can only occur in programs compiled with the $HEAP 2$ compiler option. The error message has the form:

```
*** PASCAL POINTER ERROR AT xxxxx
```

Where "xxxxx" is the address in the code, expressed in octal, where the error occurred.

Segment Errors

This error indicates that a segment was not found by the segment loader @SGLD. The error message has the form:

```
*** PASCAL SEGMENT xxxxx NOT FOUND
```

Where "xxxxx" is the name of the segment passed to @SGLD.

Error Message Printers

The standard error message printer @PRER in the Pascal library prints the long error messages described above. This routine, however, is quite large since it contains the text of all the messages. If a smaller error routine is desired, the file %PRERS can be relocated, in the LOADR, before the Pascal library is searched. This short error routine is about one third the size of the standard error message printer; it only prints the error number for PROGRAM and I/O errors, rather than the long descriptive message.

Catching Errors

Normally, the occurrence of a run-time error causes the appropriate error message to be issued and the program to be terminated. However, a programmer can choose to have a program catch run-time errors itself.

Each of the six types of run-time errors or warnings normally causes an error catching routine, @PREP (from the Pascal library), to be invoked. The version of @PREP in the library simply calls the error printer routine @PRER and then terminates the program. A program can catch and handle run-time errors by providing its own version of @PREP. An appropriate heading for @PREP is of the form:

```
TYPE
   INT        = -32768..32767;
   ERROR_TYPE = (run, ema, io, fmp, seg, warn);
   FILE_NAME  = PACKED ARRAY [1..150] OF CHAR;

PROCEDURE handle_error
   $ALIAS '@PREP'$
   (err_type:   ERROR_TYPE; err_number: INT; err_line: INT;
    err_file:   FILE_NAME;  err_flen: INT);
```

For each ERROR_TYPE the other parameters to @PREP are defined as follows:

```
run:
   err_number: The number of the Pascal program error.
   err_line:   The source line number (or zero if not applicable).
   err_file:   Undefined.
   err_flen:   Undefined.

io:
   err_number: The number of the Pascal I/O error.
   err_line:   Undefined.
   err_file:   The Pascal logical file name.
   err_flen:   The length of the file name.

fmp:
   err_number: The FMP error code.
   err_line:   Undefined.
   err_file:   The Pascal logical file name.
   err_flen:   The length of the file name.

ema:
   err_number: Undefined.
   err_line:   The address of the pointer dereference operation.
   err_file:   Undefined.
   err_flen:   Undefined.

seg:
   err_number: Undefined.
   err_line:   Undefined.
   err_file:   The segment name.
   err_flen:   The length of the segment name (always 5).

warn:
   err_number: The number of the Pascal I/O error.
   err_line:   Undefined.
   err_file:   The Pascal logical file name.
   err_flen:   The length of the file name.
```

Having processed the error, the user-supplied @PREP can cause the default error printer to be invoked (which will write a message describing the error to LU 1) by calling the procedure @PRER with the same parameters (in the same order) that they were passed to @PREP. Note that @PRER does not terminate the program; this must be done by the user-supplied @PREP routine if that is the desired action. Every attempt is made to ensure that if the user @PREP exits, that the program will continue "normally", essentially ignoring the error. However, it should be clear that some errors cannot be safely "ignored" and the results of doing so are unpredictable.

NOTE: In some cases (usually when a file has not been opened) the file name may not be correct. When this can be detected the length will be passed as zero. However, it cannot always be detected, so use of file name information is at the users risk.

A possible use of this mechanism would be to place @PRER in a segment so that it would not be present during normal execution. However, replacing @PREP with a version that does a segment load and then calls @PRER will not work, as the external reference to @PRER will cause it to be loaded from the library with @PREP (which is what you were trying to avoid). To make it work, replace @PREP with a version that does the segment load, and then calls a routine (of yours) in the segment, which in turn calls @PRER. Obviously the parameters must be passed to your routine so that it can pass them to @PRER.

NOTE: The user-supplied version of @PREP as well as any routines which are given access to any of the parameters of @PREP must be compiled with the $HEAP 1$ option.

Appendix C
Compile-Time Errors

Pascal/1000 Syntax Errors 1-99

```
  1: error in simple type
  2: identifier expected
  3: 'PROGRAM' expected
  4: ')' expected
  5: ':' expected
  6: illegal symbol
  7: error in parameter list
  8: 'of' expected
  9: '(' expected
 10: error in type
 11: '[' expected
 12: ']' expected
 13: 'end' expected
 14: ';' expected
 15: integer expected
 16: '=' expected
 17: 'begin' expected
 18: error in declaration part
 19: error in field list
 20: ',' expected
 21: '.' expected
 23: string expected
 24: '..' expected
 25: illegal character in this context
 26: ',' or ';' expected
 49: expression must be a constant
 50: error in constant
 51: ':=' expected
 52: 'THEN' expected
 53: 'UNTIL' expected
 54: 'DO' expected
 55: 'TO' or 'DOWNTO' expected
 60: error in expression
 70: external routine must be declared at outermost level
 71: aliased routine must be declared at outermost level
 72: recursive routine may not be direct
 73: actual routine may not have errorexit
 80: negative field width not allowed
```

Pascal/1000 Syntax Errors 100-149

100: duplicate or invalid external name
101: identifier redeclared
102: low bound exceeds high bound
103: identifier is not of appropriate class
104: identifier not declared
105: sign not allowed
106: scope violation
107: incompatible subrange types
108: file not allowed here
109: type must not be real
110: tagfield type must be scalar or subrange
111: incompatible with tagfield type
112: index type must not be real
113: index type must be scalar or subrange
114: base type must not be real
115: base type must be scalar or subrange
116: error in type of standard procedure parameter
117: unsatisfied forward reference
118: undeclared forward procedure or function
119: forward declared; repeated parameter list not allowed
120: function may not return this type
121: file value parameter not allowed
122: forward declared; repeated result type not allowed
123: missing result type in function declaration
124: decimal position for real only
125: error in type of standard function parameter
126: number of parameters does not agree with declaration
127: missing parameter to standard routine
128: result type of parameter function conflicts with declaration
129: type conflict of operands
130: expression is not of set type
131: only tests of equality are allowed
132: strict inclusion not allowed
133: file comparison not allowed
134: illegal type of operand(s)
135: type of operand must be Boolean
136: set element type must be scalar or subrange
137: set element types not compatible
138: type of variable is not array
139: index type is not compatible with declaration
140: type of variable is not record
141: type of variable must be file or pointer
142: illegal parameter substitution
143: illegal type of loop control variable
144: illegal type of expression
145: type conflict
146: assignment of files not allowed
147: label type incompatible with selecting expression
148: subrange bounds must be scalar
149: not assignment compatible

Pascal/1000 Syntax Errors 150-199

150: assignment to standard function is not allowed
151: assignment to formal function is not allowed
152: no such field in this record
153: type error in read
154: actual parameter must be a variable
155: loop control variable must be simple/local variable
156: multidefined case label
157: loop control variable may not be assigned to
158: missing corresponding variant declaration
159: real or string tagfields not allowed
160: previous declaration was not forward
161: again forward declared
162: type error in write
163: missing variant in declaration
164: substitution of standard proc/func not allowed
165: multidefined label
166: multideclared label
167: undeclared label
168: undefined label
169: error in base set
170: value parameter expected
171: actual parameter cannot be component of packed type
172: dynamic variables cannot be/contain a file
173: too many enumerated values
174: file cannot be textfile
175: missing file "input" in program heading
176: missing file "output" in program heading
177: only variables may be assigned to
178: invalid expression
179: function identifier not assignable here
180: type of expression must be Boolean
181: no function result defined in the body of the function
182: program or module cannot be declared forward
183: program or module cannot be declared external
184: warning: division by zero
185: undeclared external file
186: file must be a textfile
187: option conflict
188: option cannot be specified here
189: heap option must be set to use this routine
190: recursive option must be set to do recursion
191: option cannot be respecified
192: option not recognized
193: include level too deep
194: include file cannot be read
195: option has invalid parameter
196: 'ON' or 'OFF' expected

Pascal/1000 Syntax Errors 200-500

200: numeric constant too long
205: real constant exceeds range
206: missing fractional part of real
207: missing scale factor of real or longreal
209: overflow or underflow
210: integer constant exceeds range
215: string constant too long
216: string constant exceeds source line
217: null string is invalid
218: non printing character invalid in string
219: invalid non printing character
220: character constant exceeds range
225: label exceeds range
230: structured type identifier expected
231: too few constants
232: too many constants
233: field(s) not specified
234: field respecified
235: tag not set or set to another variant
236: set type id expected
237: constant of wrong type

250: too many nested scopes of identifiers
251: too many nested blocks of code
252: location counter overflow
253: unexpected end of source file
254: source line too long
255: too many errors on this source line
260: compiler label overflow: break into separate compilations
261: compiler literal pool overflow: expression too complicated
262: insufficient compiler workspace: increase EMA pages in option file

302: index expression out of bounds
303: value to be assigned is out of bounds
304: element expression out of range
305: actual parameter out of bounds
307: expression out of bounds

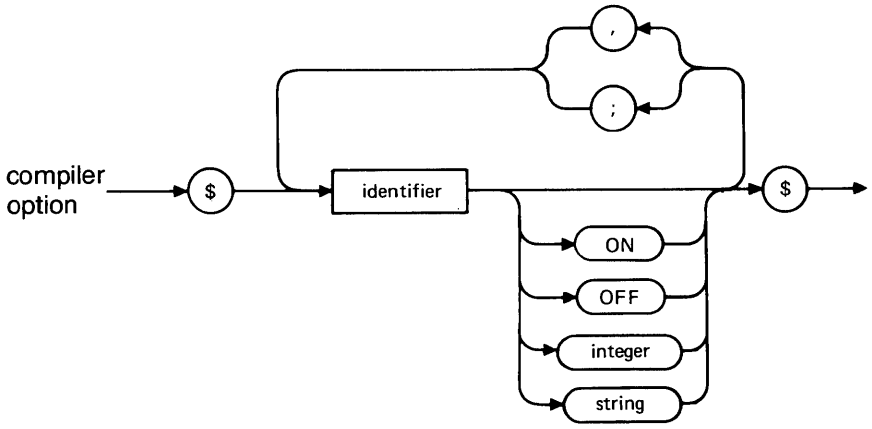
398: implementation restriction
399: "non standard" construct

400 or greater: Compiler error, contact your HP representative.

Appendix D
Compiler Options

Compiler options direct the compiler in processing the source program.

Syntax:



If no option value is specified, options which expect ON or OFF will assume ON. Options which require an integer or string value must have that value specified (with the exception of the PASCAL option). The trailing $ may be omitted, in which case the compiler option is assumed to stop at the end of the line.

ALIAS <string>

The <string> specifies a name to be used to reference the routine from a separately-compiled routine. Only the first five characters will be used if more than five appear. If less than five characters appear, the string will be blank filled on the right. If this option is used it applies only to a single routine, and it must appear following the reserved word PROCEDURE or FUNCTION and before the block or directive. If a routine is declared FORWARD, the ALIAS option must appear at the FORWARD declaration, not at the actual declaration. Only level-1 routines may be aliased. If VISIBLE is OFF (see below) the alias will be used in the generated code, but the routine will not be accessible from separately-compiled routines.

ANSI <ON or OFF>

ON causes a syntax error 399 to be issued for the use of any feature of Pascal/1000 which is not a part of "standard" Pascal (see Appendix E). Default: OFF

ASMB <string>

The <string> specifies the option string to be passed to the assembler. For example, $ASMB 'R,L'$ can be used to get an assembled listing of the emitted code (see KEEPASMB option). If this option is used, it must appear before the program heading. Default: 'R'

AUTOPAGE <ON or OFF>

On specifies that each procedure or function is to be listed on a new page. The page eject is performed after each routine has been compiled, so a nested routine and the body of its enclosing routine are listed on separate pages. A page eject is not performed between the global declarations and the first routine. Default: OFF

BUFFERS <integer>

The <integer> (range 1 to 255) specifies the number of 128 word blocks to make available for the file DCB buffer. This option applies to all succeeding type definitions involving files (its applies at variable declarations for the standard file type TEXT). If the standard files input and/or output appear in the program heading and the BUFFERS option is to apply to them, it must appear before the program heading. Default: 1

CODE <ON or OFF>

ON specifies that assembler source code is to be emitted. This option should be turned OFF with care. When it is OFF the compiler may fail to emit code that is required by the assembler and/or other pieces of code that are emitted while CODE is ON. The compiler turns CODE OFF when the first syntax error is detected (CODE may not be turned back ON by the user after a syntax error). Default: ON

DIRECT

Specifies that a routine will not use the .ENTR calling sequence. The call to .ENTR will not be emitted in the routine code and calls to the routine will not contain an initial DEF to the return address. Arguments are still passed with DEF's (arguments are never passed in registers). If this option is used, it applies only to a single routine, and it must appear after the reserved word PROCEDURE or FUNCTION and before the block or directive. If a routine is declared FORWARD, the DIRECT option must appear at the FORWARD declaration, not at the actual declaration. A DIRECT routine cannot be RECURSIVE, nor can it be passed as a actual parameter (an EXTERNAL routine declared with RECURSIVE ON can be declared DIRECT as RECURSIVE does not apply to EXTERNAL routines). Default: routine is not DIRECT

EMA <string>

The <string> specifies the parameters of the EMA instruction emitted for HEAP 2 programs (see HEAP option). The string is of the form 'emasize,msegsize' (e.g. '65,2') where emasize and msegsize are unsigned integers. If this option is used, it must appear before the program heading. EMA is only effective in the main program unit, it will be ignored in subprogram or segment units.

ERROREXIT

Specifies that an external routine's calling sequence includes an error return (a JSB ERR0 will be emitted following any call). If this option is used, it applies only to a single routine, and it must appear after the reserved word PROCEDURE or FUNCTION and before the directive. Only EXTERNAL routines may have an ERROREXIT. Default: routine does not have ERROREXIT

HEAP <integer>

0 specifies no heap is used.

1 specifies that the heap/stack area resides in the 32K logical address space after the users program. Pointers are one word addresses.

2 specifies that the heap/stack area resides in EMA. Pointers are two-word offsets from the start of EMA. An actual parameter corresponding to a VAR formal parameter is passed as a DEF to a two word address. If the first word of the two word address is negative, the second word contains the 16-bit address of the actual parameter. If the first word is not negative then the two words represent the offset of the actual parameter from the start of EMA.

This option may appear only once in a compilation unit, and if used it must appear before the program heading. Mixing HEAP options among compilation units requires special coordination (see HEAPPARMS). With HEAP 2, objects larger than 1024 words in the heap may not be assigned or passed as value parameters. Default: 1

HEAPPARMS <ON or OFF>

OFF specifies that VAR parameters cannot reside in the heap. This option is used with HEAP 2 programs for efficiency reasons or to pass VAR parameters to a separately compiled routine that was compiled with HEAP set to 0 or 1. If it is known that an actual VAR parameter(s) will never be in the heap, this option is used to permit parameter accessing to be done with a one word address instead of through a two word address (which for non-heap parameters has a negative first word, see HEAP 2). This option may be turned OFF or ON anywhere in the compilation unit and its setting affects the VAR parameters of all routines declared thereafter. In order to take effect for a given VAR parameter, it must be set or reset before the reserved word VAR. HEAPPARMS OFF is ignored in the parameter lists of formal routine parameters. A routine which has any of its VAR parameters declared with HEAPPARMS OFF may not be passed as an actual parameter. HEAPPARMS is ignored in compilation units which are HEAP 1. Default: ON

IDSIZE <integer>

The <integer> (range 1 to 150) specifies the number of significant characters in identifiers. The identifiers may still be up to 150 characters in length, but only the first N characters (where N is the current IDSIZE) will be used as the actual identifier. Default: 150

IMAGE <integer>

The <integer> (range 0 to 32767) reserves the specified number of words from the dynamic memory area for use by the IMAGE subsystem. The area always resides in the 32K logical address space (even in HEAP 2 programs). If this option is used, it must appear before the program heading. A run-time error occurs when the program starts if there is not enough dynamic memory area available to meet the IMAGE specification. IMAGE can be used with HEAP 0. IMAGE is only effective in the main program unit; it will be ignored in subprogram or segment units. Default: 0

INCLUDE <string>

The <string> names a file whose contents are to be included at the current position in the source. Included code may not contain additional INCLUDE options. After including the named file, the compiler continues processing the line on which the INCLUDE option appears. An error will occur if the file is not of type TEXT.

KEEPASMB

Causes the file of generated assembly language code to be kept. If an error occurs in the assembly, the file will be kept whether or not KEEPASMB is specified. If this option is used, it must be specified before the program heading. KEEPASMB turns MIX ON, MIX may be turned OFF after KEEPASMB is specified. If KEEPASMB is specified, the name of the saved assembly language source is displayed after the assembly terminates. The name is the same as that of the relocatable with a (^) replacing the initial (%). If the relocatable name does not start with (%) then the name is prefixed with (^), dropping the last character if the name was already 6 characters long. Default: purge assembler source at end of assembly

LINESIZE <integer>

The <integer> (range 1 to 256) specifies the maximum number of characters in a line that a TEXT file will be able to handle. It applies to all successive type definitions and variable declarations that involve TEXT files. If the standard files input and/or output appear in the program heading and the LINESIZE option is to apply to them, it must appear before the program heading. Default: 128

LIST <ON or OFF>

ON causes the source to be listed. OFF suppresses the listing except for lines that contain errors. While LIST is OFF, AUTOPAGE, PAGE, STATS, and TABLES are effectively OFF. Default: ON

LIST_CODE <ON or OFF>

ON specifies that the program listing is to contain emitted code in symbolic form. Default: OFF

MIX <ON or OFF>

ON specifies that the generated assembly source is to contain Pascal source lines as comments. The program heading and any source lines before the heading are not included. Default: OFF (see KEEPASMB option)

PAGE

Causes the listing to resume at the top of the next page (if LIST is ON).

PARTIAL_EVAL <ON or OFF>

ON suppresses the evaluation of the right operand of an AND (OR) operator when the left operand is FALSE (TRUE). OFF causes all operands of a Boolean expression to be evaluated. Default: ON

PASCAL <string>

If this option is specified, it must appear before the program heading. The system utilities COMPL and CLOAD look for:

```
$PASCAL
```

as the first characters of a source file in order to schedule the Pascal compiler.

The optional <string> is used to place information in the NAM record of the program unit. If the first character of the string is a comma (,) then the string specifies the contents of the NAM statement following the name field. This includes the program type, the priority, other options and a comment:

```
$PASCAL ',4,89 My Program'$
PROGRAM trial (input, output);
```

will generate a NAM statement of the form:

```
NAM TRIAL,4,89 My Program                                YYMMDD.HHMM
```

where YYMMDD.HHMM is a date/time stamp provided by the compiler on all NAM records. If the first character is NOT a comma (,) then the string specifies only the comment field of the NAM statement; the relocatable type, followed by a space, will be provided by the compiler.

```
$PASCAL 'Sample Program', SUBPROGRAM$
PROGRAM sample (input, output);
```

will generate a NAM statement of the form:

```
NAM SAMPL,7 Sample Program                               YYMMDD.HHMM
```

Only the first 40 characters of the string will be used in either case.

Note: The use of the first form requires specification of the relocatable type by the user. This must agree with the type of program unit being compiled. The use of the SEGMENT or SUBPROGRAM option is still required if the program unit is to be one of those types, but the option will not be checked against the user-supplied NAM statement. If the user-specified relocatable type does not agree with the program unit type known to the compiler the results are unpredictable.

RANGE <ON or OFF>

ON specifies that run-time checks of array indices, subrange assignments, actual value parameters, and pointer dereferencing are to be performed. Note: Range-checking is always performed for set expressions upon assignment whether or not this option is ON. Default: ON

RECURSIVE <ON or OFF>

ON specifies that subsequent routines can be called recursively. OFF specifies that subsequent routines cannot be called recursively. For the option to take effect for a given routine it must be set or reset appropriately before the block or directive. If a routine is declared FORWARD, the RECURSIVE option must be set appropriately at the FORWARD declaration. The RECURSIVE option has no effect on routines declared EXTERNAL. A RECURSIVE routine cannot be DIRECT. Default: ON

SEGMENT

Specifies that the current compilation unit is a segment, rather than a subprogram or main program. A segment is identical to a subprogram except that a "segment main" is created by the compiler that returns to the point following the segment load call (See Chapter 3). A type 5 relocatable is produced for a segment. Each level-1 routine in a segment unit is normally an entry point (see VISIBLE option). If this option is specified, it must appear before the program heading. Default: compilation unit is a main program

STATS

Specifies that the state of the compiler options and certain configuration information for the current compilation be displayed at the end of the compilation (the state of the options is displayed in the listing file, the configuration information both in the listing and to the terminal). If this option is specified, it must appear before the program heading. The STATS information will only appear in the listing file if LIST is ON at the end of the compilation.

Example:

```
                Dynamic Memory Allocation

          102 Pages of EMA.
           20 Pages requested.
           23 Pages allocated.
           10 Pages for one more memory segment.
           36 Pages for one less memory segment.

Compilation started:   Sat Feb 23, 1980   1:48 pm
Compilation completed: Sat Feb 23, 1980   1:49 pm
```

COMPILER OPTIONS:

```
ANSI            OFF
ASMB            R
AUTOPAGE        OFF
BUFFERS         1
CODE            ON
HEAP            1
HEAPPARMS       ON
IDSIZE          150
IMAGE           0
KEEPASMB        OFF
LINESIZE        128
LIST            ON
LIST_CODE       OFF
MIX             ON
PARTIAL_EVAL    ON
RANGE           ON
RECURSIVE       ON
STATS           ON
SUBTITLE
TABLES          ON
TITLE           Sample Program
TRACE           OFF
UNIT            PROGRAM
VISIBLE         ON
WIDTH           80
XREF            OFF
```

The "Dynamic Memory Allocation" information tells the user that:

The compiler had 102 pages of EMA space. This can be modified by the SZ operator command.

The user had requested a minimum of 20 pages for the heap/stack area. This value can be changed by specifying another value in the option file.

After the configuration process, 23 pages were actually allocated for the heap/stack area. This is because 3 pages were left over from the memory segments area after as many segments as possible were made memory segments. A user request of 24 pages may cause one less segment to be memory resident or perhaps a smaller segment (that was disc resident) to become memory resident. In either case 24 pages is next request that will change the actual allocation.

If the user had requested a minimum of 10 pages for the heap/stack area (which would have resulted in a larger memory segments area), then one more segment would have been made memory resident. This would have resulted in a decrease in compiler workspace and a possible increase in the speed of compilation.

If the user had requested a minimum of 36 pages for the heap/stack area (which would have resulted in a smaller memory segments area), then one less segment would have been made memory resident. This would have resulted in an increase in compiler workspace and a possible decrease in the speed of compilation.

SUBPROGRAM

Specifies that the current compilation unit is a subprogram. A subprogram is identical to a main program except that there is no body for the main (See Chapter 3). A type 7 relocatable is produced for a subprogram. Each level-1 routine in a subprogram is normally an entry point (see VISIBLE option). If this option is specified, it must appear before the program heading. Default: compilation unit is a main program

SUBTITLE <string>

The first 25 characters of the <string> will be printed under the TITLE string (see below), if any, at the top of the next and subsequent pages of the listing. Default: all blanks

TABLES <ON or OFF>

ON specifies that symbol table information is to be displayed following routines and/or the program. The option must be ON before the final semicolon of a routine (or period in the case of the program) and LIST must be ON for the appropriate TABLES to be displayed. Default: OFF

Example:

```
 1 00000 PROGRAM example (INPUT, OUTPUT);
 2 00676 {
 3 00676 program to demonstrate the different kind of table entries
 4 00676 }
 5 00676
 6 00676 CONST
 7 00676    exclamation = '!';
 8 00676    perfect     = 8128;
 9 00676    pi          = 3.1415926;
10 00676    big         = 123456789L10;
11 00676    string      = 'A string indeed';
12 00676
13 00676 TYPE
14 00676    POSINT  = 0..32767;
15 00676    CHARSET = SET OF CHAR;
16 00676    ANARRAY = PACKED ARRAY [1..100] OF BOOLEAN;
17 00676    ARECORD = RECORD
18 00676                 one: INTEGER;
19 00676                 CASE two: POSINT OF
20 00676                    10: (ten:    POSINT);
21 00676                    20: (twenty: REAL);
22 00676                    30: (thirty: LONGREAL);
23 00676              END;
24 00676
25 00676 CONST
26 00676    frano = ANARRAY [50 OF false, 50 OF true];
27 00676
28 00676 VAR
29 00676    once:    CHARSET;
30 00717    upon:    TEXT;
31 01256    a_time:  ARECORD;
32 01265    year:    1900..2000;
33 01266    era:     (past, present, future);
34 01267
35 01267 PROCEDURE donothing;
36 00000 BEGIN {donothing}
37 00027 END;  {donothing}
38 00047
39 00047 $ TABLES ON $ {only want main program tables}
40 00047 BEGIN {example}
41 00100 END.  {example}
```

TABLES continued

GLOBAL IDENTIFIER TABLES

```
ANARRAY              <1> TYPE
<1>                 6/ 4 PACKED ARRAY <3> OF <2>
<3>                 0/ 7 SUBRANGE 1..100 OF <4>
<4>                 1/ 0 SCALAR STANDARD
<2>                 0/ 1 BOOLEAN
ARECORD              <5> TYPE
<5>                 7/ 0 RECORD
ONE                  <6> FIELD 0/2
<6>                 2/ 0 INTEGER
TEN                  <7> FIELD 3/1
<7>                 0/15 SUBRANGE 0..32767 OF <4>
THIRTY               <8> FIELD 3/4
<8>                 4/ 0 LONGREAL
TWENTY               <9> FIELD 3/2
<9>                 2/ 0 REAL
TWO                  <7> FIELD 2/1
<10>                3/ 0 TAG <7>
<11>                4/ 0 VARIANT 10
<12>                5/ 0 VARIANT 20
<13>                7/ 0 VARIANT 30
A_TIME               <5> ACTUAL VARIABLE @1+686
BIG                  <8> CONSTANT  1.23456789000000000L+18
CHARSET             <14> TYPE
<14>               17/ 0 SET OF <15>
<15>                0/ 8 CHAR
DONOTHING
                         DECLARED PROCEDURE ACTUAL RECURSIVE
ERA                 <17> ACTUAL VARIABLE @1+694
<17>                0/ 2 SCALAR DECLARED
EXCLAMATION
                    <15> CONSTANT '!'
FRANO                <1> CONSTANT ARRAY @6+0
FUTURE              <17> CONSTANT 2
ONCE                <14> ACTUAL VARIABLE @1+446
PAST                <17> CONSTANT 0
PERFECT              <4> CONSTANT 8128
PI                   <9> CONSTANT 3.1415925L+00
POSINT               <7> TYPE
PRESENT             <17> CONSTANT 1
STRING              <18> CONSTANT STRING @2+6
<18>                7/ 8 PACKED ARRAY <19> OF <15>
<19>                1/ 0 SUBRANGE 1..15 OF <4>
UPON                <20> ACTUAL VARIABLE @1+463
<20>              223/ 0 TEXT
YEAR                <22> ACTUAL VARIABLE @1+693
<22>                0/11 SUBRANGE 1900..2000 OF <4>
```

A TABLES listing contains two types of entries: identifier entries and type entries. The first line of the example is an identifier entry:

```
ANARRAY                 <1> TYPE
```

An identifier entry has three fields. The first is the identifier itself. If an identifer is longer than eight characters it is listed alone on a line and the other two identifier entry fields follow on the next line (see DONOTHING and EXCLAMATION for examples). The second field is the type number, for ANARRAY this is <1>, and it is used to locate the type entry for the identifier. The third field is the class of the identifer, ANARRAY is an identifier which defines a type. Other possibilities for identifier class can be seen in the table.

Identifier entries are listed in alphabetical order with one exception. When a record type is encountered, the fields of the record are listed in alphabetical order at that point. The fields of ARECORD are an example (ONE, TEN, THIRTY, TWENTY, and TWO); then the identifier entries go back into alphabetical order with A_TIME.

The second line of the example is a type entry:

```
<1>                     6/ 4 PACKED ARRAY <3> OF <2>
```

A type entry has three fields. The first is the type number (referred to above). Type entries are only listed once, immediately following the first identifier entry of the type. The type number is used to reference a type entry from an identifier of that type appearing later in the list. The second field is the minimum size of the type in the form words/bits. An object of the specified type within a PACKED object can have its minimum size (see Chapter 4 PACKED TYPES). Objects not within PACKED objects will have an actual size rounded up to the next number of whole words. The third field is a description of the type. In this case a "packed array" with an index type of <3> and an element type of <2>.

Record type entries are followed by their fields in alphabetical order:

```
fieldid <typenum> field <word-offset>/<words>/<bit-offset>/<bits>
```

The word offset of the start of the field and the number of words the field occupies are always specified. If the record is PACKED then the bit offset of the start of the field (from the word offset), and the number of bits the field occupies are specified. If the record has a VARIANT part, the VARIANT selection constants are listed next:

```
<typenum> <words>/<bits> variant <tag value>
```

The <typenum> is arbitrary and is not referenced anywhere. The size field is the minimum size of the entire record when that particular VARIANT is selected. The <tag value> is the value of the tag that corresponds to the particular VARIANT.

The standard scalar types appear in the tables with their predefined names in the type entry. The type entry that indicates a one-word subrange of INTEGER will appear as 1/0 SCALAR STANDARD.

Structured constants and variables occupy storage and their identifier entry specifies where this storage starts. The compiler generates two kinds of labels. If a structured constant or variable is global it is located at an offset from a label of the form "@DDDD", if it is not global (i.e. cannot be accessed from other compilation units) it is located at an offset from a label of the form ".DDDD". The offset is a decimal value. Structured constants and variables declared while CODE is OFF are not guaranteed to have accurate label entries.

TITLE <string>

The first 25 characters of the <string> will be printed at the top of the next and subsequent pages of the listing. Default: all blanks

.TRACE <integer>

The <integer> specifies the LU number where a history of routine entries and exits are displayed. LU 0 (default) specifies that no trace is desired. If the option is used, calls to the following routines are placed in the emitted code:

```
TYPE
   LU    = 0..63;
   LEN   = 1..64;
   STR64 = PACKED ARRAY [1..64] OF CHAR;

PROCEDURE trace_init   {Call emitted at start of program      }
   $ALIAS '^TIN'$$
   (trace_lu: LU);     {LU of trace output                    }
   EXTERNAL;

PROCEDURE trace_close  {Call emitted at end of program        }
   $ALIAS '^TCL'$
   EXTERNAL;

PROCEDURE trace_begin  {Call emitted at start of each routine}
   $ALIAS '^TBG'$
   (name_len: LEN;     {Length of routine name                }
    name    : STR64);  {Name of routine                       }
   EXTERNAL;
```

```
PROCEDURE trace_end     {Call emitted at end of each routine  }
   $ALIAS '^TND^$
   (name_len: LEN;      {Length of routine name               }
    name    : STR64);   {Name of routine                      }
   EXTERNAL;
```

If any compilation unit of a program has TRACE on, and one of the standard trace packages is to be used (%TRACA, %TRACB, or %TRACC), the main program must have TRACE on to the same LU when the BEGIN and END of the main program are encountered to ensure that the necessary calls to ^TIN and ^TCL are emitted. This is the user's responsibility as the compiler cannot know when compiling the main whether tracing was specified in any separately compiled subprogram or segment. Default: 0

VISIBLE <ON or OFF>

ON specifies that subsequent level-1 routines are to be entry points and thus must have names unique within the first five characters. OFF specifies that subsequent routines are not entry points, and are not subject to the five character name system restriction (and are thus not accessible from other program units). If a routine is declared FORWARD, the VISIBLE option must be set appropriately at the FORWARD declaration. Default: ON

WIDTH <integer>

The <integer> specifies the number of significant characters in a source line (range 20 to 150). Additional characters on the line are ignored. This option takes effect starting with the line it appears on. Default: 80

XREF

Specifies that a cross reference of the program will be generated at the end of the listing (see Appendix F). If this option is specified, it must appear before the program heading. Default: no cross reference

Appendix E
Program To Program Communication

This appendix illustrates a major application of EXEC calls in Pascal/1000 programs. Program-to-program communication via message queues is implemented with class I/O calls.

The following four procedures manipulate the queue:

qallocate	Allocate a queue.
qrelease	Release (deallocate) a queue.
send	Enter a message into the tail of a queue.
receive	Retrieve a message from the head of a queue.

These routines can be conveniently packaged as a subprogram unit, as they are in this appendix.

```
$ SUBPROGRAM, RECURSIVE OFF $

PROGRAM ptopc;

{
*****************************************
*                                       *
*             Routines for              *
*    PROGRAM-TO-PROGRAM COMMUNICATION   *
*                                       *
*****************************************
}

CONST
   no_deallocate_bit = 8192;      {bit 13}
   no_abort_bit      = -32768;    {bit 15}

   exec18 = 18 + no_abort_bit;    {class write}
   exec20 = 20 + no_abort_bit;    {class write/read}
   exec21 = 21 + no_abort_bit;    {class get}

   maxmsglen = 80;

TYPE
   INT     = -32768..32767;
   POSINT  = 0..32767;
   LENGTH  = 0..maxmsglen;
   MSGTYPE = PACKED ARRAY [1..maxmsglen] OF CHAR;

PROCEDURE class_write
   $ALIAS 'EXEC'$
   (icode,
    icnwd : INT;
    ibuff : MSGTYPE;
    ilen,
    iop1,
    iop2,
    iclas : INT);
   EXTERNAL;

PROCEDURE class_write_read
   $ALIAS 'EXEC'$
   (icode,
    icnwd : INT;
    ibuff : MSGTYPE;
    ilen,
    iop1,
    iop2,
    iclas : INT);
   EXTERNAL;
```

```
PROCEDURE class_get
   $ALIAS 'EXEC'$
   (    icode : INT;
        iclas : POSINT;
    VAR ibuff : MSGTYPE;
        ilen  : INT;
    VAR iop1,
        iop2  : INT);
   EXTERNAL;

PROCEDURE abreg
   (VAR a_reg : INT;
    VAR b_reg : LENGTH);
   EXTERNAL;

PROCEDURE error_handler;
   $DIRECT$
   EXTERNAL;
```

```
{
*****************************************************************
* QALLOCATE  *
**************

Allocate  a  queue  by reserving a queue  id (class number).  This is
accomplished  by a class write with  a dummy buffer (length specified
to  be  0)  and a 0 queue id.   The no-deallocate bit of the returned
queue id is set on, and the class write is completed with a class get
(also with the dummy buffer).

The returned queue id is to be used unchanged in all future references
to the queue.
}

PROCEDURE qallocate        {Purpose - Allocate a queue}
   (VAR q_id : POSINT);    {Output  - Queue id        }

VAR
   zero         : INT;
   dummy_buffer : MSGTYPE;   {Dummy buffer}

BEGIN {qallocate}

   q_id := 0;
   zero := 0;

   {Allocate a queue id (class number).}
   class_write (exec18, 0, dummy_buffer, 0, 0, 0, q_id);
   error_handler;

   {Set no-deallocate bit.}
   q_id := q_id + no_deallocate_bit;

   {Complete above class write.}
   class_get (exec21, q_id, dummy_buffer, zero, zero, zero);
   error_handler;

END {qalloc};
```

```
{
*****************************************************************
* QRELEASE *
************
```

Release a queue. This is accomplished by turning off the no-deallocate bit of the queue id, and using it in a class write with a dummy buffer. The class write is completed with a class get, also with a dummy buffer.

The caller must first ensure that the queue to be released is empty, and that there are no requests still pending on it.

```
}

PROCEDURE qrelease    {Purpose - Release a queue}
   (q_id : POSINT);   {Input   - Queue id       }

VAR
   zero         : INT;
   dummy_buffer : MSGTYPE;   {Dummy buffer}

BEGIN

   {Turn off no-deallocate bit.}
   q_id := q_id - no_deallocate_bit;

   {Release the queue.}
   class_write (exec18, 0, dummy_buffer, 0, 0, 0, q_id);
   error_handler;

   {Complete above class write.}
   class_get (exec21, q_id, dummy_buffer, zero, zero, zero);
   error_handler;

END {qrelease};
```

```
{
*************************************************************************
* SEND *
********

Enter  a  message, along with a sender id  and a message id, into the
tail of a queue.  This is accomplished by a class write/read with the
appropriate queue id.

If  the queue is already full, the  routine will wait until the queue
becomes nonfull.
}

PROCEDURE send                          {Purpose - Send a message      }
   (q_id             : POSINT;          {Input   - Queue id            }
    sender_id,                          {Input   - Sender id           }
    message_id       : INT;             {Input   - Message id          }
    message          : MSGTYPE;         {Input   - Message to be sent  }
    message_length   : LENGTH);         {Input   - Char. length of msg.}

BEGIN {send}

   {Send the message, along with the sender id and the message id.}
   class_write_read (exec20, 0, message, -message_length,
                     sender_id, message_id, q_id);
   error_handler;

END {send};
```

```
{
**************************************************************************
* RECEIVE *
***********

Retrieve a message, along with the sender id and the message id, from
a  queue.  This is accomplished by a  class get, which completes the
class write/read of a previous call to "send".

If the queue is already empty, the routine will wait until the queue
becomes nonempty.
}
PROCEDURE receive                          {Purpose - Receive a message   }
   (    q_id             : POSINT;         {Input   - Queue id            }
    VAR sender_id,                         {Output  - Sender id           }
        message_id       : INT;            {Output  - Message id          }
    VAR message          : MSGTYPE;        {Output  - Message received    }
    VAR message_length   : LENGTH);        {Output  - Char. length of msg.}

VAR
   dummy : INT;

BEGIN
   {Retrieve the message, along with the sender id and the mesage id.}
   class_get (exec21, q_id, message, -maxmsglen,
              sender_id, message_id);
   error_handler;

   {Retrieve message length from B register.}
   abreg (dummy, message_length);
END {receive};
.   {end of subprogram}
```

The sample programs in this section use the program-to-program communication subprogram to send and receive message via class I/O. Using the INCLUDE compiler option and subprograms one can easily set up a variety of similar programmer utilities.

```
$RECURSIVE OFF$
PROGRAM dadptp (INPUT, OUTPUT);

{
DADPT is a sample program that uses the class I/O subprogram, PTOPC",
to do program to program communication.  By using this subprogram
the program can be broken down into 5 simple steps:

  1. Allocate a queue (class number).
  2. Give the user instructons.
  3. Read and send messages.
  4. Schedule son program to pick up messages.
  5. Release the queue (class number).
}

CONST
   max_length = 80;                    { maximum message length      }
   terminator = 'XX';                  { program terminating message }
   no_abort_bit = -32768;              { bit 15, 100000 octal        }
   exec23 = 23 + no_abort_bit;
   sons_name = 'SONPTP';               { name of son to be scheduled }

TYPE
   INT = -32768..32767;
   POSINT = 0..32767;
   MESG_LENGTH = 1..max_length;
   NAME_TYPE = PACKED ARRAY [1..6] OF CHAR;
   MESG_TYPE = PACKED ARRAY [MESG_LENGTH] OF CHAR;
   ASCII_WORD = PACKED ARRAY [1..2] OF CHAR;

CONST
   blank_message = MESG_TYPE [max_length OF ' '];

VAR
   q_id          : POSINT;
   mlength,
   sender_id,
   message_id    : INT;
   message       : MESG_TYPE;
```

```
PROCEDURE qallocate
   (VAR q_id : POSINT);
   EXTERNAL;

PROCEDURE qrelease
   (VAR q_id : POSINT);
   EXTERNAL;

PROCEDURE send
   (q_id          : POSINT;
    sender_id,
    message_id    : INT;
    message       : MESG_TYPE;
    mlength       : MESG_LENGTH);
   EXTERNAL;

PROCEDURE abreg
   (VAR areg,
        breg : ASCII_WORD);
   EXTERNAL;

PROCEDURE prog_sched  $ALIAS 'EXEC'$
   (exec_code : INT;
    name      : NAME_TYPE;
    q_id      : POSINT);
   EXTERNAL;

PROCEDURE error_handler;
   $DIRECT$

VAR
   areg,
   breg : ASCII_WORD;

BEGIN
   abreg (areg, breg);
   writeln('**ERROR IN EXEC CALL!  ERROR CODE IS : ',areg:3, breg:3);
   writeln('Class number is ',q_id);
   halt(13);
END  {error_handler};
```

```
BEGIN
   {allocate a queue (class number) for the messages}
   qallocate (q_id);

   {give the user instructions}
   writeln('Enter a message of ',max_length:0,' characters');
   writeln('Enter ',terminator,' to terminate program.');

   REPEAT

      {erase the old message}
      message := blank_message;

      {prompt user and read in message}
      write ('>');
      prompt;
      read (message);

      {send the message}
      send (q_id, sender_id, message_id, message, max_length);

   UNTIL message = terminator;

   {schedule sonptp}
   prog_sched (exec23, sons_name, q_id);
   error_handler;

   {release the class number}
   qrelease (q_id);

END  {dadptp}
.
```

```
$RECURSIVE OFF$
PROGRAM sonptp;

{
SONPTP is a program scheduled by DADPTP. SONPTP also uses the
subprogram PTOPC to do class I/O. SONPTP can be broken down into 3
steps:

   1. Get the queue number passed by DADPTP.
   2. Associate output with LU 1.
   3. Receive and print messages.
}

CONST
   max_length = 80;                    { maximum message length     }
   terminator = 'XX';                  { indicates the last message }

TYPE
   INT = -32768..32767;
   POSINT = 0..32767;
   MESG_LENGTH = 1..max_length;
   MESG_TYPE = PACKED ARRAY [MESG_LENGTH] OF CHAR;
   ASCII_WORD = PACKED ARRAY [1..2] OF CHAR;
   PARM_ARRAY = ARRAY [1..5] OF INT;

VAR
   q_id          : POSINT;
   mlength,
   sender_id,
   message_id    : INT;
   message       : MESG_TYPE;
   parms         : PARM_ARRAY;
   output        : TEXT;

PROCEDURE receive
   (    q_id          : POSINT;
    VAR sender_id,
        message_id    : INT;
    VAR message       : MESG_TYPE;
    VAR mlength       : INT);
   EXTERNAL;

PROCEDURE rmpar
   (VAR parms : PARM_ARRAY);
   EXTERNAL;

PROCEDURE abreg
   (VAR areg,
        breg : ASCII_WORD);
   EXTERNAL;

PROCEDURE error_handler;
   $DIRECT$
```

```
VAR
   areg,
   breg : ASCII_WORD;

BEGIN
   abreg (areg, breg);
   writeln(output,'**ERROR IN EXEC CALL!  ERROR CODE IS : ',
           areg:3, breg:3);
   writeln(output,'Class number is ',q_id);
   halt(13);
END  {error_handler};

BEGIN

   {get the class number passed by dadptp}
   rmpar (parms);
   q_id := parms [1];

   {associate the file output with LU 1}
   rewrite (output,'1');

   {receive and send messages until the terminating message is received}
   REPEAT
      receive (q_id, sender_id, message_id, message, mlength);
      writeln(output,'Message > ',message:mlength)
   UNTIL message = terminator;

END  {sonptp}
.
```

Appendix F
User-Callable Pascal/1000 Library Routines

The Pascal library contains several routines that are directly callable from user programs. The external names of these routines as given below may be aliased to user-chosen valid Pascal procedure or function names.

RSPAR Run string parameter (function)

This function either retrieves the entire run string, or it extracts selected parameters from the run string. The function returns a one-word integer value that is the character length of the run string or selected parameter. If the selected character parameter does not exist, 0 is returned.

Note that the "run string" that this function operates upon is either the run string typed at the terminal when the program is scheduled interactively, or the "optional buffer" when the program is scheduled programmatically with a call to EXEC 9, 10, 23, or 24. In the latter case, the function expects the buffer to include the entire run string that one would type at the terminal, including the beginning "RU,<program name>,".

1) Position. A value parameter of type one-word integer that passes the position of the desired parameter, where 0 is the program name, 1 is the first parameter, 2 is the second parameter, etc. If any negative position value is passed, the entire run string is retrieved.

2) Parameter. A VAR parameter of type PACKED ARRAY [1..80] OF CHAR that contains upon return either the entire run string or a selected parameter, depending on the value of position (above). If the entire run string is returned, all blanks around the comma separators are stripped. If a selected parameter is returned, leading and trailing blanks are stripped. If the selected parameter does not exist, the return value is undefined.

3) Length. A VAR parameter of type one-word integer that passes the maximum number of characters to be returned in parameter (above). (The actual number of characters returned is the function value.) If fewer characters than specified are returned, parameter (above) is blank-filled.

```
FUNCTION rspar
   (    position  : INT;
    VAR parameter : STRING80;
        length    : INT)
   : INT;
   EXTERNAL;
```

where: TYPE INT = -32768..32767;

and TYPE STRING80 = PACKED ARRAY [1..80] OF CHAR;

Note: The $HEAPPARMS OFF$ compiler option must be in effect for the VAR parameter if the $HEAP 2$ option is in effect.

@GHS1 Get heap 1/stack information (procedure)

This procedure, to be called from $HEAP 1$ programs only, returns a record containing current heap and stack information parameters.

```
TYPE
   ADDR1 = 0..32767;              {one-word logical addr. }

   INFO_REC1 =
      RECORD
         tos,                     {top of stack          }
         toh,                     {top of heap           }
         init_tos,                {initial top of stack  }
         init_toh,                {initial top of heap   }
         high_tos,                {highest top of stack  }
         high_toh,                {highest top of heap   }
         free_block,              {current free list block}
         mark_block : ADDR1;      {current mark list block}
      END;

   PROCEDURE get_heap_1_stack_info
      $ALIAS '@GHS1'$
      (VAR heap_info1 : INFO_REC);
      EXTERNAL;
```

NOTES: The stack grows toward increasing addresses, and the heap grows toward decreasing addresses.

The addresses of the current top of stack and top of heap are returned in fields tos and toh, respectively. The initial top of stack and top of heap addresses are returned in fields init_tos and init_toh, respectively. Their "high_water marks" are returned in high_tos and high_toh, respectively.

At all times:

```
init_tos <= tos <= high_tos
high_toh <= toh <= init_toh
tos < toh
```

The addresses of the current free space block and the current mark block are returned in fields free_block and mark_block, respectively.

@SHS1 Set heap 1/stack information (procedure)

This procedure, to be called from $HEAP 1$ programs only, allows the user to set the heap and stack information parameters described above. It is to be called only by users desiring to do their own heap and stack management.

```
PROCEDURE set_heap_1_stack_info
   $ALIAS '@SHS1'$
   (heap_info1 : INFO_REC1);
   EXTERNAL;
```

@INH1 Initialize heap 1/stack information (procedure)

This parameterless procedure, to be called from $HEAP 1$ programs only, initializes heap and stack information. Its effects are described in Chapter 8.

```
PROCEDURE initialize_heap_1_stack_info
   $ALIAS '@INH1'$
   EXTERNAL;
```

@GHS2 Get heap 2/stack information (procedure)

This procedure is similar to @GHS1, except that it is to be called from $HEAP 2$ programs only. It returns a record containing current heap and stack information parameters.

```
TYPE
   ADDR2 = 0..maxint;          {two-word EMA address    }

   INFO_REC2 =
      RECORD
         tos,                  {top of stack            }
         toh,                  {toh of heap             }
         init_tos,             {initial top of stack    }
         init_toh,             {initial top of heap     }
         high_tos,             {highest top of stack    }
         high_toh,             {highest top of heap     }
         free_block,           {current free list block }
         mark_block : ADDR2;   {current mark list block }
      END;

PROCEDURE get_heap_2_stack_info
   $ALIAS '@GHS2'$
   (VAR heap_info2 : INFO_REC2);
   EXTERNAL;
```

@SHS2 Set heap 2/stack information (procedure)

This procedure is similar to @SHS1, except that it is to be called from $HEAP 2$ programs only. It allows users desiring to do their own heap and stack management to set the heap and stack information parameters described above.

```
PROCEDCURE set_heap_2_stack_info
   $ALIAS '@SHS2'$
   (heap_info2 : INFO_REC2);
   EXTERNAL;
```

@INH2 Initialize heap 2/stack information (procedure).

This procedure is similar to @INH1 above, except that it is to be called by $HEAP 2$ programs only. The procedure initializes heap and stack information, and is described in Chapter 8.

```
PROCEDURE initialize_heap_2_stack_information
   $ALIAS '@INH2'$
   EXTERNAL;
```

@TIME Time (procedure)

This procedure requires a VAR parameter of type PACKED ARRAY [1..26] OF CHAR in which the current time is returned. A sample time string:

```
       Sun Jan  6, 1980   3:01 pm
```

```
PROCEDURE get_time
   $ALIAS '@TIME'$
   (VAR time_string : STRING26);
   EXTERNAL;
```

@SGLD Segment load (procedure)

This procedure is used to load a Pascal compilation unit that was compiled with the $SEGMENT ON$ option. It requires a parameter of type PACKED ARRAY [1..5] OF CHAR in which the program name of the compilation unit is passed.

```
TYPE
   SEG_NAME = PACKED ARRAY [1..5] OF CHAR;

PROCEDURE seg_load
   $ALIAS '@SGLD'$
   (name: SEG_NAME);
   EXTERNAL;
```

Appendix G
Pascal/1000 Cross-Referencer

The Pascal/1000 Cross-Referencer (PXREF) produces a cross-reference table for each procedure and function in a Pascal/1000 program, and for the main program itself.

The cross-referencer can be run from file manager or it can be scheduled from the compiler by including the $XREF ON$ compiler option in the program.

Cross-Reference Table Content

For each program block a cross-reference table is generated. The table consists of a list of all identifiers local to the block, and any predefined or non-local identifiers that are used in the block. For each identifier in the table, the following information is given:

1. THE KIND OF IDENTIFIER IT IS -- Label, Constant, Type, Variable, Record field, Tag field, Enumerated Constant, Program, Procedure, Function, Program Parameter, Formal Variable by Value, Formal Variable by Address, Formal Function, Formal Procedure, or Formal Procedure or Function Parameter.

2. THE BLOCK AND LINE NUMBER WHERE IT IS DECLARED -- (Not given for predefined identifiers.) The 'kind' of block in which it is declared is given also. For example,

   ```
   ALPHA
     Variable declared at 56 by Procedure ONE
   ```

   A procedure or function that is declared forward or external is so labelled. For example,

   ```
   EXEC_READ
     Declared external at 10 by Program STAT
   ```

3. LINE NUMBERS OF LINES WHERE IT IS USED IN THE BLOCK -- This includes occurrences of identifiers in blocks nested within the block being cross-referenced. For example, a program's cross-reference table lists uses of global variables in procedures as well as in the body of the program.

Following the cross-reference table for the main program is a list of all predefined identifiers used in the program. Each such identifier is followed by its 'kind' and the line number for each occurrence.

Order of Identifiers

Identifiers are listed in alphabetical order. Numbers precede letters and letters precede the underscore ('_'). (Note that since '0' precedes '1' alphabetically, 'ident101' precedes 'ident11' despite the fact that 11 is less than 101.) When two or more identifiers have the same name, they are listed in the order in which they were declared.

Paging

The cross-reference table or listing of any block that is not a formal or external procedure or function always begins on a new page.

Using the Cross-Reference Generator

There are two ways to obtain a cross-reference of a program: run the cross-referencer by itself or have the compiler run it when it compiles the program. The command which runs the cross-referencer by itself is:

RU,PXREF,<source>,<cross-reference>[,{L, N}]

where <source> is a Pascal source file and <cross-reference> is the file or device on which the cross-reference will be printed. If 'L' is specified, each block's listing will be printed before its cross-reference; if 'N' is specified, it will not. Default is L when the cross-referencer is run by itself, but N is the only option used by the compiler. There is no way to have the compiler run the cross-referencer with the L option.

To have the compiler run the cross-referencer, include the option '$ XREF ON $' or '$ XREF $' anywhere in the source. (Default is '$ XREF OFF $'.)

Errors and Warnings

The cross-referencer does not detect all syntax errors. A program which compiles without errors will cross-reference with neither warnings nor errors, but the converse is not necessarily true.

The cross-referencer prints a warning if:

- an input line is too long (the entire line is printed, but the extra characters are ignored, which may result in errors).
- an identifier is used without having been declared.
- an identifier is declared more than once.
- a procedure or function is declared FORWARD but never appears in the program.

EMA Version

The cross referencer maintains internal symbol tables in memory. The HEAP 1 version of the cross-referencer can run out of space for its tables and be forced to abort. There are two solutions to this condition. There is a HEAP 2 version of the cross referencer that will cross reference programs as large as its EMA partition will permit. For a large EMA partition, this size is probably larger than any practical program will ever require. The compiler will use whichever version is currently called PXREF, so the HEAP 1 or the HEAP 2 version can be made the default used by the compiler. The disadvantage of using the HEAP 2 version is that it required an EMA partition. Since the Pascal compiler requires an EMA partition and locks itself in the partition when running, a system with many Pascal compilations running would prevent the HEAP 2 cross-referencer from running at all. The HEAP 2 version is also slightly slower. Another possible solution is to size the HEAP 1 version (with the system SZ command) to occupy a larger partition. When the HEAP 1 version is loaded it requires 19 pages and is sized to 26 pages. If either 27 or 28 page partitions are available, it can be sized up to one of those values, and it should then be able to cross-reference slightly larger programs.

Refer to the Pascal/1000 Configuration Guide (part number 92832-90003) for information about loading either version of the cross-referencer.

Sample Cross-Reference Table (with blocks listed)

```
Pascal/1000 Cross Reference of &EXAMP

    1 PROGRAM example (library, output);
    2
    3 LABEL
    4    1, 2, 10;
    5
    6 CONST
    7    MIN = 0;
    8    MAX = 2;
    9
   10 TYPE
   11 RANGE   = MIN..MAX;
   12    POINTER = ^TREC;
   13
   14    TREC    = RECORD
   15                 char_field: CHAR;
   16                 enum_field: (choice1, choice2, choice3);
   17                 rec_field:  RECORD
   18                                char_field: CHAR;
   19                             END;
   20              END;
   21
   22 VAR
   23    library: FILE OF CHAR;
   24    n:       POINTER;
   25    vrec:    RECORD
   26                char_field: CHAR;
   27                CASE boo_field: BOOLEAN OF
   28                   true:  (t_field: INTEGER);
   29                   false: (f_field: CHAR);
   30             END;
   31
   32 FUNCTION factorial
   33    (x: INTEGER)
   34    : INTEGER; FORWARD;

Function FACTORIAL
Cross Reference

FACTORIAL
   Function declared forward at 32 by Program EXAMPLE

INTEGER
   Predefined Type
   Used at 33,34

X
   Formal Variable by Value declared at 33 by Function FACTORIAL
```

```
35
36 PROCEDURE fill_trec
37     (VAR rec: TREC;
38          c:   CHAR);
39
40 VAR
41 m, n: INTEGER;
42
43 FUNCTION choice_123 {within procedure fill_trec}
44     (n: INTEGER)
45     : RANGE;
46
47 BEGIN {choice_123}
48    choice_123 := (m + n) mod 3;
49 END; {choice_123}
```

```
Function CHOICE_123
Cross Reference

CHOICE_123
   Function declared at 43 by Procedure FILL_TREC
   Used at 48

INTEGER
   Predefined Type
   Used at 44

M
   Variable declared at 41 by Procedure FILL_TREC
   Used at 48

N
   Formal Variable by Value declared at 44 by Function CHOICE_123
   Used at 48

RANGE
   Type declared at 11 by Program EXAMPLE
   Used at 45
```

```
50
51 BEGIN {fill_trec}
52
53    m := ord(c);
54    n := factorial(m);
55
56    WITH rec DO BEGIN
57       char_field := c;
58
59       CASE choice_123(n) OF
60          MIN: enum_field := choice1;
61       MAX: enum_field := choice3;
62          OTHERWISE
63                enum_field := choice2;
64       END;
65
66       WITH rec_field DO BEGIN
67          char_field := c;
68       END;
69    END;
70 END; {fill_trec}
```

```
Procedure FILL_TREC
Cross Reference

C
   Formal Variable by Value declared at 38 by Procedure FILL_TREC
   Used at 53,57,67

CHAR
   Predefined Type
   Used at 38

CHAR_FIELD
   Record Field declared at 15 by Program EXAMPLE
   Used at 57

CHAR_FIELD
   Record Field declared at 18 by Program EXAMPLE
   Used at 67

CHOICE1
   Enumerated Constant declared at 16 by Program EXAMPLE
   Used at 60

CHOICE2
   Enumerated Constant declared at 16 by Program EXAMPLE
   Used at 63

CHOICE3
   Enumerated Constant declared at 16 by Program EXAMPLE
   Used at 61

CHOICE_123
   Function declared at 43 by Procedure FILL_TREC
   Used at 48,59

ENUM_FIELD
   Record Field declared at 16 by Program EXAMPLE
   Used at 60,61,63

FACTORIAL
   Function declared forward at 32 by Function FACTORIAL
   Used at 54

FILL_TREC
   Procedure declared at 36 by Program EXAMPLE
   *** NEVER USED ***

INTEGER
   Predefined Type
   Used at 41,44

M
   Variable declared at 41 by Procedure FILL_TREC
   Used at 48,53,54
```

```
MAX
   Constant declared at 8 by Program EXAMPLE
   Used at 61

MIN
   Constant declared at 7 by Program EXAMPLE
   Used at 60

N
   Variable declared at 41 by Procedure FILL_TREC
   Used at 54,59

ORD
   Predefined Function
   Used at 53

REC
   Formal Variable by Address declared at 37 by Procedure FILL_TREC
   Used at 56

REC_FIELD
   Record Field declared at 17 by Program EXAMPLE
   Used at 66

TREC
   Type declared at 14 by Program EXAMPLE
   Used at 37
```

```
71
72 FUNCTION factorial; {recursive}
73
74 BEGIN {factorial}
75    IF x = 0 THEN BEGIN
76       factorial := 1;
77    END
78    ELSE BEGIN
79       factorial := x * factorial(x - 1);
80    END;
81 END; {factorial}
```

```
Function FACTORIAL
Cross Reference

FACTORIAL
   Function declared at 72 by Program EXAMPLE
   Used at 76,79

X
   Formal Variable by Value declared at 33 by Function FACTORIAL
   Used at 75,79
```

```
 82
 83 BEGIN {example}
 84    reset(library);
 85
 86    WITH vrec DO BEGIN
 87       read(library, char_field);
 88       boo_field := true;
 89       t_field   := MAXINT;
 90    END;
 91
 92    new(n);
 93    fill_trec(n^, vrec.char_field);
 94
 95     1:  goto 2;
 96     2:  goto 10;
 97    10:  writeln('Demonstration of label use');
 98 END. {example}
```

```
Program EXAMPLE
Cross Reference

1
   Label declared at 4 by Program EXAMPLE
   Used at 95

10
   Label declared at 4 by Program EXAMPLE
   Used at 96,97

2
   Label declared at 4 by Program EXAMPLE
   Used at 95,96

BOOLEAN
   Predefined Type
   Used at 27

BOO_FIELD
   Tag Field declared at 27 by Program EXAMPLE
   Used at 88

CHAR
   Predefined Type
   Used at 15,18,23,26,29,38

CHAR_FIELD
   Record Field declared at 15 by Program EXAMPLE
   Used at 57

CHAR_FIELD
   Record Field declared at 18 by Program EXAMPLE
   Used at 67

CHAR_FIELD
   Record Field declared at 26 by Program EXAMPLE
   Used at 87,93

CHOICE1
   Enumerated Constant declared at 16 by Program EXAMPLE
   Used at 60

CHOICE2
   Enumerated Constant declared at 16 by Program EXAMPLE
   Used at 63

CHOICE3
   Enumerated Constant declared at 16 by Program EXAMPLE
   Used at 61

ENUM_FIELD
   Record Field declared at 16 by Program EXAMPLE
   Used at 60,61,63
```

```
EXAMPLE
   Program

FACTORIAL
   Function declared at 72 by Program EXAMPLE
   Used at 54,76,79

FALSE
   Predefined Enumerated Constant
   Used at 29

FILL_TREC
   Procedure declared at 36 by Program EXAMPLE
   Used at 93

F_FIELD
   Record Field declared at 29 by Program EXAMPLE
   *** NEVER USED ***

INTEGER
   Predefined Type
   Used at 28,33,34,41,44

LIBRARY
   Program Parameter declared at 23 by Program EXAMPLE
   Used at 1,84,87

MAX
   Constant declared at 8 by Program EXAMPLE
   Used at 11,61
MAXINT
   Predefined Constant
   Used at 89

MIN
   Constant declared at 7 by Program EXAMPLE
   Used at 11,60

N
   Variable declared at 24 by Program EXAMPLE
   Used at 92,93

NEW
   Predefined Procedure
   Used at 92

OUTPUT
   Predefined Program Parameter
   Used at 1
POINTER
   Type declared at 12 by Program EXAMPLE
   Used at 24
```

```
RANGE
   Type declared at 11 by Program EXAMPLE
   Used at 45

READ
   Predefined Procedure
   Used at 87

REC_FIELD
   Record Field declared at 17 by Program EXAMPLE
   Used at 66

RESET
   Predefined Procedure
   Used at 84

TREC
   Type declared at 14 by Program EXAMPLE
   Used at 12,37

TRUE
   Predefined Enumerated Constant
   Used at 28,88

T_FIELD
   Record Field declared at 28 by Program EXAMPLE
   Used at 89

VREC
   Variable declared at 25 by Program EXAMPLE
   Used at 86,93

WRITELN
   Predefined Procedure
   Used at 97
```

```
 Predefined Identifiers Used:

 BOOLEAN
    Predefined Type
    Used at 27

 CHAR
    Predefined Type
    Used at 15,18,23,26,29,38

 FALSE
    Predefined Enumerated Constant
    Used at 29

 INTEGER
    Predefined Type
    Used at 28,33,34,41,44

 MAXINT
    Predefined Constant
    Used at 89

 NEW
    Predefined Procedure
    Used at 92

 ORD
    Predefined Function
    Used at 53

 OUTPUT
    Predefined Program Parameter
    Used at 1
 READ
    Predefined Procedure
    Used at 87

 RESET
    Predefined Procedure
    Used at 84

 TRUE
    Predefined Enumerated Constant
    Used at 28,88

 WRITELN
    Predefined Procedure
    Used at 97


 End of Cross Reference
    98 Lines read
     0 Warnings/Errors
```

INDEX

READER COMMENT SHEET

Pascal/1000
Reference Manual

92832-90001 **May 1980**

Update No. _____
(If Applicable)

We welcome your evaluation of this manual. Your comments and suggestions help us improve our publications. Please use additional pages if necessary.

FROM:

Name _____

Company _____

Address _____

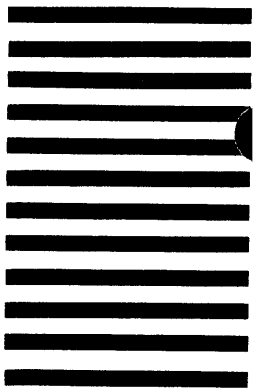
_____

_____

FOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 141 CUPERTINO, CA.

— POSTAGE WILL BE PAID BY —

Hewlett-Packard Company
Data Systems Division
11000 Wolfe Road
Cupertino, California 95014
ATTN: Technical Marketing Dept.

FOLD

FOLD

MANUAL PART NO. 92832-90001
Printed in U.S.A. May 1980

HEWLETT-PACKARD COMPANY
Data Systems Division
11000 Wolfe Road
Cupertino, California 95014