



Pascal/1000

Reference Manual

RTE-6/VM • RTE-A
HP 1000 Computer Systems



HEWLETT-PACKARD COMPANY
Computer Language Lab
19447 Pruneridge Avenue
Cupertino, California 95014

MANUAL PART NO. 92833-90005
Printed in U.S.A. September 1984
U0385

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 replacement pages to be merged into the manual, including an updated copy of this Printing History page. Also, the update may contain write-in instructions.

Each reprinting of this manual 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. New editions of this manual will contain new information, as well as all Updates.

To determine what software manual edition and update is compatible with your current software revision code, refer to the appropriate Software Numbering Catalog, Software Product Catalog, or Diagnostic Configurator Manual.

First Edition Sep 1984
Update 1 Mar 1985

NOTICE

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

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

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another program language without the prior written consent of Hewlett-Packard Company.

Preface

This is the Pascal reference manual for the HP 1000 Computer System. Pascal/1000 operates under the RTE-6/VM and RTE-A operating systems.

Information for the Pascal/1000 programmer can be found in the following documents:

- Language Standards
 - "American National Standard Pascal Computer Programming Language," ANSI/IEEE770X3.97-1983, Library of Congress Catalog Number 82-84259. This book defines ANSI Pascal which is the basis for HP Pascal and Pascal/1000.
- Language Tutorial
 - "Programming in Pascal with Hewlett-Packard" by Peter Grogono.
 - Many other Pascal tutorials are also available from most bookstores and libraries.
- Language Reference
 - Chapters 1 through 7 and Appendices B and C of this manual describe the language supported by Pascal/1000.
 - "HP Pascal Language Reference," HP Manual Part Number 98680-90015. This HP 9000 Series 200 computer manual contains a language reference manual for HP Pascal, and can be used as a second reference to this manual. (Where the two manuals disagree, the Pascal/1000 reference should be considered the more accurate one for this particular implementation.)
- Programmer's Guide
 - Chapters 8 and 9, and Appendices A and D of this manual describe implementation, usage, and efficiency details of Pascal/1000.
 - "Pascal/1000 Configuration Guide," HP Manual Part Number 92833-90003. This guide gives the hardware and software requirements for running Pascal/1000, installation instructions, and compiler performance tuning information.

This manual may be used for reference by the Pascal/1000 user and is organized as follows:

- Chapter 1 Introduces Pascal/1000. Table 1-1 gives an overview of the program vocabulary.
- Chapter 2 Discusses general form acceptable to the compiler.
- Chapter 3 Describes the compilation units in Pascal/1000.
- Chapter 4 Describes the declarations and definitions of the objects to be used by a program or routine.
- Chapter 5 Explains the executable parts of a program, routine or subprogram.
- Chapter 6 Discusses interface with outside objects via input/output files.
- Chapter 7 Defines standard procedures and functions.
- Chapter 8 Presents a detailed discussion of system implementation considerations.
- Chapter 9 Provides program development information.
- Appendix A Lists and describes run-time, I/O, and compiler error and warning messages as well as a suggested user response to help correct the error.
- Appendix B Presents all the syntax diagrams used in this manual.
- Appendix C Describes the compiler options and their usage.
- Appendix D Discusses the user callable library routines available in Pascal/1000.

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Table of Contents

Chapter 1 GENERAL INFORMATION	Page	Chapter 3 COMPILEATION UNITS	Page
Introduction	1-1	Introduction	3-1
Pascal Standards and Extensions	1-1	Main Program Unit	3-3
HP Pascal	1-2	Program Heading	3-4
Assignment Compatibility	1-2	Main Program Block	3-5
Type Compatibility	1-2	Module Unit	3-6
CASE Statement	1-2	Module Heading	3-6
Compiler Options (Directives)	1-2	Module Block	3-6
Constant Expressions	1-2	Subprogram Unit	3-7
Constructors (Structured Constants)	1-3	Subprogram Unit Program Heading	3-7
Declaration Part	1-3	Subprogram Unit Block	3-8
Halt Procedure	1-3	Segment Unit	3-10
Identifiers	1-3	Segment Unit Program Heading	3-10
Function Return	1-3	Segment Unit Block	3-11
Longreal Numbers	1-3	Loading Segment Overlays at Runtime	3-12
Minint	1-3		
File I/O	1-3		
String Type	1-4		
WITH Statement	1-4		
Numeric Conversion Functions	1-4		
EXTERNAL Routines	1-4		
Modules	1-5		
Record Variant Declaration	1-5		
String Literals	1-5		
Pascal/1000	1-6		
Nested Comments	1-6		
REAL, LONGREAL, constant expressions	1-6		
Separate Compilation	1-6		
PAC Types	1-6		
Parameters to reset, rewrite, open and append	1-6		
Compiler Options	1-6		
Pascal Program Vocabulary	1-7		
 Chapter 2 GENERAL FORM	 Page	 Chapter 4 DECLARATIONS	 Page
Introduction	2-1	Program Heading	4-1
Basic Symbols	2-2	Declaration Part	4-2
Reserved Words	2-3	Label Declaration	4-3
Identifiers	2-4	Constant Definition	4-4
Predefined Identifiers	2-6	Simple Constants	4-4
Predefined Symbolic Constants	2-6	Structured Constants	4-5
Predefined Types	2-6	Array Constant	4-6
Predeclared Variables	2-6	Record Constant	4-7
Predefined Procedures and Functions	2-6	Set Constant	4-8
Directives	2-7	String Constant	4-9
Numbers	2-8	Type Definition	4-10
String Literals	2-8	Predefined Types	4-11
Comments	2-8	Boolean	4-11
Compiler Options	2-9	Char	4-11
Separators	2-9	Integer	4-11

Table of Contents (Continued)

Procedure and Function Parameters	4-29	Selectors	5-29
Parameter List Compatibility	4-30	Array and String Subscripts	5-30
Function Results	4-30	Field Selection	5-31
Routine Declaration Part	4-31	Pointer Dereferencing	5-31
Routine Body	4-31	File Buffer Selection	5-31
Level-1 Routines	4-31	Operators	5-32
Directives	4-32	Arithmetic Operators	5-34
Forward	4-32	Boolean Operators	5-36
External	4-33	Set Operators	5-37
Recursive Routines	4-34	Relational Operators	5-39
Module Declaration	4-35	String Operator	5-41
Module Overview	4-36	Function References	5-41
Program Structuring	4-36	Constant Expressions	5-42
Encapsulation	4-36	Type Compatibility	5-44
Type-Safe Separate Compilation	4-36	Identical Types	5-44
Library Creation	4-46	Compatible Types	5-44
Definitions	4-37	Assignment Compatibility	5-45
Data	4-37	Special Cases	5-46
Services	4-37		
Module Structure	4-38		
Module Usage	4-43		
Where Modules May Be Declared	4-43	Chapter 6	Page
Where Modules May Be Imported	4-44	FILES	
How Module Interfaces are Found		Introduction	6-1
by the Compiler	4-45	Pascal File Types	6-1
Module Interface Search Pattern	4-46	Text Files	6-2
Module Access Rights	4-48	Sequential Files	6-4
Explicit Import Access Rights	4-49	Direct-Access Files	6-6
Implicit Import Access Rights	4-50	Associating Logical and Physical Files	6-8
Accessing EXTERNAL Routines from a Module	4-51	Opening Files	6-11
Dangling Pointer Access Rights	4-52	I/O Standard Procedures and Functions	6-14
Dangling Pointers	4-53	Append	6-15
Scope	4-55	Close	6-17
		Eof	6-18
Chapter 5	Page	Eoln	6-19
EXECUTABLE PARTS		Get	6-20
Statements	5-2	Lastpos	6-22
Statement Labels	5-4	Linepos	6-23
Assignment Statement	5-5	Maxpos	6-24
Procedure Statement	5-7	Open	6-25
Compound Statement	5-9	Overprint	6-27
IF Statement	5-10	Page	6-29
Case Statement	5-12	Position	6-30
WHILE Statement	5-15	Prompt	6-31
REPEAT Statement	5-16	Put	6-33
FOR Statement	5-17	Read	6-35
WITH Statement	5-20	Readdir	6-39
GOTO Statement	5-22	Readln	6-41
EMPTY Statement	5-23	Reset	6-43
Expressions	5-24	Rewrite	6-45
Operands	5-24	Seek	6-47
Literals	5-24	Write	6-49
Symbolic Constants	5-27	Writedir	6-56
Variables	5-28	Writeln	6-58

Table of Contents (Continued)

Chapter 7	Page	Chapter 8	Page
STANDARD PROCEDURES AND FUNCTIONS		IMPLEMENTATION CONSIDERATIONS	
Dynamic Allocation and De-Allocation Procedures	7-1	Data Allocation	8-1
Overview	7-1	Allocations of Scalar Variables	8-1
New	7-2	Allocations for Structured Variables	8-1
Dispose	7-3	Allocations for Elements of Packed Structures	8-4
Mark	7-4	Examples of Packed and Unpacked Structures	8-6
Release	7-4	Non-CDS Memory Configuration	8-12
String Procedures	7-5	Base Page	8-13
Setstrlen	7-5	Main Area	8-13
Strappend	7-6	Segment Overlay Area	8-15
Strdelete	7-7	Image Area	8-15
Strinsert	7-8	Heap/Stack	8-15
Strmove	7-9	CDS Memory Configuration	8-17
Strread	7-10	Code Partition	8-19
Strwrite	7-11	Base Page	8-19
String Functions	7-12	Global Data Area	8-19
Str	7-12	Non-CDS Routine Area	8-19
Strlen	7-13	Stack Area	8-20
Strltrim	7-14	Heap Area	8-20
Strmax	7-15	Image Area	8-20
Strpos	7-16	Heap/Stack	8-21
Strrpt	7-17	Data Management	8-22
Strrtrim	7-18	Non-CDS Management	8-23
Arithmetic Functions	7-19	CDS Stack Management	8-24
Abs	7-20	Heap Management	8-25
Arctan	7-20	Overview of Heap Organization	8-25
Cos	7-20	Heap Initialization	8-29
Exp	7-21	New	8-30
Ln	7-21	Dispose	8-33
Odd	7-21	Mark	8-37
Round	7-22	Release	8-39
Sin	7-22	EMA Heap Management \$HEAP 2\$	8-41
Sqr	7-22	CDS Heap Management	8-42
Sqrt	7-23	Short Versions of Heap Management	8-42
Trunc	7-23	Routines	8-43
Numeric Conversion Functions	7-24	Programs	8-45
Binary	7-25	Mixed Heap Programs	8-45
Hex	7-25	Modules	8-46
Octal	7-25	Relocating and Searching Modules	8-46
Ordinal Functions	7-26	Overlay Considerations	8-46
Chr	7-26	Mixed Heap Modules	8-47
Ord	7-27	Using Files Within Modules	8-48
Pred	7-28	Overlay Considerations	8-48
Succ	7-29	Creating Unique External Names	8-49
Packing and Unpacking Procedures	7-30	Strings	8-50
Pack	7-30	Passing Strings to Other Languages	8-50
Unpack	7-32	Using Strings of Unknown Length	8-50
Program Control Procedures	7-33	I/O	8-51
Halt	7-33	Program Heading Files	8-51

Table of Contents (Continued)

Standard Files	8-52	Loading a Simple Pascal Program	9-20
Automatic Association	8-52	Load-Time Errors	9-21
Direct-Access Files	8-53	Running a Pascal Program	9-22
Files in the Heap	8-53	Run-Time Errors	9-22
Relationship Between Logical Files and FMP Files	8-54	Debugging a Pascal Program	9-23
Sequential Files	8-54	Debug/1000	9-24
Direct-Access Files	8-54	Pascal/Debug Limitations	9-25
Text Files	8-56	Procedure Tracing	9-26
Interactive File I/O	8-56	Procedure Traceback	9-27
Closing Files with PURGE	8-57	Mixed Listing	9-28
Naming Restrictions	8-58	Interfacing Pascal Routines to Other Routines	9-29
Efficiency Considerations	8-59	Calling Non-Pascal Routines from Pascal	9-29
Data Access	8-59	Calling Pascal Routines from	
Accessing Variables and Parameters	8-59	Non-Pascal Routines	9-30
Passing Parameters	8-61	Calling FMP Routines from Pascal	9-31
Packed vs. Unpacked Data	8-64	Pascal and FORTRAN	9-32
Heap 1 vs. Heap 2 (Non-CDS)	8-68	Pascal and IMAGE	9-35
Heap 1 vs. Heap 2 (CDS)	8-69	EXEC Calls	9-37
Expressions	8-70	Encoding The Calls	9-37
Common Subexpressions	8-70	No-Abort Bit and Error Return	9-40
Numeric Data Types	8-70	ABREG Calls	9-42
Range Checking	8-71		
Sets	8-71		
Statements	8-72	Appendix A	Page
WITH Statement	8-72	ERROR MESSAGES AND WARNINGS	
FOR Statement	8-72	Program Errors	A-1
CASE Statement	8-73	I/O Errors and Warnings	A-5
Procedures and Functions	8-74	FMP Errors	A-9
Recursion	8-74	EMA Errors	A-10
Space Considerations	8-74	Segment Errors	A-10
Time Considerations (CDS)	8-75	Error Message Printers	A-10
Time Considerations (Non-CDS)	8-76	Catching Errors	A-10
Direct Calling Sequences (Non-CDS)	8-77	Compile-Time Warnings	A-12
FMP vs. Pascal I/O	8-78	Compile-Time Errors	A-12
Reducing the Size of a Loaded Program	8-79		
Short Versions of Library Routines	8-79	Appendix B	Page
Using Segmentation to Save Space	8-80	SYNTAX DIAGRAMS	B-1
Putting Globals in the Heap	8-80		
Structured Constants	8-81		
 Chapter 9	 Page		
HOW TO USE PASCAL/1000			
Relevant Files	9-1	Appendix C	Page
Compiling a Program	9-3	COMPILER OPTIONS	
Source File	9-4	ALIAS <string>	C-3
Listing File	9-5	ANSI <ON or OFF>	C-4
Assembly File	9-6	ASMB <string>	C-4
Relocatable File	9-7	AUTOPAGE <ON or OFF>	C-4
Option File and Runstring Compiler Options	9-8	BASIC_STRING <ON or OFF>	C-5
Use of Default File Names	9-10	BUFFERS <integer>	C-5
Alternate Compiler Designation	9-13	CDS <ON or OFF>	C-6
Compiler Workspace	9-14	CODE <ON or OFF>	C-6
The Listing	9-14	CODE_CONSTANTS <ON or OFF>	C-6
Loading a Pascal Program	9-17	CODE_INFO <ON or OFF>	C-7

Table of Contents (Continued)

FIXED_STRING <ON or OFF>	C-11	TITLE <string>	C-34
HEAP <integer>	C-12	TRACE <integer>	C-35
HEAP_DISPOSE <ON or OFF>	C-12	TRACE_BACK <ON or OFF>	C-36
HEAPPARMS <ON or OFF>	C-13	WARN <ON or OFF>	C-36
IDSIZE <integer>	C-16	WIDTH <integer>	C-36
IMAGE <integer>	C-16	WORK <integer>	C-37
INCLUDE <string>	C-16		
INCLUDE_DEPTH	C-17		
KEEPASMB	C-17		
LINES <integer>	C-17		
LINESIZE <integer>	C-18		
LIST <ON or OFF>	C-18		
LIST_CODE <ON or OFF>	C-18		
MIX <ON or OFF>	C-18		
NOABORT	C-19		
PAGE	C-20		
PARTIAL_EVAL <ON or OFF>	C-20		
PASCAL <string>	C-21		
PRIVATE_TYPES	C-22		
RANGE <ON or OFF>	C-22		
RECURSIVE <ON or OFF>	C-23		
RESULT <string>	C-23		
RUN_STRING <integer>	C-24		
SEARCH <string>	C-24		
SEGMENT	C-25		
SEGMENTED <ON or OFF>	C-26		
SKIP_TEXT <ON or OFF>	C-26		
SMALL_TEMPS <integer>	C-27		
STANDARD_LEVEL <'ANSI', 'HP', or 'HP1000'>	C-27		
STATS	C-28		
SUBPROGRAM	C-30		
SUBTITLE <string>	C-30		
TABLES <ON or OFF>	C-30		
Appendix D			
USER-CALLABLE PASCAL/1000			
LIBRARY ROUTINES			
Library Routines	D-1		
Pas.Parameters Function	D-2		
Pas.Sparameters Function	D-4		
Pas.NumericParms Procedure	D-5		
Pas.GetNetParms Procedure	D-6		
Pac.DcbAddress1 Procedure	D-7		
Pas.DcbAddress2 Procedure	D-8		
Pas.FileNamr Function	D-9		
Pas.InitMemInfo1	D-10		
Pas.GetMemInfo1 Procedure	D-11		
Pas.SetMemInfo1 Procedure	D-13		
Pas.InitialHeap1 Procedure	D-14		
Pas.InitMemInfo2 Procedure	D-15		
Pas.GetMemInfo2 Procedure	D-16		
Pas.SetMemInfo2 Procedure	D-17		
Pas.InitialHeap2 Procedure	D-18		
Pas.Coalesce1 and Pas.Coalesce2 Procedures	D-19		
Pas.TimeString Procedure	D-20		
Pas.SegmentLoad Procedure	D-21		
Pas.TraceBack Procedure	D-22		
Pas.StringData1 and Pas.StringData2 Functions	D-23		
Pas.StrEndCheck	D-24		
System Common Access Routines	D-26		
Shareable EMA Access	D-28		

List of Illustrations

Title	Page	Title	Page
Figure 1-1. Pascal/1000 Language Constructs ..	1-8	Figure 8-5. Heap/Stack Area After Initialization	8-29
Figure 3-1. Structure of a Sample Pascal/1000 Program	3-2	Figure 8-6. Allocation of a 3-word Variable ..	8-31
Figure 8-1. Pascal/1000 Non-CDS Memory Configuration	8-12	Figure 8-7. Allocation of a 4-word Variable ..	8-32
Figure 8-2. Main Area	8-14	Figure 8-8. Disposing of a 4-word Variable ..	8-34
Figure 8-3. Pascal/1000 CDS Memory Configuration	8-17	Figure 8-9. Allocation of a 1-word Variable ..	8-35
Figure 8-4. Heap Management Declarations and Routines	8-28	Figure 8-10. Creating a New Mark Region	8-38
		Figure 8-11. Releasing a Mark Region	8-40
		Figure 8-12. Definitions Used by Short Heap Management Routines	8-43

List of Tables

Title	Page	Title	Page
Table 1-1. HP Pascal Program Vocabulary	1-9	Table 8-4. Pascal/1000 Variable and Parameter Access	8-60
Table 2-1. Special Symbols	2-2	Table 8-5. Pascal/1000 Parameter Passing and Access	8-62
Table 2-3. Reserved Words	2-3	Table 8-6. Packed and Unpacked Data Access	8-65
Table 5-1. Pascal Operators	5-33	Table 8-7. Overhead Times for Routines with .ENTR vs. \$DIRECT\$ Calling Sequences (in microseconds, Non-CDS)	8-77
Table 8-1. Allocation for Scalar Variables	8-2		
Table 8-2. Allocation for Structured Variables	8-3		
Table 8-3. Allocations for Elements of Packed Structures	8-5		

Chapter 1

General Information

Introduction

Pascal is a high-level, block structured language. It is easy to learn, combining simplicity with a rich set of control structures and powerful data structures. Pascal programs are easy to develop due to compiler-time and runtime checking. Compared to other languages, Pascal programs are easy to read and maintain, as they tend to be self-documenting.

On the HP 1000 computer family, Pascal's usefulness is enhanced by the fact that it is compatible with other HP 1000 subsystems, including:

- Debug/1000 for symbolic debugging, tracing, and profiling
- Image/1000 for Data Base Management applications
- Graphics/1000-II for 2- and 3- dimensional graphics applications
- DS/1000-IV for networking applications

Also, routines written in Pascal can call, and be called by, routines written in other languages on the HP 1000, including:

- FTN7X (FORTRAN 77 language)
- MACRO/1000 (Assembly language)

Pascal Standards and Extensions

Pascal/1000 is an extension of HP Pascal, which in turn is an extension of ANSI Pascal. As noted in the preface, there are documents describing these standards that can be used for reference in addition to this manual.

This section summarizes the differences between the three versions of the Pascal languages. The first part describes the extensions HP Pascal has made to ANSI Pascal. The second part describes Pascal/1000 extensions of HP Pascal.

HP Pascal

Note

In ANSI Pascal, the term *string* refers to any PACKED ARRAY OF CHAR with a starting index of 1. Since HP Pascal defines the standard type STRING, the term PAC is used to refer to any PACKED ARRAY OF CHAR with a starting index of 1.

The following is a summary of HP Pascal extensions of ANSI Pascal:

Assignment Compatibility

ANSI Pascal defines assignment compatibility requirements. In HP Pascal, T2 is assignment compatible with T1 under the following additional conditions:

- If T1 is a PAC variable and T2 is a string literal or PAC expression, then T2 is assignment compatible with T1 provided that T2 is not longer than T1. If T2 is shorter than T1, then T2 is padded with blanks.
- If T1 and T2 are both real types (REAL or LONGREAL). If T1 is REAL and T2 is LONGREAL, then T2 is truncated before the assignment. If T1 is LONGREAL and T2 is REAL, then T2 is converted to LONGREAL before the assignment.
- If T1 and T2 are string types then T2 is assignment compatible with T1 provided that the length of the value of T2 is less than or equal to the maximum length of T1. The length of T1 is set to the length of T2.

Type Compatibility

ANSI Pascal defines type compatibility requirements. In HP Pascal, T1 and T2 are also type compatible if the following is true:

- T1 and T2 are PAC types with the same number of components, or if either T1 or T2 is a PAC expression whose length is less than the length of the other type. In this case, the shorter expression is extended on the right with blanks to reach a compatible length.
- T1 and T2 are string types.

CASE Statement

An OTHERWISE clause may be specified in the CASE statement to specify the actions to be performed should the case selector be a value not specified in any of the case constant lists.

Subranges of constant expressions may appear as case constants in the case constant lists.

Compiler Options (Directives)

Compiler options may be specified to control various aspects of the compilation and its output. HP Pascal defines five options: ANSI, PARTIAL_EVAL, LIST, PAGE, and INCLUDE.

Constant Expressions

A constant expression may appear in an HP Pascal program anywhere that a constant may appear in ANSI Pascal. A constant expression returns an ordinal value and may contain only declared constants, literals, calls to the functions ord, chr, pred, succ, hex, octal, binary, and the operators +, -, *, DIV, and MOD.

Constructors (Structured Constants)

Constants of structured types (records, arrays, strings, or sets) may be declared in the CONST section of a block. Set constructors may also appear in expressions in executable statements.

Declaration Part

The CONST, TYPE, and VAR sections of a declaration may be intermixed and repeated.

Halt Procedure

The halt procedure causes an abnormal termination of a program.

Identifiers

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

Function Return

A function may return a structured type (array, record, set, or string). File types, and structures containing file types, may not be returned.

Longreal Numbers

The type LONGREAL is identical with the type REAL except that it provides greater precision. The letter "L" or "l" precedes scale factor in a LONGREAL literal.

Minint

The standard constant minint is defined to be the smallest integer representable on the machine.

File I/O

Direct access I/O is supported, using the predefined routines open, seek, readdir, wriedir, maxpos, and lastpos.

The append procedure is provided to open a file for writing, positioned at the end of file.

The close procedure is provided to explicitly close any file.

To permit interactive input, the primitive file operation get is defined as "deferred get".

The procedure read accepts variables of enumerated types, PAC types, and string types.

The procedure write accepts expressions of enumerated types, and string types.

The function position returns the index of the current position for any file which is not a textfile.

The function linepos returns the integer number of characters which the program has read from or written to a textfile since the last line marker.

The procedure prompt flushes the output buffer of a textfile without writing a line marker.

The procedure overprint causes a line to be overprinted when a textfile is printed. A carriage return is performed without a line feed operation.

String Type

HP Pascal supports the predefined type STRING. A string is a packed array of CHAR with a declared maximum length and an actual length that may vary at runtime.

Several operators, procedures, and functions manipulate strings:

Assignment (:=) operator may be used to assign strings or string literals to strings.

Concatenation (+) produces a string composed of two other strings.

Relational (=, <>, <, <=, >, >=) operators may be used to compare two strings.

Strlen returns the current length of a string.

Strmax returns the maximum length of a string.

Strwrite writes one or more values to a string.

Strread reads values from a string.

Strpos returns the position of the first occurrence of a specified string within another string.

Strltrim and *strrtrim* trim leading and trailing blanks, respectively, from a string.

Strrpt returns a string composed of a designated string repeated a specified number of times.

Strappend appends one string to another.

Str returns a specified portion of a string, i.e., a substring.

Setstrlen sets the current length of a string without changing its contents.

Strmove copies a substring from a source string to a destination string.

Strinsert inserts one string into another.

Strdelete deletes a specified number of characters from a string.

WITH Statement

The record list in a WITH statement may include a call to a function which returns a record as its result.

Numeric Conversion Functions

The functions binary, octal, and hex convert a string literal or an expression of type string or PAC, to an integer.

EXTERNAL Routines

The keyword "EXTERNAL" can be used following a procedure or function heading to declare an external routine, i.e., one whose body does not appear in the current compilation unit.

Modules

HP Pascal defines a mechanism based on modules which:

- provides type-safe separate compilation of program fragments,
- provides a convenient method of structuring large programs,
- provides a means of encapsulating program objects.

A module defines objects (constants, types, variables, procedures, and functions). Only those objects "exported" by the module may be "imported" and used by a program or another module.

Record Variant Declaration

The variant part of a record field list may have subranges of constant expressions as case constants.

String Literals

HP Pascal permits the encoding of control characters or any other single ASCII character after the sharp symbol (#). For example, #G represents CNTL-G, and #0 represents NULL.

Pascal/1000

The following is a summary of Pascal/1000 extensions of HP Pascal:

Nested Comments

In ANSI and HP Pascal, the two symbols '{' and '(*' are equivalent. Likewise, '}' and '*)' are equivalent symbols. To provide compatibility with previous versions of the compiler, the above pairs of symbols are not equivalent when the setting of the STANDARD_LEVEL compiler option is 'hp1000'.

Thus, in HP Pascal, comments may not nest. In Pascal/1000, comments may nest one level by using opposite comment symbols, e.g. { ... (* ... *) ... }

Instead of relying on this feature, programmers are encouraged to use the SKIP_TEXT compiler option when commenting out code that contains comments.

REAL, LONGREAL constant expressions

In HP Pascal, constant expressions may not contain values of type REAL or LONGREAL. For compatibility with previous versions of the compiler, REAL and LONGREAL constant expressions are allowed if the STANDARD_LEVEL compiler option is set to 'hp1000'.

Separate Compilation

In HP Pascal, Modules provide the only form of separate compilation. For compatibility with earlier versions of the compiler, Pascal/1000 also provides the SUBPROGRAM and SEGMENT compiler options for separate compilation.

This facility is not type-safe (the compiler cannot check type compatibility across compilation units). Programs utilizing this feature are not portable, and are more prone to errors.

PAC types

In HP Pascal, a PAC is defined as a PACKED ARRAY [1..n] OF CHAR. For compatibility with previous versions of the compiler, Pascal/1000 also treats PACKED ARRAY [m..n] OF CHAR as a PAC. This is true only at STANDARD_LEVEL 'hp1000'. Use of this feature is discouraged. (Note: the Pascal/1000 extension does not apply to PAC parameters in the standard routine strmove, which must be HP Pascal PACs).

Parameters to reset, rewrite, open and append

In HP Pascal, the 2nd parameter to these routines must be specified. An all-blank PAC or string, or the null string, may be passed to specify that the default file name is to be used. For compatibility with previous versions of the compiler, Pascal/1000 will allow the omission of the 2nd parameter if the \$STANDARD_LEVEL\$ compiler option is set to 'hp1000', i.e., `reset (input,, 'CCTL')` is equivalent to `reset (input, '', 'CCTL')`. Use of this feature is discouraged.

Compiler Options

Pascal/1000 defines additional compiler options, described in Appendix C.

User-Callable Library Routines

Pascal/1000 provides routines in a runtime library that are callable from a user's program. These routines perform a variety of system-dependent functions, such as returning current information about the heap and stack, providing access to system common, returning the current date and time, etc.

Pascal Program Vocabulary

Table 1-1 gives a summary of HP Pascal language constructs, reserved words, keywords, and pre-defined identifiers which apply for both RTE-6 and RTE-A Operating Systems unless otherwise specified. It is meant to be a short, concise reference and to give the reader a flavor of the language. When determining exact syntax and semantics of a language construct, reference should be made to the appropriate chapter in the text, and/or Appendix B Syntax Diagrams.

Figure 1-1 is a quick guide to the language constructs and where they are described in this manual.

Items in Table 1-1 which are HP Pascal extensions of ANSI Pascal are flagged with a "*".

Items in Table 1-1 which are Pascal/1000 extensions of HP Pascal are flagged with a "+".

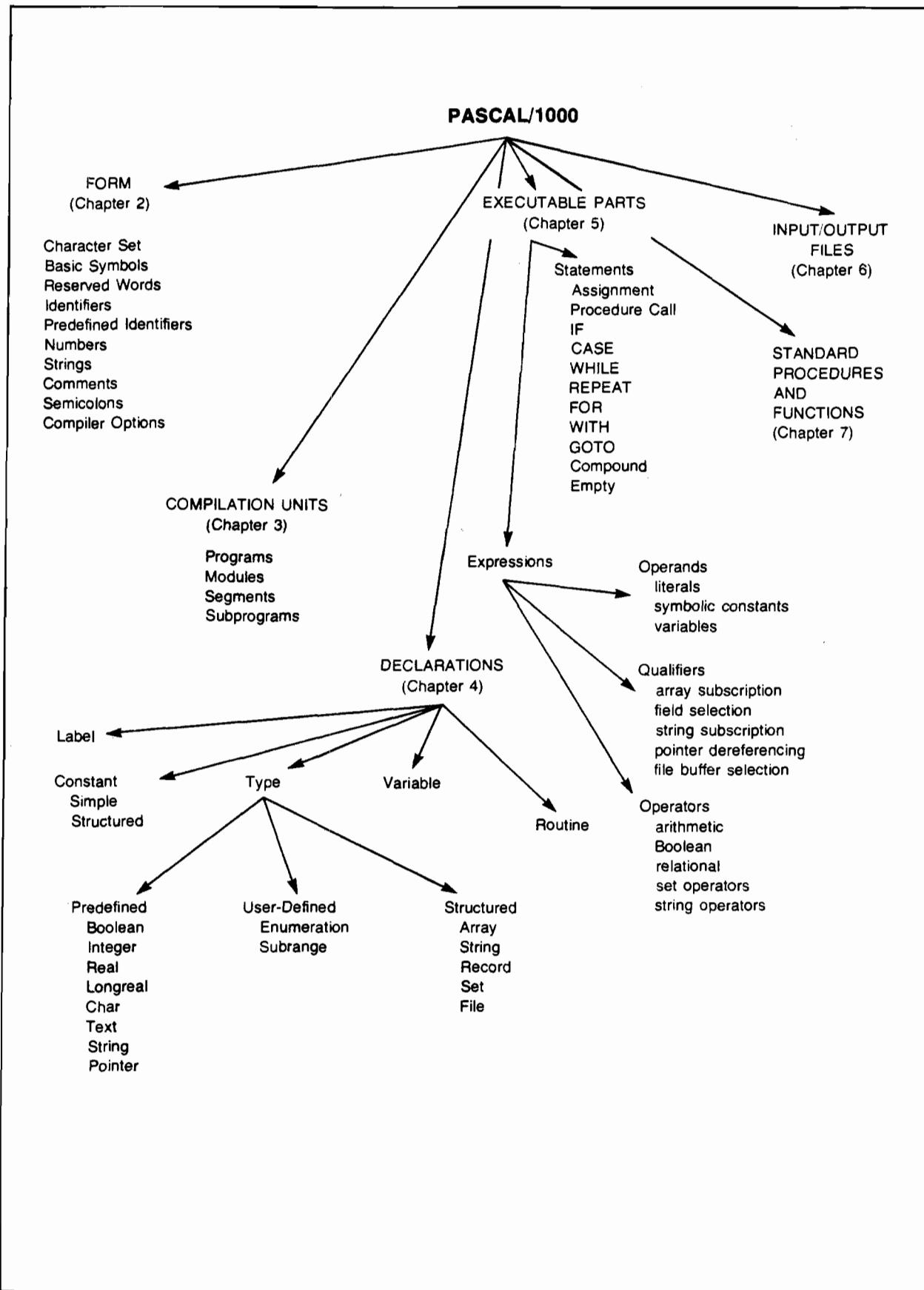


Figure 1-1. Pascal/1000 Language Constructs

Table 1-1. HP Pascal Program Vocabulary

<ul style="list-style-type: none"> ● Programs <pre>PROGRAM ProgramName (FileNames); Declarations RoutineDeclarations BEGIN Statements END.</pre> <ul style="list-style-type: none"> * ● Modules <ul style="list-style-type: none"> * MODULE ModuleName; <ul style="list-style-type: none"> * IMPORT <pre>ModuleNames;</pre> <ul style="list-style-type: none"> * EXPORT <pre>Declarations RoutineHeadings</pre> <ul style="list-style-type: none"> * IMPLEMENT <pre>Declarations RoutineDeclarations END;</pre> <ul style="list-style-type: none"> ● Procedures and Functions <pre>PROCEDURE ProcedureName (Parameters); Declarations RoutineDeclarations BEGIN Statements END; FUNCTION FunctionName (Parameters): ReturnType; Declarations RoutineDeclarations BEGIN Statements END;</pre> <ul style="list-style-type: none"> * PROCEDURE ProcedureName (Parameters); EXTERNAL; PROCEDURE ProcedureName (Parameters); FORWARD; 	
<p>* Identifies extension to ANSI Pascal</p>	

Table 1-1. Pascal Program Vocabulary (Continued)

- Declarations

```
LABEL
  LabelNumbers;

CONST
  ConstantName = Constant
  ConstantName = ConstantExpression
  ConstantName = StructuredConstant
```

```
TYPE
  TypeName = TypeDefinition
```

```
VAR
  VariableName : TypeDefinition
```

- Type Definitions

```
BOOLEAN
INTEGER
REAL
LONGREAL
CHAR
TEXT
```

```
TypeId
ConstantExpression..ConstantExpression
(EnumerationIds)
```

```
ARRAY [IndexTypes] OF ElementType
PACKED ARRAY [IndexTypes] OF ElementType
```

```
RECORD
  Fields
END
```

```
PACKED RECORD
  Fields
END
```

```
ATypeId
```

```
SET OF TypeDefinitions
PACKED SET OF TypeDefinition
```

* STRING [StringSize]

```
FILE OF TypeId
PACKED FILE OF TypeId
```

* Identifies extension to ANSI Pascal

Table 1-1. Pascal Program Vocabulary (Continued)

<ul style="list-style-type: none">• Statements<ul style="list-style-type: none">Variable := ExpressionProcedureName (Parameters)GOTO LabelNumberBEGIN<ul style="list-style-type: none">StatementsENDIF Condition THEN<ul style="list-style-type: none">StatementELSE<ul style="list-style-type: none">StatementCASE Expression OF<ul style="list-style-type: none">Constants: StatementConstants: Statement* OTHERWISE StatementsENDWHILE Condition DO<ul style="list-style-type: none">StatementREPEAT<ul style="list-style-type: none">StatementUNTIL ConditionFOR Variable := Start TO Finish DO<ul style="list-style-type: none">StatementFOR Variable := Start DOWNTO Finish DO<ul style="list-style-type: none">StatementWITH RecordExpression DO<ul style="list-style-type: none">Statement	
* Identifies extension to ANSI Pascal	

Table 1-1. Pascal Program Vocabulary (Continued)

- Expressions

Literal

ConstantName

VariableName

FunctionName (Parameters)

ArrayVariable [Subscripts]

* StringVariable [Subscript]

RecordVariable . FieldName

PointerVariable ^

FileVariable ^

* SetTypeName [SetElements]
[SetElements]

ArithmeticOperator x

x ArithmeticOperator y

BooleanOperator p

p BooleanOperator q

m RelationalOperator n

s SetOperator t

* a StringOperator b

- Arithmetic Operators

+x Identity

-x Negate

x + y Add

x - y Subtract

x * y Multiply

x / y REAL Divide

x DIV y INTEGER Divide

x MOD y Modulus

- Boolean Operators

NOT p Logical NOT

p AND q Logical AND

p OR q Logical OR

- Relational Operators

m = n Equal

m <> n Not equal

m < n Less than

m <= n Less than or equal

m > n Greater than

m >= n Greater than or equal

* Identifies extension to ANSI Pascal

Table 1-1. HP Pascal Program Vocabulary (Continued)

<ul style="list-style-type: none"> • Set Operators <table border="0"> <tr><td>s = t</td><td>Set equality</td></tr> <tr><td>s >= t</td><td>Superset</td></tr> <tr><td>s <= t</td><td>Subset</td></tr> <tr><td>e IN s</td><td>Set inclusion</td></tr> <tr><td>s + t</td><td>Union</td></tr> <tr><td>s - t</td><td>Difference</td></tr> <tr><td>s * t</td><td>Intersection</td></tr> </table> <ul style="list-style-type: none"> • String Operators <p>* a + b Concatenation</p> <ul style="list-style-type: none"> • Arithmetic Functions <table border="0"> <tr><td>abs (x)</td><td>Absolute value</td></tr> <tr><td>odd (x)</td><td>True if x is odd</td></tr> <tr><td>sqr (x)</td><td>x*x</td></tr> <tr><td>sin (x)</td><td>sine</td></tr> <tr><td>cos (x)</td><td>cosine</td></tr> <tr><td>exp (x)</td><td>e to the power of x</td></tr> <tr><td>ln (x)</td><td>natural log</td></tr> <tr><td>sqrt (x)</td><td>square root</td></tr> <tr><td>arctan (x)</td><td>arctangent</td></tr> <tr><td>trunc (x)</td><td>integer portion of real x</td></tr> <tr><td>round (x)</td><td>real x rounded to an integer</td></tr> </table> <ul style="list-style-type: none"> • Ordinal Functions <table border="0"> <tr><td>pred (x)</td><td>predecessor of x</td></tr> <tr><td>succ (x)</td><td>successor of x</td></tr> <tr><td>ord (x)</td><td>ordinal value of x</td></tr> <tr><td>chr (x)</td><td>character equivalent of x</td></tr> </table> <ul style="list-style-type: none"> • Numeric Conversion Functions <p>* hex (x) integer interpreted from hex digits 0-F in string x</p> <p>* octal (x) integer interpreted from octal digits 0-7 in string x</p> <p>* binary (x) integer interpreted from binary digits 0-1 in string x</p> <ul style="list-style-type: none"> • Dynamic Allocation Procedures <table border="0"> <tr><td>new (p, t1,...tn)</td><td>Allocate variable, point p to it</td></tr> <tr><td>dispose (p, t1,...tn)</td><td>De-allocate variable, set p to nil</td></tr> <tr><td>mark (p)</td><td>Save heap state in pointer p</td></tr> <tr><td>release (p)</td><td>Restore heap state per p</td></tr> </table> <ul style="list-style-type: none"> • Packing, Unpacking Procedures <table border="0"> <tr><td>pack (a, i, z)</td><td>Assign elements from unpacked array a into packed array z, starting at element a[i].</td></tr> <tr><td>unpack (z, a, i)</td><td>Assign elements from packed array z into unpacked array a, starting at element a[i].</td></tr> </table>	s = t	Set equality	s >= t	Superset	s <= t	Subset	e IN s	Set inclusion	s + t	Union	s - t	Difference	s * t	Intersection	abs (x)	Absolute value	odd (x)	True if x is odd	sqr (x)	x*x	sin (x)	sine	cos (x)	cosine	exp (x)	e to the power of x	ln (x)	natural log	sqrt (x)	square root	arctan (x)	arctangent	trunc (x)	integer portion of real x	round (x)	real x rounded to an integer	pred (x)	predecessor of x	succ (x)	successor of x	ord (x)	ordinal value of x	chr (x)	character equivalent of x	new (p, t1,...tn)	Allocate variable, point p to it	dispose (p, t1,...tn)	De-allocate variable, set p to nil	mark (p)	Save heap state in pointer p	release (p)	Restore heap state per p	pack (a, i, z)	Assign elements from unpacked array a into packed array z, starting at element a[i].	unpack (z, a, i)	Assign elements from packed array z into unpacked array a, starting at element a[i].	
s = t	Set equality																																																								
s >= t	Superset																																																								
s <= t	Subset																																																								
e IN s	Set inclusion																																																								
s + t	Union																																																								
s - t	Difference																																																								
s * t	Intersection																																																								
abs (x)	Absolute value																																																								
odd (x)	True if x is odd																																																								
sqr (x)	x*x																																																								
sin (x)	sine																																																								
cos (x)	cosine																																																								
exp (x)	e to the power of x																																																								
ln (x)	natural log																																																								
sqrt (x)	square root																																																								
arctan (x)	arctangent																																																								
trunc (x)	integer portion of real x																																																								
round (x)	real x rounded to an integer																																																								
pred (x)	predecessor of x																																																								
succ (x)	successor of x																																																								
ord (x)	ordinal value of x																																																								
chr (x)	character equivalent of x																																																								
new (p, t1,...tn)	Allocate variable, point p to it																																																								
dispose (p, t1,...tn)	De-allocate variable, set p to nil																																																								
mark (p)	Save heap state in pointer p																																																								
release (p)	Restore heap state per p																																																								
pack (a, i, z)	Assign elements from unpacked array a into packed array z, starting at element a[i].																																																								
unpack (z, a, i)	Assign elements from packed array z into unpacked array a, starting at element a[i].																																																								
<p>* Identifies extension to ANSI Pascal</p>																																																									

Table 1-1. Pascal Program Vocabulary (Continued)

- String Procedures and Functions
- * **str (s, pos, n)**
Return the string containing the n characters starting at s [pos]
- * **strpos (s1, s2)**
Return position of s2 in s1
- * **strlen (s)**
Return current length of s
- * **setstrlen (s, e)**
Set string length of s to e
- * **strmax (s)**
Return maximum length of s
- * **strmove (n, s1, pos1, s2, pos2)**
Move n characters from s1 [pos1] to s2 [pos2]
- * **strappend (s1, s2);**
Append s2 to s1
- * **strinsert (s1, s2, n)**
Insert s1 into s2 starting at s2 [n]
- * **strdelete (s, pos, n)**
Delete n characters starting at s [pos]
- * **strread (s, start, next, v1, ..., vn)**
Read variables from s
- * **strwrite (s, start, next, e1, ..., en)**
Write expressions to s
- * **strltrim (s)**
Trim leading blanks
- * **strrtrim (s)**
Trim trailing blanks
- * **str rpt (s, n)**
Return a string with s repeated n times

* Identifies extension to ANSI Pascal

Table 1-1. HP Pascal Program Vocabulary (Continued)

<ul style="list-style-type: none"> ● File Handling Procedures and Functions 	
reset (f)	Open files:
rewrite (f)	Open f at start of file for reading
	Open f at start of file for writing
*reset (f, n, o)	Open files with name n and option string o:
*rewrite (f, n, o)	Open f at start of file for reading
*open (f, n, o)	Open f at start of file for writing
*append (f, n, o)	Open f at start of file for direct access
	Open f past eof for writing
read (f, v1,...,vn)	Read variables from f
readln (f, v1,...,vn)	Read variables from f, position to next line
*readdir (f, pos, v1,...,vn)	Read variables from f at position pos
get (f)	Read next component of f
write (f, e1,...,en)	Write expressions to f
writeln (f, e1,...,en)	Write expressions to f, position to next line
*writedir (f, pos, e1,...,en)	Write expressions to f at position pos
put (f)	Write next component of f
*seek (f, pos)	Position f at element pos
page (f)	Page eject textfile f
*prompt (f)	Flush output buffer, suppress carriage return
*overprint (f)	Flush output buffer, suppress line feed
eoln (f)	True if f is at end of line
eof (f)	True if f is at end of file
*linepos (f)	Number of chars read or written on this line
*lastpos (f)	Highest position written to file f
*maxpos (f)	Highest possible position for file f
*close (f, o)	Close file f with option string o
<ul style="list-style-type: none"> ● Program Control Procedures 	
*halt(n)	Terminate program with status n
<ul style="list-style-type: none"> ● Compiler Options 	
(HP Pascal)	
* ANSI	Issue warnings for non-ANSI constructs.
* INCLUDE	Include source from another file for compilation.
* LIST	Generate compiler listing.
* PAGE	Page eject the compiler listing.
* PARTIAL_EVAL	Perform partial evaluation of Boolean expressions.
<p>* Identifies extension to ANSI Pascal</p>	

Table 1-1. HP Pascal Program Vocabulary (Continued)

(Pascal/1000)	
+ ALIAS	Specify the externally-accessible name of a Pascal object.
+ ASMB	Specify options to the assembler.
+ AUTOPAGE	Automatically page before each routine in the listing.
+ BUFFERS	Specify the number of buffers for a file.
+ CDS	Generate CDS instructions.
+ CODE	Enable code generation.
+ CODE_CONSTANTS	Place structured constants in code space.
+ CODE_OFFSETS	Print the code offset of each Pascal line in the listing.
+ CODE_INFO	Print size information about generated code.
+ DEBUG	Generate information needed by Debug/1000.
+ DIRECT	Use faster calling sequence for given procedure.
+ EMA	Specify EMA size and MSEG size.
+ EMA_VAR	Allocate selected global variables in EMA/VMA.
+ ERROREXIT	Specify an error return on an external routine.
+ FAST_REAL_OUT	Use faster (and less precise) output routines for reals.
+ FIXED_STRING	Convert Pascal string parameters to FTN7X strings.
+ HEAP	Choose the small or large heap model for the program.
+ HEAP_DISPOSE	Choose heap management algorithm.
+ HEAPPARMS	Specify 1- or 2-word VAR parameter addresses.
+ IDSIZE	Specify number of significant characters in identifiers.
+ IMAGE	Reserve buffer space for IMAGE/1000 programs.
+ INCLUDE_DEPTH	Specify the maximum depth of include file nesting.
+ KEEPASMB	Save the generated assembly file after compilation.
+ LINES	Specify the number of lines per page in the listing.
+ LINESIZE	Specify the maximum characters in a textfile line.
+ LIST_CODE	Print emitted instructions in the compiler listing.
+ MIX	Emit Pascal source as comments in assembly file.
+ NOABORT	Specify "no-abort" error return on an external routine.
+ PASCAL	Specify the header information in the relocatable file.
+ PRIVATE_TYPES	Define end of common globals for Debug/1000.
+ RANGE	Enable range checking of variables, subscripts, pointers.
+ RECURSIVE	Enable recursive invocation of a routine.
+ RESULTS	Specify a file to contain compilation errors and results.
+ RUN_STRING	Specify size of run string passed to a program.
+ SEARCH	Specify list of files to search for imported modules.
+ SEGMENT	Define the current compilation unit as a segment.
+ SEGMENTED	Program which uses modules as overlays.
+ SMALL_TEMPS	Specify number of words to reserve for small temporaries.
+ STANDARD_LEVEL	Choose standard level to which compilation unit adheres.
+ STATS	Print status of compiler options and configuration info.
+ SUBPROGRAM	Define the current compilation unit as a subprogram.
+ SUBTITLE	Print a subtitle under the title in the listing.
+ TABLES	Print symbol table information.
+ TRACE	Print a procedure-call history of the executing program.
+ TITLE	Print a title at the top of each page in the listing.
+ TRACE_BACK	Display procedure call chain at point of run-time error.
+ WARN	Print warning messages in the compiler listing.
+ WIDTH	Specify number of significant characters in source lines.
+ WORK	Specify compiler's memory-resident workspace size.

+ Identifies extension to HP Pascal

Table 1-1. HP Pascal Program Vocabulary (Continued)

<ul style="list-style-type: none"> • Pascal/1000 User-Callable Library Routines 	
+ Pas.Parameters	Get the nth (PAC) parameter passed to the program.
+ Pas.SParameters	Get the nth (string) parameter passed to the program.
+ Pas.NumericParms	Get the numeric (RMPAR) parameters passed to the program.
+ Pas.GetMemInfo1	Return current status of the HEAP 1 heap.
+ Pas.SetMemInfo1	Change current status of the HEAP 1 heap.
+ Pas.InitialHeap1	Reset the HEAP 1 to heap to its initial state.
+ Pas.GetMemInfo2	Return current status of the HEAP 2 heap.
+ Pas.SetMemInfo2	Change current status of the HEAP 2 heap.
+ Pas.InitialHeap2	Reset the HEAP 2 heap to its initial state.
+ Pas.Coalesce1	Coalesce the free space in the HEAP 1 heap.
+ Pas.Coalesce2	Coalesce the free space in the HEAP 2 heap.
+ Pas.SharedSize	Return the size of the sharable EMA partition. (RTE-6)
+ Pas.A1SharedSize	Return the size of the sharable EMA partition (RTE-A)
+ Pas.SetShared	Allocate a heap/stack area from shared EMA (RTE-6)
+ Pas.A1SetShared	Allocate a heap/stack area from shared EMA (RTE-A)
+ Pas.RealTimeSize	Return size of realtime common (RTE-6)
+ Pas.RealTimeCom1	Return 1-word pointer to realtime common (RTE-6)
+ Pas.RealTimeCom2	Return 2-word pointer to realtime common (RTE-6)
+ Pas.BackGrndSize	Return size of background common (RTE-6)
+ Pas.BackGrndCom1	Return 1-word pointer to background common (RTE-6)
+ Pas.BackGrndCom2	Return 2-word pointer to background common (RTE-6)
+ Pas.BlankSize	Return size of blank system common (RTE-A)
+ Pas.BlankCom1	Return 1-word pointer to blank system common (RTE-A)
+ Pas.BlankCom2	Return 2-word pointer to blank system common (RTE-A)
+ Pas.LabelSize	Return size of labelled system common (RTE-A)
+ Pas.LabelCom1	Return 1-word pointer to labelled system common (RTE-A)
+ Pas.LabelCom2	Return 2-word pointer to labelled system common (RTE-A)
+ Pas.DCBAddress1	Return 1-word pointer to DCB of a Pascal file.
+ Pas.DCBAddress2	Return 2-word pointer to DCB of a Pascal file.
+ Pas.FileNamr	Return external name of a Pascal file.
+ Pas.TimeString	Get the current time, date, and year.
+ Pas.SegmentLoad	Load a segment into the overlay area.
+ Pas.TraceBack	Display the current procedure call chain.
+ Pas.StrEndCheck	Disable string overflow/underflow checks at runtime.
+ Pas.StringData1	Return 1-word pointer to the char part of a string.
+ Pas.StringData2	Return 2-word pointer to the char part of a string.
+ Identifies extension to HP 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.

Character (or character groups) that have special meaning in Pascal/1000 are shown in Table 2-1.

Table 2-1. Special Symbols

SYMBOL	DESCRIPTION
+	Add.
-	Subtract/Negate.
*	Multiply.
/	Real divide.
=	Equality.
<	Less than.
<=	Less than or equal to.
<>	Subset of.
>=	Not equal.
>	Greater than or equal to.
()	Superset of.
[]	Indicates an expression group or a parameter list.
,	Set, structured constant, array index, and string index delimiters.
;	Argument, structured constant, and enumeration separator.
.	Statement separator or 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.

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.

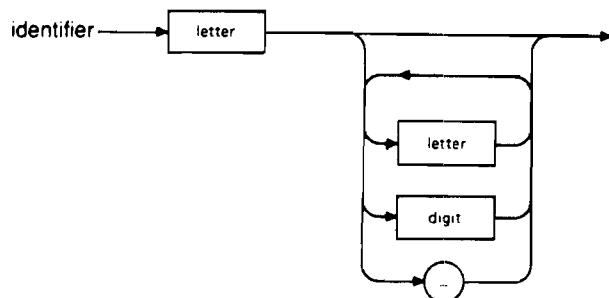
Table 2-2. Reserved Words

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.
EXPORT	Indicates module export section.
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.
IMPORT	Indicates module import section.
IMPLEMENT	Indicates module implement section.
IN	Set inclusion operator.
LABEL	Indicates label definition section.
MOD	Integer modulus operator.
MODULE	Indicates a module declaration.
NIL	Special pointer value.
NOT	Boolean negation operator.
OR	Boolean disjunction operator.
PACKED	Controls storage allocation for structured type.
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 scopes.

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
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 identifiers.

Each identifier must be unique within its scope (i.e., within the procedure, function, record, or program in which it is 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 for a more detailed description.)

Symbol	Type
false	BOOLEAN
true	BOOLEAN
maxint	INTEGER
minint	INTEGER

Predefined Types

(Refer to TYPE DEFINITIONS in Chapter 4 for a more detailed description.)

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).
STRING	Variable-length sequence of CHAR.

Predeclared Variables

(Refer to the detailed description of TEXT FILES in Chapter 6).

Symbol	Type
input	TEXT
output	TEXT

Predefined Procedures and Functions

(Refer to Chapter 7 for a more detailed description.)

abs	eoln	new	pred	round	strinsert	strwrite
arctan	exp	octal	prompt	seek	strlen	succ
append	get	odd	put	setstrlen	strltrim	trunc
binary	halt	open	read	sin	strmax	unpack
chr	hex	ord	readdir	sqr	strmove	write
close	lastpos	overprint	readln	sqrt	strpos	writedir
cos	ln	pack	release	str	strread	writeln
dispose	mark	page	reset	strappend	strrpt	
eof	maxpos	position	rewrite	strdelete	strrtrim	

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.

String Literals

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

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

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

Comments

A *comment* is a sequence of characters which start with either of the equivalent symbols '{' or '(' and ends with either of the equivalent symbols '}' or ')'. Comments may not be nested.

Comments do not have to be on lines by themselves and may cross line boundaries.

Example of legal comments

```
{A simple comment}
(* Another comment *)
{And another with mixed symbols*}

***** * This comment occupies 3 source lines *
*****)
```

Example of illegal comments

```
{An illegal {attempt to nest}
{This one doesn't work (* either *)}
```

At STANDARD_LEVEL 'HP1000', for compatibility with previous versions of the compiler, the symbol pairs above are not equivalent. Use of this feature is not portable, and is discouraged. The programmer is encouraged to use the SKIP_TEXT compiler option instead.

Example

```
$STANDARD_LEVEL 'HP1000'$ {Now a legal (* nested *) comment}
```

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 C for descriptions of the compiler options.

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.



Chapter 3

Compilation Units

Introduction

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

- Main program unit
- Module units
- Subprogram units
- Segment units

Every program must have a main program unit (unless a subprogram or module unit is intended to provide routines to be called from a program written in another language). 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 routines from subprogram and module units. Since subprogram, segment, and module units are optional, many Pascal/1000 programs will contain only a main program unit. Separately compiled program units can aid program development. If errors are found and corrected in a program unit, in general only that unit needs to be recompiled. The entire program is then reloaded.

The main program unit together with routines from subprogram and module units (along with any non-Pascal and library routines) constitute the main area. A segment unit together with routines from subprogram and module 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 and module units should be combined with the main program unit to form the main area, and which subprogram and module units are to be combined with each segment unit to form a segment overlay. The result is a disk-resident absolute module. Programs which operate in a CDS environment have a similar structure, but are automatically segmented at load time, and segments may reside in memory or on disk. Refer to Loading a Program in Chapter 9. Figure 3-1 shows the structure of a Non-CDS Pascal/1000 program which uses overlays.

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.

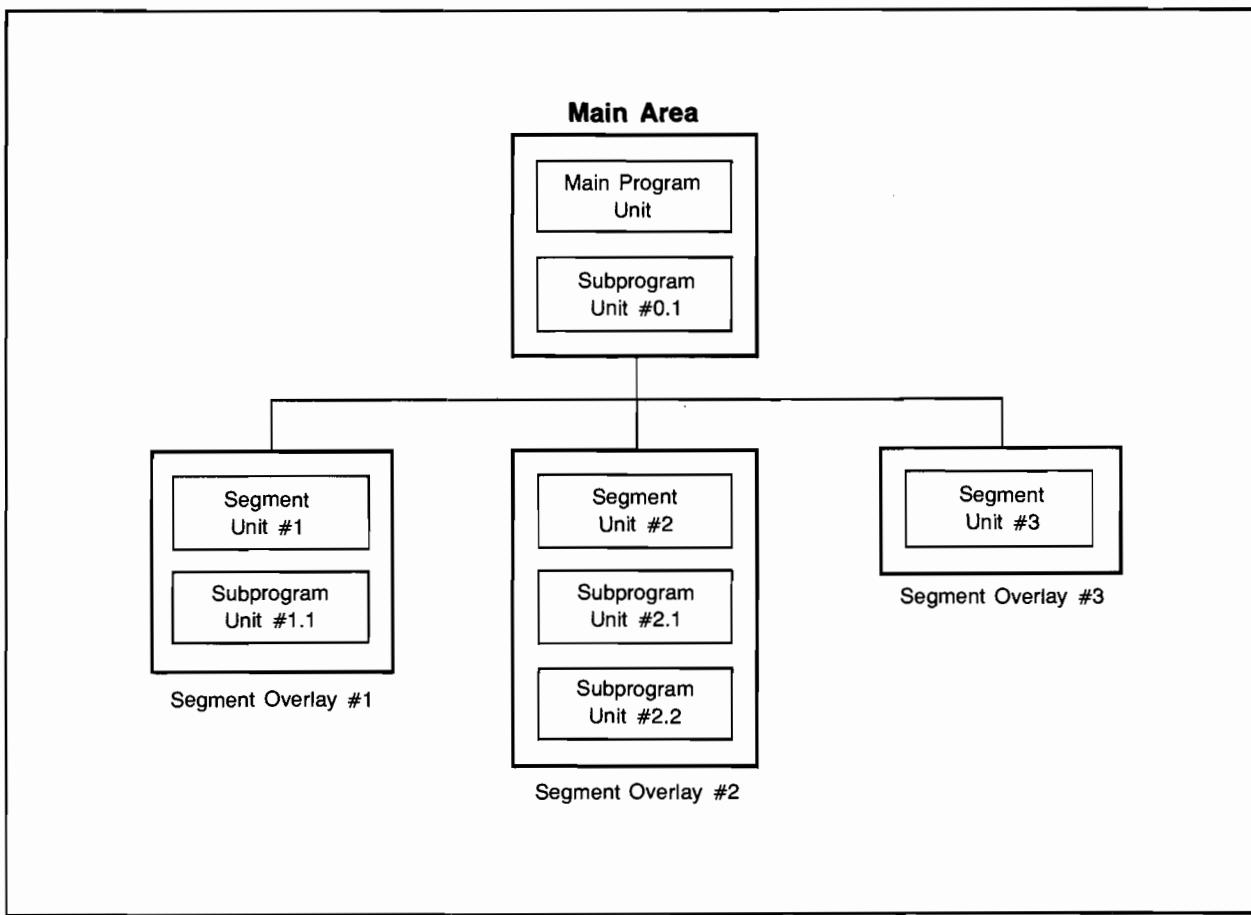


Figure 3-1. Structure of a Sample Pascal/1000 Program.

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 runtime, 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;

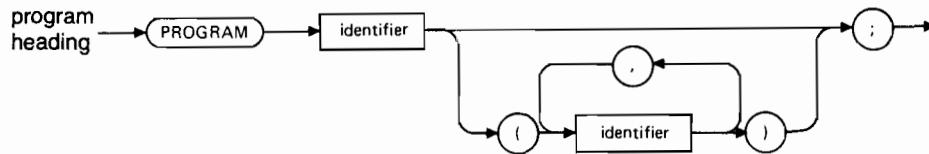
  BEGIN
    sum := x + y;
  END;

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 as well as the program. It follows the scoping rules of Pascal and is defined in its own scope between the predefined identifiers and the program globals. Therefore, it can redefine, and render inaccessible, a predefined identifier.

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 is 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 may consist of constants and types that are defined and labels, variables, and modules that are declared in (or imported into) 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.

A type-7 relocatable will be generated for each level-1 routine in the main program unit and a type-6 relocatable for the program body.

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             }

```

Module Unit

Modules provide a foundation for structuring programs that allows for type-safe separate compilation.

A module is a program entity which can be compiled independently and later used to complete otherwise incomplete programs. A module usually defines constants and data types, and declares both variables and routines which can operate on the data.

Selected constants, data types, variables and routines can be made available to the user of a module with an EXPORT declaration. A module or program can make use of the exported attributes of other modules with an IMPORT declaration.

Example of a simple module list unit

```

MODULE new;           {Module heading}
IMPORT              {Module block }
  old;
EXPORT
  PROCEDURE p (VAR t : INTEGER);
IMPLEMENT
  PROCEDURE p (VAR t : INTEGER):
    BEGIN
      t = old_var + 13
    END;
END.                 {Ending period}

```

A module unit consists of one or more module declarations, each separated from the next by a semicolon, and the last followed by a period. For module declaration details, refer to Chapter 4.

Type-7 relocatables are generated for the module unit as well as for each level-1 routine in the module.

Module Heading

The module heading associates an identifier with the module.

The module identifier is the name by which the module is known to the operating system and potential importers of the module.

Module Block

A module block consists of an IMPORT declaration, an EXPORT declaration and an IMPLEMENT declaration.

The import declaration lists modules whose attributes are to be visible (usable) in the current module. This section does not have to exist in a module block.

The export declaration defines/declares constants, types, and variables and lists headings of routines which are to be accessible to an importer of the module. The export section must exist and cannot be empty.

The implement declaration contains general declarations and routine declarations which constitute the functionality of the module. This section must exist, but may be empty.

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. A SUBPROGRAM unit cannot contain a module, although an import declaration may appear in the global declaration section of a SUBPROGRAM.

Type-7 relocatables are generated for the subprogram unit as well as for each level-1 routine in the subprogram unit.

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 has no significance within the program, or to the operating system, except that it must be unique within the first sixteen characters with respect to all other subprogram unit names, segment unit names, module names, and level-1 routine names, as well as with respect to the main program name. It also follows the scoping rules of Pascal and is defined in its own scope between the predefined identifiers and the program globals. Therefore, it can redefine, and render inaccessible, a predefined identifier.

The parameter identifiers are the names of the program parameters. These parameters, for each subprogram unit, must match those of the main program in name, order, number, and type.

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.

The standard files INPUT and OUTPUT can be accessed from any routine in a SUBPROGRAM (or SEGMENT) compilation unit, regardless of whether they are specified in the subprogram unit program heading (assuming INPUT/OUTPUT appear in the main program heading and have not been redefined).

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 normally accessible to other compilation units, via an external declaration, subject to the following restrictions:

1. For programs loaded using single-level segmentation, routines from SUBPROGRAM units which are loaded with the main program are accessible to:
 - a. the main program;
 - b. any other routines from SUBPROGRAM units loaded with the main program;
 - c. the currently active segment (if any);
 - d. any other routines from SUBPROGRAM units loaded with the currently active segment (if any).
2. Routines from a SUBPROGRAM unit which are loaded with a segment are accessible only when the segment is currently active. They are then accessible to the same list of units (1-a through 1-d).

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 (as there is in a main program).

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. There are three types of segmentation available; single-level segmentation (RTE-6/VM and RTE-A), multi-level segmentation (RTE-6/VM), and CDS segmentation (RTE-A). Information about segmentation in this manual refers to single-level segmentation. For information about using multi-level or CDS segmentation, refer to the appropriate loader/linker reference manual. The main differences between single-level segmentation and either of the other two methods are:

1. Using multi-level or CDS segmentation, segments can reside in memory as well as on disc.
2. Using multi-level or CDS segmentation, requests for segment overlays in the program are not needed.

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 runtime 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. A SEGMENT cannot contain a module, although an import declaration may appear in the global declaration part of a SEGMENT.

Segment Unit Program Heading

A segment program heading is the same as a subprogram program heading. The segment identifier is the name 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 of the program, and any subprogram, module or level-1 routine with a name of five or fewer characters.

A type-5 relocatable is generated for the segment unit and a type-7 relocatable for each level-1 routine in the segment.

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 for a more detailed explanation.

Example

```
$SEGMENT$                                {Compiler option}

PROGRAM Extrema;                         {Segment unit program heading}

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

VAR
  x : INT;                             {Global variable declaration}

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 (as there is in a main program).

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 Pas.SegmentLoad. Pas.SegmentLoad loads the segment overlay, then control returns to the statement following the call to Pas.SegmentLoad. No level-1 routine contained in the segment overlay is invoked by procedure Pas.SegmentLoad. Once the segment overlay has been loaded at runtime, 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 Pas.SegmentLoad 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 Pas.SegmentLoad.

```

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

PROCEDURE load_segment
  $ALIAS 'Pas.SegmentLoad'
  (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 Pas.SegmentLoad to load another overlay. Pas.SegmentLoad should be called only from the main program unit and its subprogram units.

NOTE

Overlays exist in non-CDS programs only. To allow the same source to be used in both CDS and non-CDS versions of a particular program, a Pas.SegmentLoad routine is provided in both the CDS and non-CDS Pascal libraries. The CDS version does nothing.

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, functions, and modules.

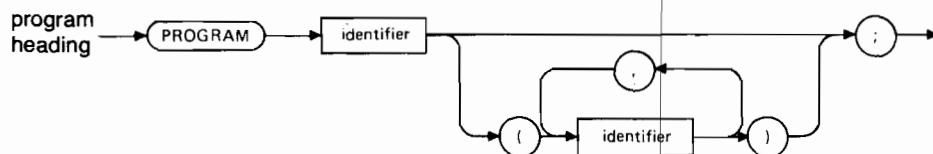
Some identifiers are predefined in Pascal/1000 (see Chapter 2), although they may be redefined by the user. All identifiers that are not predefined must be defined before they are used. There are two exceptions to this rule: program parameters (see Chapter 3) and pointer types (see below). Labels (which 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 identifier is the name by which the program is known to the operating system as well as the program. It follows the scoping rules of Pascal and is defined in its own scope between the predefined identifiers and the program globals. Therefore, it can redefine, and render inaccessible, a predefined identifier.

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 is 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);

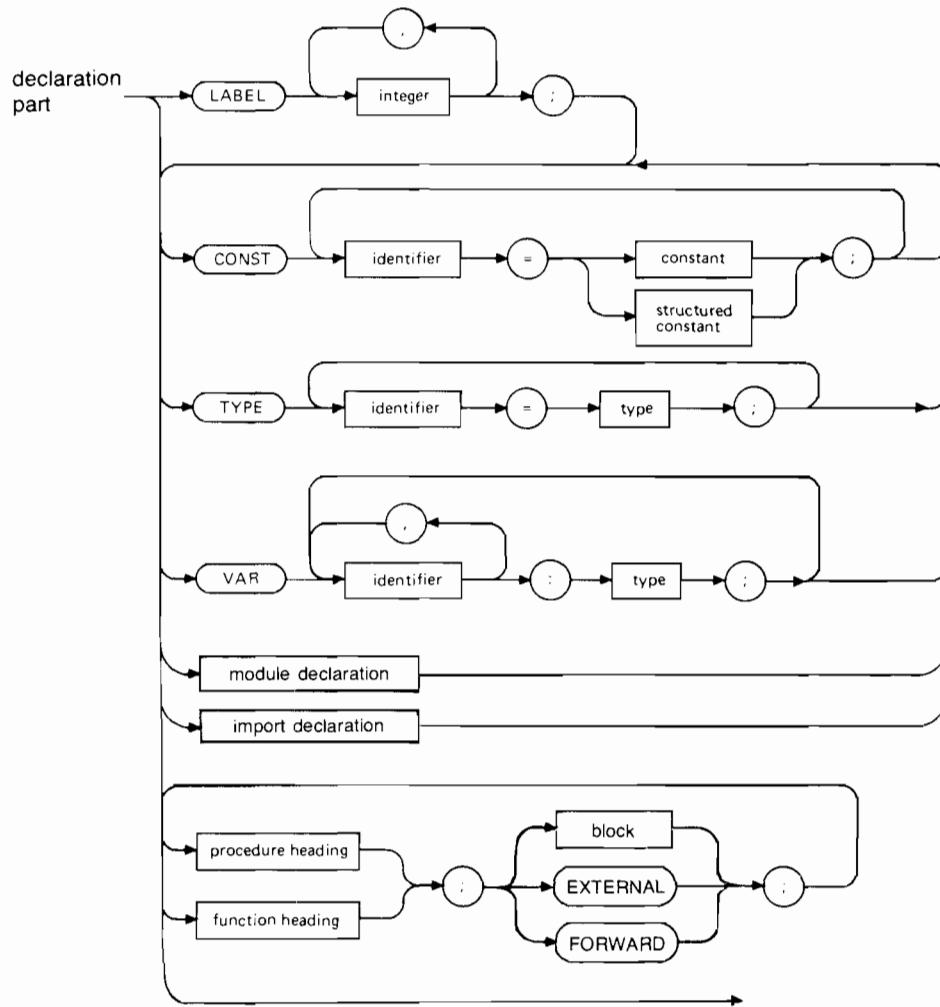
PROGRAM lister (data, output);

PROGRAM file_merge (master_file, updates_file, new_master);

```

Declaration Part

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

Syntax

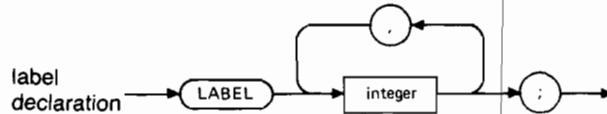
The LABEL declaration (if any) must come first. CONST, TYPE, and VAR sections may follow, in any order, and may be repeated as often as required. In the program declaration section, MODULE and IMPORT declarations may also appear, in any order, and may be repeated as often as required.

PROCEDURE and FUNCTION declarations may follow and may be repeated as often as required.

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 0000000006 are identical.

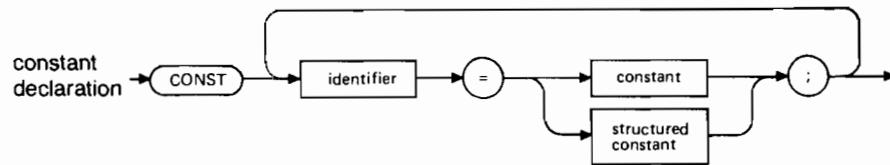
Example:

```
LABEL  
0, 0000000006, 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, subrange, or enumerated type) or a string literal. 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';
  
```

HP Pascal does not allow constant expressions to contain REAL or LONGREAL constants or literals. However, Pascal/1000 does allow them at STANDARD_LEVEL 'HP1000'. Mixing reals and integers is never allowed in constant expressions.

Examples

```

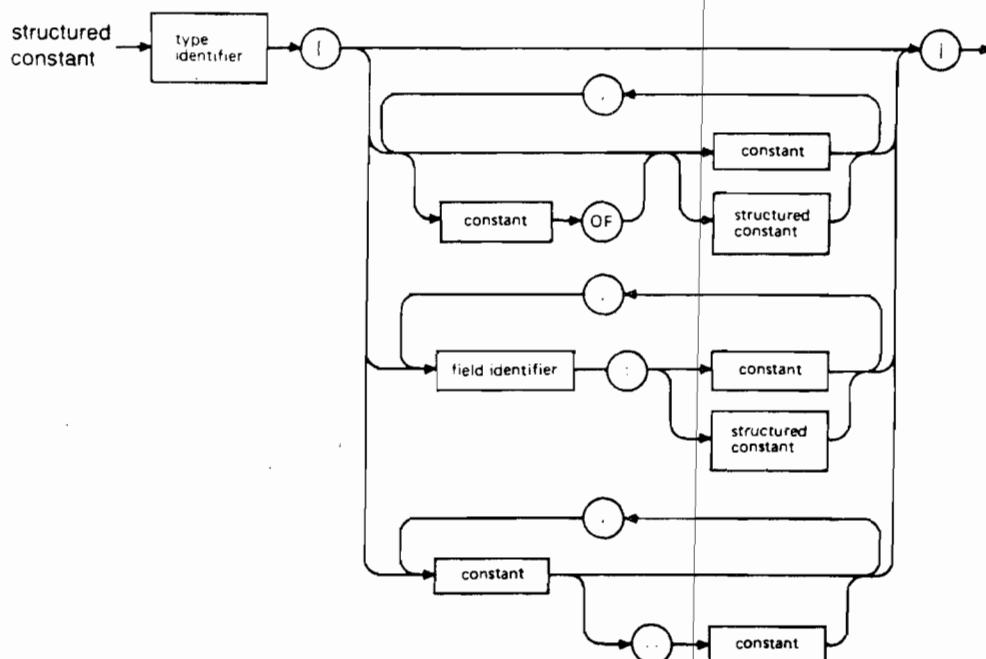
CONST
  {Correct}
  $STANDARD_LEVEL 'HP1000'$ pi = 3.0 + 0.1415926;

  {Incorrect}
  $STANDARD_LEVEL 'HP'$ pi = 3.0 + 0.1415926; {Not HP Standard}
  $STANDARD_LEVEL 'HP1000'$ pi = 3 + 0.1415926; {No mixing}
  
```

Structured Constants

A structured constant is a constant of a structured type (e.g., SET, STRING, RECORD, or ARRAY).

Syntax



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 (with the exception of PAC and STRING constants) 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 can also be used as constants but may not appear in the definition of other constants (structured or otherwise). A previously declared structured constant can be used in its entirety in the declaration of other structured constants.

Example

```

TYPE
  A5 = ARRAY [1..5] OF INTEGER;
  RT = RECORD
    a: BOOLEAN;
    b: A5;
  END;

CONST
  first  = A5 [1, 3, 5, 7, 9];
  second = RT [a: true, b: first];
  third  = RT [a: true, b: A5 [1, 3, 5, 7, 9]];
  
```

The constant declaration for second shows the use of a structured constant within the declaration of another structured constant. Second and third have the same contents.

Declarations

The reserved word OF is used in structured constants to indicate that a value is to appear in the constant several times. In the following example, initial_vector is a constant array of five real values (1.0,0.0,0.0,0.0,-1.0).

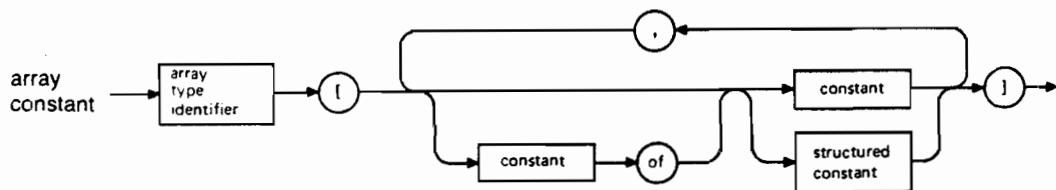
Example

```
TYPE
  vector = ARRAY [1..5] OF real;
CONST
  initial_vector = vector [1.0,3 OF 0.0,-1.0];
```

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;

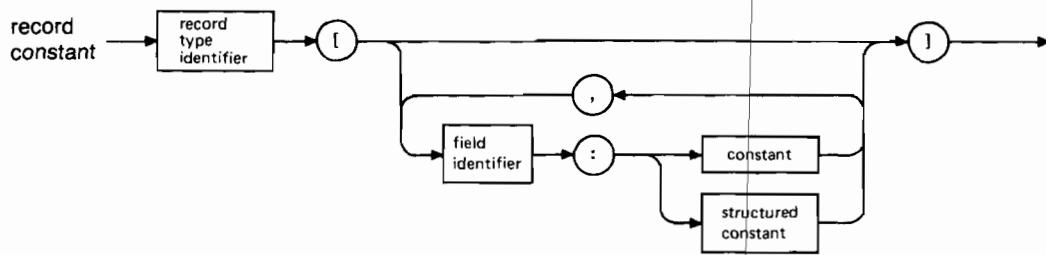
CONST
  true_values    = BOOLEAN_TABLE [true, true, true, true, true];
  init_values1   = TABLE [100 OF 0];
  init_values2   = TABLE [60 OF 0, 40 OF 1];
  identity       = MATRIX [ ROW [1, 0, 0, 0, 0],
                           ROW [0, 1, 0, 0, 0],
                           ROW [0, 0, 1, 0, 0],
                           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'        ]];
```

- Notice that in the last example, where the array was a PAC, that a combination of string literals and characters can be used. This is the only case where the constant (string literals) is permitted to be of a type different than the element type (CHAR). In addition not all of the character elements need to be specified. Blanks will be used to fill out any incomplete specifications (i.e. after 'BLUE' in the last example).

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 preceded 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;

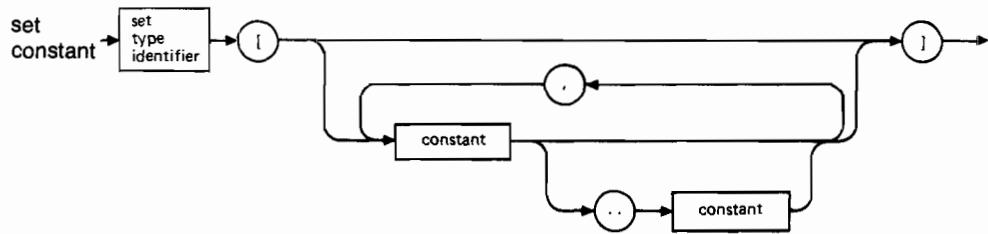
CONST
  no_count = COUNTER_RECORD [pages: 0, characters: 0, lines: 0];
  big_report = REPORT_RECORD
    [revision: 'C',
    price: 27.50,
    info: COUNTER_RECORD
      [pages: 6, lines: 28, characters: 496],
    secret: true,
    code: 8128];
  
```

Declarations

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

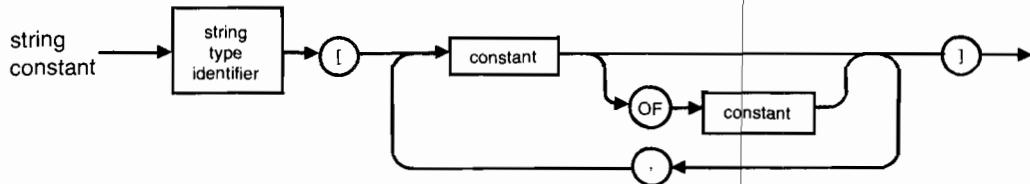
```
TYPE
  DIGITS = SET OF 0..9;
  CHARSET = SET OF CHAR;

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

String Constant

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

Syntax



Examples

```

TYPE
  STR_10 = STRING[10];
  SALUTATIONS = ARRAY [BOOLEAN] OF STR_10;
CONST
  blank = '';
  greeting = STR_10['Hello!'];
  farewell = STR_10['G', 2 OF 'o', 'd', 'bye'];
  aloha = SALUTATIONS
    [STR_10 ['Hello'],
     STR_10 ['Goodbye']];
  
```

As in a constant of a PAC type, a combination of string literals and characters can be used. The string contents need not be completely specified; the constant will have an appropriate "current length."

Type Definition

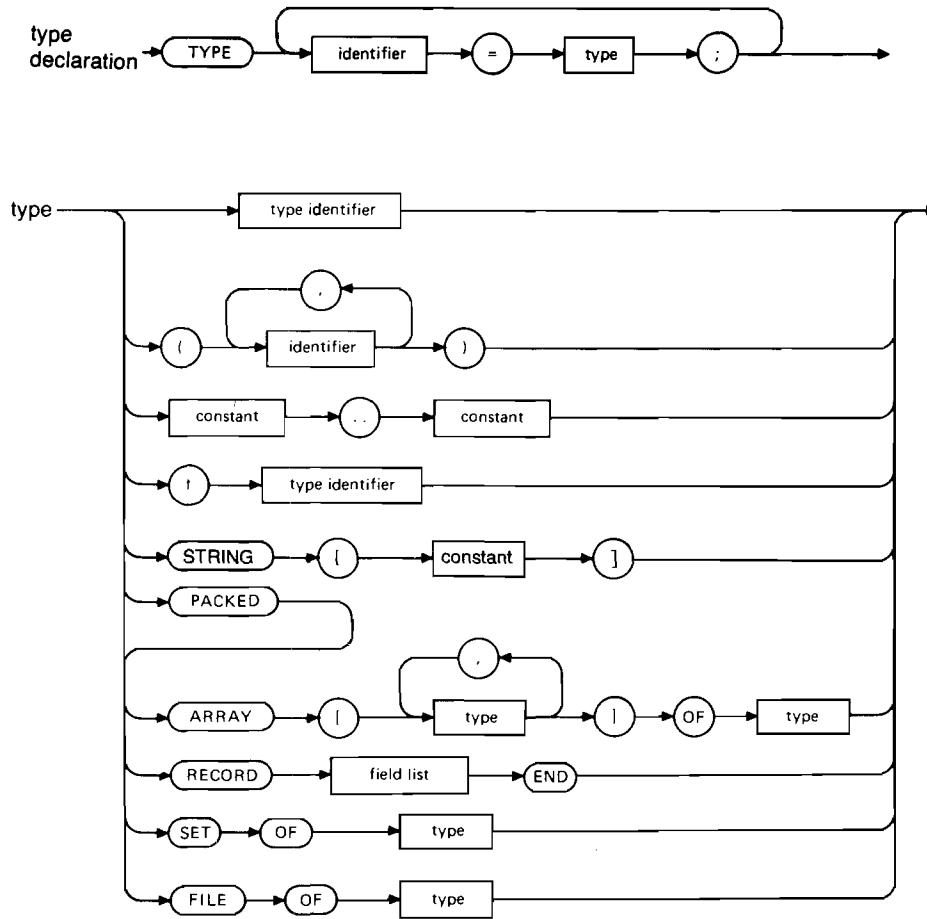
Every literal, constant, variable, function, and expression is of one and only one type. The type defines a set of attributes:

- 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



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 (char, 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. Arithmetic operators (+, -, *, /, DIV, MOD)
- d. Predefined functions (abs, arctan, chr, cos, exp, linepos, ln, maxpos, odd, ord, position, pred, round, sin, sqr, sqrt, succ, trunc)

Declarations

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:

-1.70141183460469231L+38 to -1.46936793852785946L-39

0.0

1.46936793852785938L-39 to 1.70141183460469227L+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)

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
- e. subrange of INTEGER
- f. PAC
- g. STRING
- h. enumerated (including BOOLEAN)

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
- e. subrange of INTEGER
- f. PAC
- g. STRING
- h. enumerated (including BOOLEAN)

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

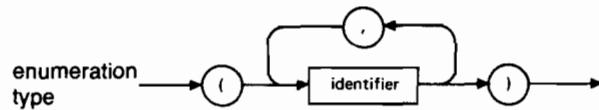
Further information on the special attributes of text files will be found in Chapter 6.

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

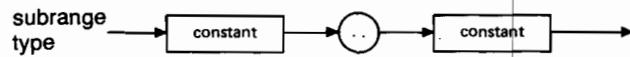
```

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 type 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

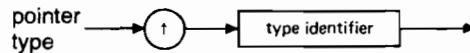
Pointer

There are two types of variables in Pascal: 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 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.

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. 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;
  
```

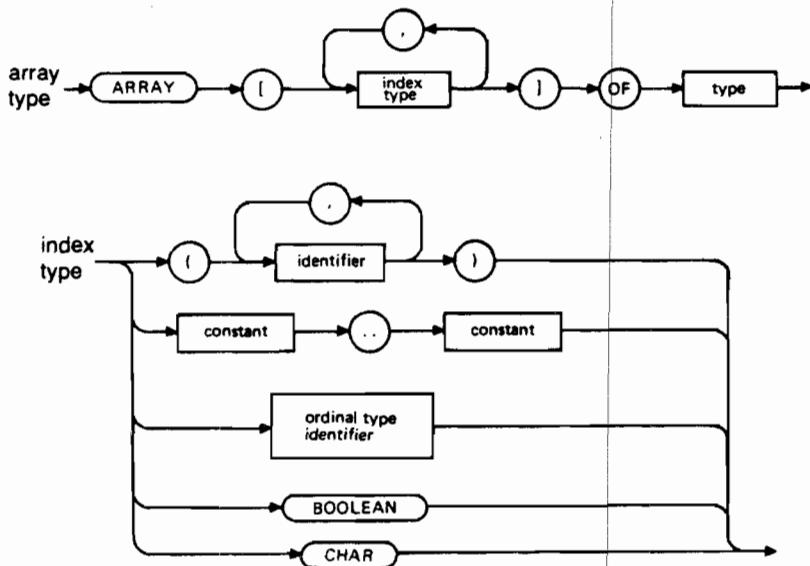
In Pascal/1000, the size of a pointer is either one or two 16-bit words, depending on the setting of the HEAP compiler option.

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



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

Elements can be of any type, including FILE, STRING, RECORD, and ARRAY and the total number of array elements must be less than or equal to maxint.

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

$$\text{number of words} = (\text{words per element}) * (\text{number of elements})$$

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)

Declarations

Examples

```
TYPE
  ASTRING = STRING[80]
  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;
  PAGE   = ARRAY [1..23] of ASTRING;
```

PACs — In ANSI Pascal, the term 'string' designates a packed array of CHAR with a starting index of 1. HP Pascal defines a standard type STRING which is identical with a packed array of CHAR except that its length may vary at runtime. To distinguish these two data types, the acronym PAC will denote

PACKED ARRAY [1..n] OF CHAR

throughout this manual.

Assignment of PACs requires that the target PAC be the same length or longer than the source PAC; the target will be blank padded if necessary. Comparison will cause the shorter PAC to be blank padded to the length of the longer PAC before the comparison occurs. Passing a PAC expression by value to a routine with a longer PAC formal parameter will cause the expression to be blank-padded before the call is made.

NOTE

For compatibility with previous versions of the compiler, Pascal/1000 also treats PACKED ARRAY [m..n] OF CHAR as a PAC, where m is any integer. This is true only at STANDARD_LEVEL 'hp1000', and does not apply to the parameters in strmove which may be PACs. Use of this feature is discouraged.

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

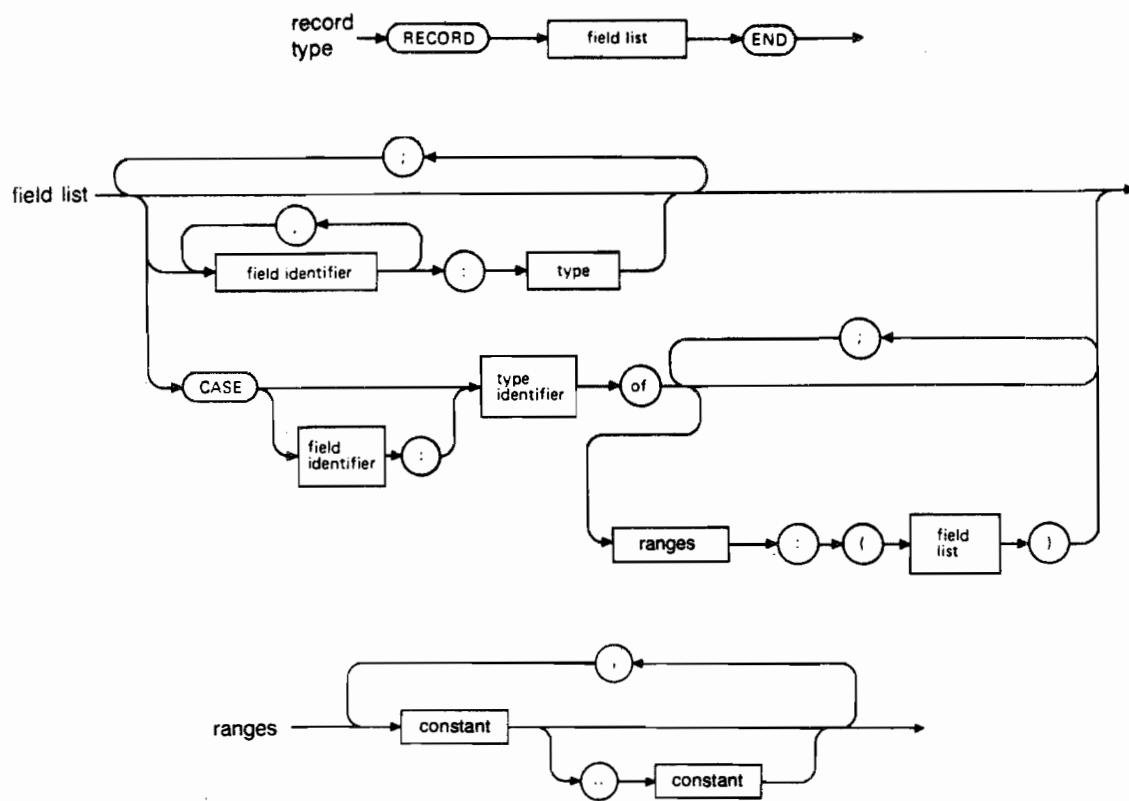
Array "rows" and "planes" can be assigned to identical "rows" and "planes", and passed as parameters to identical "rows" and "planes". This can be generalized to any number of dimensions.

Declarations

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. (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 allowed. 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:

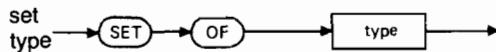
```
TYPE
  NAME_STRING     = PACKED ARRAY [1..30] OF CHAR;
  DATE_INFO       = PACKED RECORD
    mo: (jan, feb, mar, apr, may, jun,
          jul, aug, sep, oct, nov, dec);
    da: 1..31;
    yr: 1900..2100;
  END;
  MARITAL_STATUS = (married, single, divorced);
  PERSON_INFO     = RECORD
    name: NAME_STRING;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

Declarations

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)

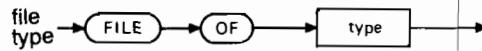
Examples

```
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

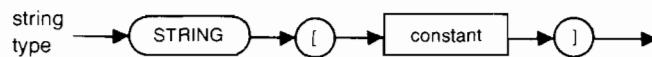
```

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;
  
```

String

A STRING type is a packed structure consisting of a variable-length sequence of components of the standard type char. Single-character components may be accessed.

Syntax



The symbol "string" is a predefined identifier, not a reserved word. The constant expression represents the maximum length, and must have a positive integer value within the range 1 .. 32767.

The current length of a value of a STRING type is the number of characters contained in the value, and may vary from zero to the maximum length of the type. A string may not be accessed outside of the components defined by its current length.

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

- a. Assignment operator (:=)
- b. Concatenation operator (+)
- c. Relational operator (<, <=, =, >>, >=, >)
- d. Predefined procedures and functions (strlen, strmax, strpos, strltrim, strrtrim, strrpt, str, strappend, strdelete, strinsert, strmove, strread, strwrite).

Examples

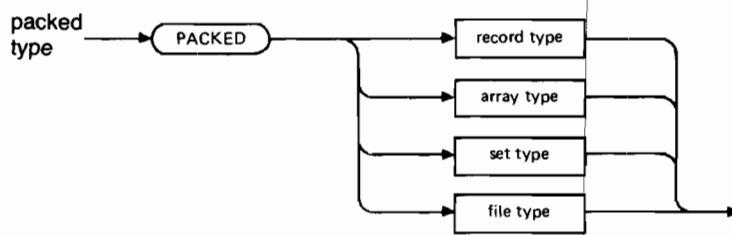
```
TYPE
  LINE      = STRING[80];
  BUFFER    = STRING[80*2];
  PAGE      = ARRAY [1..22] OF LINE;
  PERSON    = RECORD
              name: STRING[30];
              age: 1..100;
            END;
```

Packed Type Modifier

The representation of a variable in Pascal/1000 is usually determined by the compiler. Ease and speed 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, except STRING, 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. However, the PACKED modifier does distribute across multiple array dimensions. For example, these are equivalent:

```

x = PACKED ARRAY [1..10,1..10] OF BOOLEAN;
x = PACKED ARRAY [1..10] OF PACKED ARRAY [1..10] OF BOOLEAN;
  
```

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).

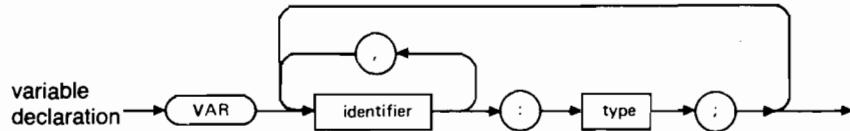
The standard procedures PACK and UNPACK can be used to assign components from an unpacked array to a packed array, and vice-versa (See Chapter 7).

In Pascal/1000, the PACKED type modifier has no effect on FILE or SET data types.

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^* , may be used to access the components of the file F.

Examples

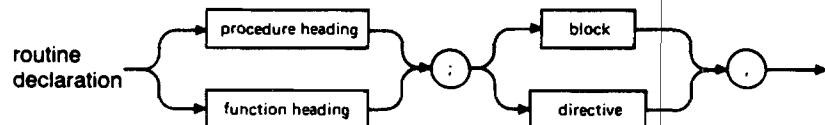
```

VAR
  { predefined types }
  pageCount,
  lineCount,
  charCount:           INTEGER;
  currentGrade:        CHAR;
  average:             REAL;
  standardDeviation:  LONGREAL;
  currentLine:          STRING[80];
  debugging, done:      BOOLEAN;
  terminalFile:         TEXT;
  someFiles:            ARRAY [0..4] OF TEXT;
  manyNumbers:          ARRAY [-1000..1000] OF INTEGER;
  manyChars:            PACKED ARRAY [1..10000] OF CHAR;
  manyTruths:           ARRAY [0..1999] OF BOOLEAN;
  { user-defined types }
  today, tomorrow:      DAYS;
  bewareOf:              FOREST_ANIMALS;
  ordinalDate:           DAY_OF_YEAR;
  firstRec1:              PTR1;
  friends:                PEOPLE;
  shape:                  POLYGON;
  tempShape:              ^POLYGON;
  weHave:                 FRUITSET;
  numberRight:             0..100;
  personnelFile:          PERSON_FILE;
  haveSeen:                SET OF FOREST_ANIMALS;
  savedShapes:             FILE OF POLYGON;
  storesHave:              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

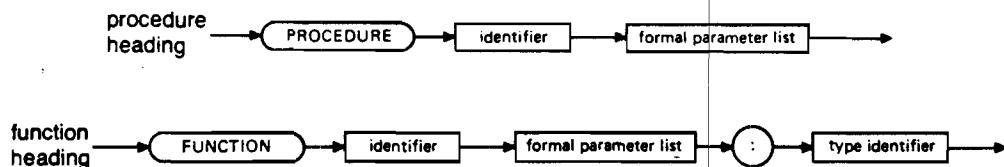


The routine heading specifies the identifier to be associated with the routine, any parameters to the routine, and the type of the result if the routine is a function. The routine block contains a declaration part which specifies the labels, constants, types, variables, and routines which are local to 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

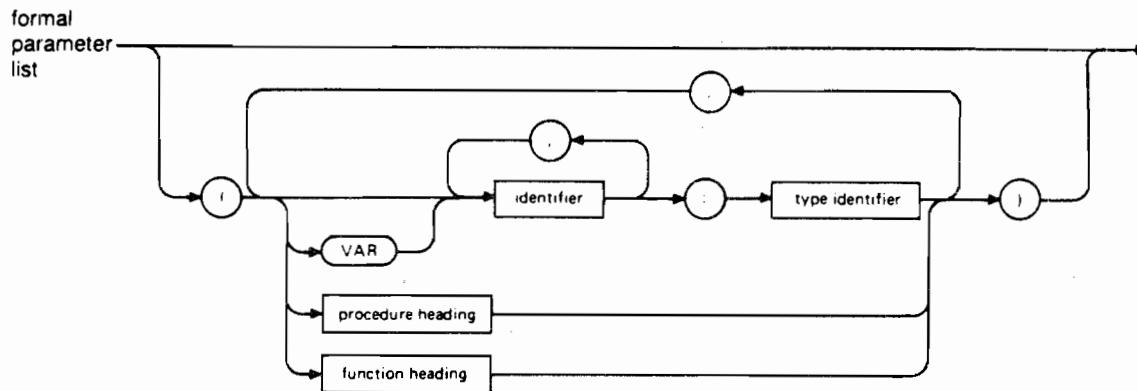


The identifier following the reserved word PROCEDURE or FUNCTION is the name by which the routine is known in the source code. Certain restrictions apply to level-1 routine names. These are discussed in the section Level-1 Routines.

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 the 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 if x,y,z is 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 "PAC" is compatible with an actual parameter of type "PAC" 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 Pascal/1000 programs using HEAP 2, the size of an actual value parameter in VMA/EMA cannot be more than 1024 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 by 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.

As a special case, a variable parameter may be of type STRING, without any maximum length given. Such parameters assume the type of their corresponding actual parameter. Any STRING variable can be passed to such a parameter. The function

```
FUNCTION index_of_last_non_blank
  (VAR string_2_check: STRING) {Note the missing max length.}
    : INTEGER;
```

will accept any string variable as an argument.

Value parameters may not be of type STRING without a maximum length given (they must be of a specific string type).

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 check1 (x, y, z: REAL): BOOLEAN;
BEGIN
  {perform some type of validity check on x, y, z
   and return appropriate value}
END;

FUNCTION check2 (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 (check1)
    ELSE read_data (check2);
  ...
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 (or the formal parameter is of type STRING and the actual parameter is any STRING variable).
- 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.

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. The operating system requires that level-1 routines must have names which are unique within the first sixteen characters. The program name must also be unique within the first sixteen characters with respect to any level-1 routine.

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 (except within the EXPORT section of a module where the location of the block is implied. See Module Declaration.)

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, the parameter list (and result type in the case of functions) may be respecified. The parameter lists must match in terms of VAR/value, identifier names and types (parameters of the form (a,b:CHAR) are considered to match (a:CHAR; b:CHAR)).

Example

```

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

PROCEDURE error (code : INTEGER);
  FORWARD;

.

.

.

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

PROCEDURE error (code : INTEGER);
BEGIN
  writeln ('Error', code, 'occurred');
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 another compatible language. 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 pre-defined 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.

- a. In a non-CDS environment if the RECURSIVE compiler option is ON at the time the routine is first declared (see Appendix C).
- b. In a CDS environment whether the RECURSIVE compiler option is ON or OFF (RECURSIVE is ignored in CDS programs as all procedures and functions can be recursive).

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

Example

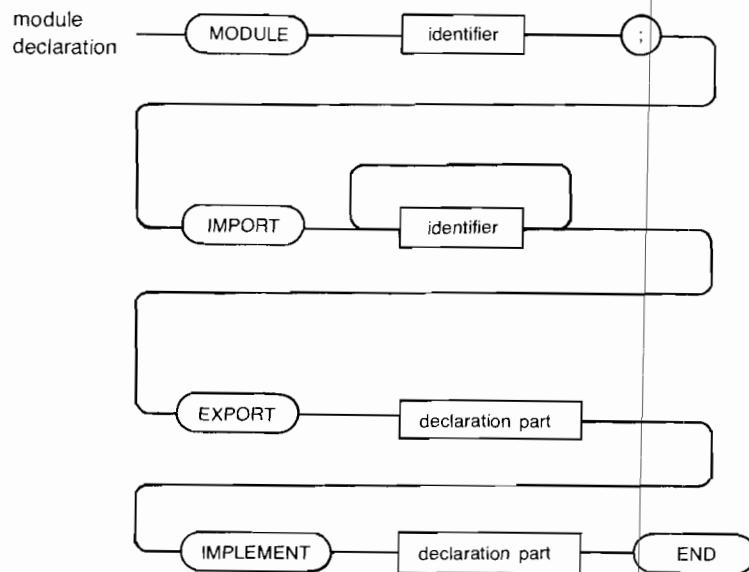
```
{ calculate factorial recursively }

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

Module Declaration

Modules are program fragments which can be compiled separately, and used to provide program objects to one or more programs.

Syntax



The declaration and use of modules is described in this section, organized as follows:

Module Overview

Module Structure

Module Heading

Import Section

Export Section

Implement Section

Module Usage

Where Modules May Be Declared

Where Modules May Be Imported

How Module Interfaces Are Found by the Compiler

Module Interface Search Pattern

Module Access Rights

Explicit Import Access Rights

Implicit Import Access Rights

Accessing EXTERNAL Routines from a Module

Dangling Pointer Access Rights

Dangling Pointers

Module Overview

Modules perform several major functions:

Program Structuring

Large programs are made up of a number of logical groupings of program objects that communicate with each other. The module facility is a convenient tool for creating and managing these logical groupings, and controlling the interfaces between them.

Encapsulation

By using modules, a programmer can control the interface to a logical grouping of program objects. That is, which objects are visible to other parts of the program, and which objects can be used only from within the module may be specified. By knowing which objects are invisible and visible, a programmer knows which objects can and cannot be changed without affecting the operation of other parts of the program. Looking at it another way, as long as the module interface is kept the same, drastic changes can be made to the invisible objects without worrying about affecting the users of the module.

Type-Safe Separate Compilation

The module facility provides a way to compile pieces of a program separately and yet preserve Pascal's parameter type checking.

Library Creation

Since modules are independent collections of objects, they may be used to create general-purpose libraries of routines and data that can be loaded with many different programs.

A module can provide a program with several different kinds of program objects. That is, a module can establish some objects, and they can be referenced elsewhere in the program. A module can supply any combination of the following:

Definitions

Constants and types may be defined in a module and used elsewhere in the program.

Data

Variables may be declared in a module, and referenced elsewhere in the program.

Services

Procedures and functions may be declared in a module, and called from other parts of the program.

Here are two examples of modules:

```
MODULE data;  
  
EXPORT  
  TYPE  
    CHARACTERS = CHAR;  
    TRUE_OR_FALSE = BOOLEAN;  
  
IMPLEMENT  
END;  
  
MODULE upshifter;  
  
IMPORT  
  data;  
  
EXPORT  
  FUNCTION upper_case (ch: CHARACTERS): TRUE_OR_FALSE;  
  
IMPLEMENT  
  FUNCTION upper_case (ch: CHARACTERS): TRUE_OR_FALSE;  
  BEGIN  
    upper_case := ch IN ['A'..'Z'];  
  END;  
END.  
END.
```

Declarations

Module Structure

The four parts of a module are:

- Heading
- Import section
- Export section
- Implement section

The heading introduces the module and names it. The name is a Pascal identifier.

The import section names all other modules on which the present one depends. One module, m1, depends on another, m2, if m1 makes use of the objects exported from m2. For example, m1 calls procedures in m2, or assigns to m2's variables, or declares variables of a type exported from m2. There is no import section if the module is independent of all other modules.

The export section defines the constants, types, variables, procedures, and functions which the module will supply to any program or module which imports it. Constants, types, and variables are declared just as in a program or procedure block. Procedures and functions are presented as headings without bodies. The export section may make use of things that were exported from modules listed in the import section.

Every module must have an export section.

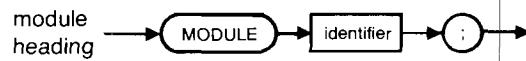
The implement section defines the constants, types, variables, procedures, and functions which will be known only within the module. In addition, it contains the bodies of the procedures and functions whose headings appeared in the export section. A module does not have to export procedures or functions. It may be used simply to create data or data types. In such a case, there will be nothing between the words IMPLEMENT and END. That is, every module must have an implement section, but it may be empty.

A more detailed definition of the sections of a module appears below.

The import and export sections define the module's interface to other modules or programs. This interface is public; the information it contains is available to any importer of the module.

The implement section contains the private parts of the module: everything between the words IMPLEMENT and END is hidden, even from the importers of the module.

The private and public parts of the module are separated in this way so that the private part can safely be changed without altering programs or other modules which import it. Another implication is that modules can only be dependent on other modules, not on programs. This independence of modules from programs is a key to developing software libraries.

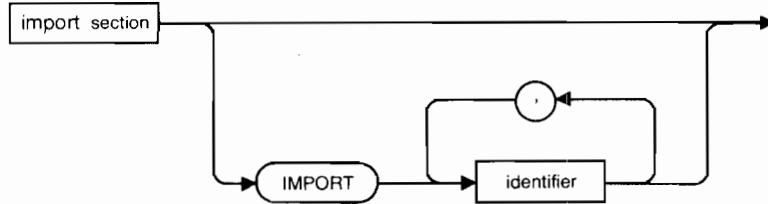
Module Heading**Syntax**

The module heading is used to start the definition of a new module. It consists only of the reserved word MODULE, and the module identifier.

In Pascal/1000, the module name is used in creating the entry point names for the level-1 routines within the module, and thus should be chosen carefully. Module naming conventions are described in Chapter 8.

Import Section

Syntax



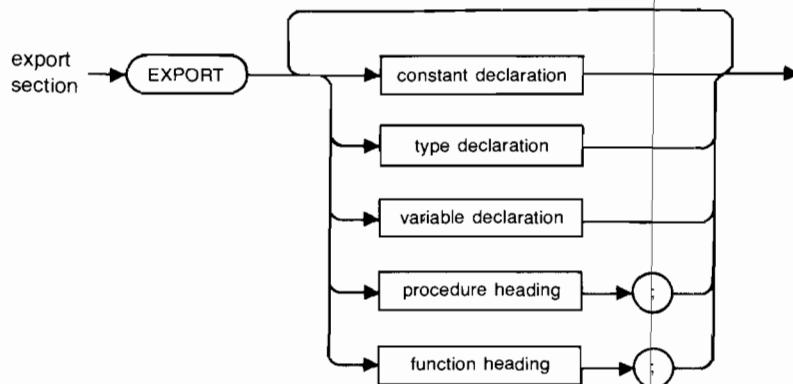
The import section of a module M identifies those modules upon which module M is dependent. M has access to the objects exported by all of the modules in the list, and cannot access the exported objects of any other module (see Module Access Rights).

Before module M is compiled, each module in the list must be compiled, and the compiler must have access to its module interface information (see How Module Interfaces Are Found by the Compiler).

The import section in a module is optional. If no import section appears, the module is independent of all other modules.

Export Section

Syntax



The export section of a module M defines which objects are to be accessible to any importer of the module. Exported objects may be constants, types, variables, procedures or functions.

The constants and types used in defining and declaring one of M's exported objects, x, must be defined in one of the following places:

- In the export section of a module that M imported.
- In the export section of M, prior to the definition of x.
- In the export section of M, after the definition of x. This is allowed only if x is the pointer type $\wedge y$, and y is defined after the definition of x.
- In the implement section of M. This is allowed only if x is the pointer type $\wedge y$, and y is defined in the implement section. x is then called a "dangling pointer type". See Dangling Pointers, below.

Most items must be completely defined or declared in the export section. That is, all of the constants and types which are used to define the object must be defined in the export section of M, or they must be imported from other modules.

Other items may be partially specified in the export section, and partially specified in the implement section:

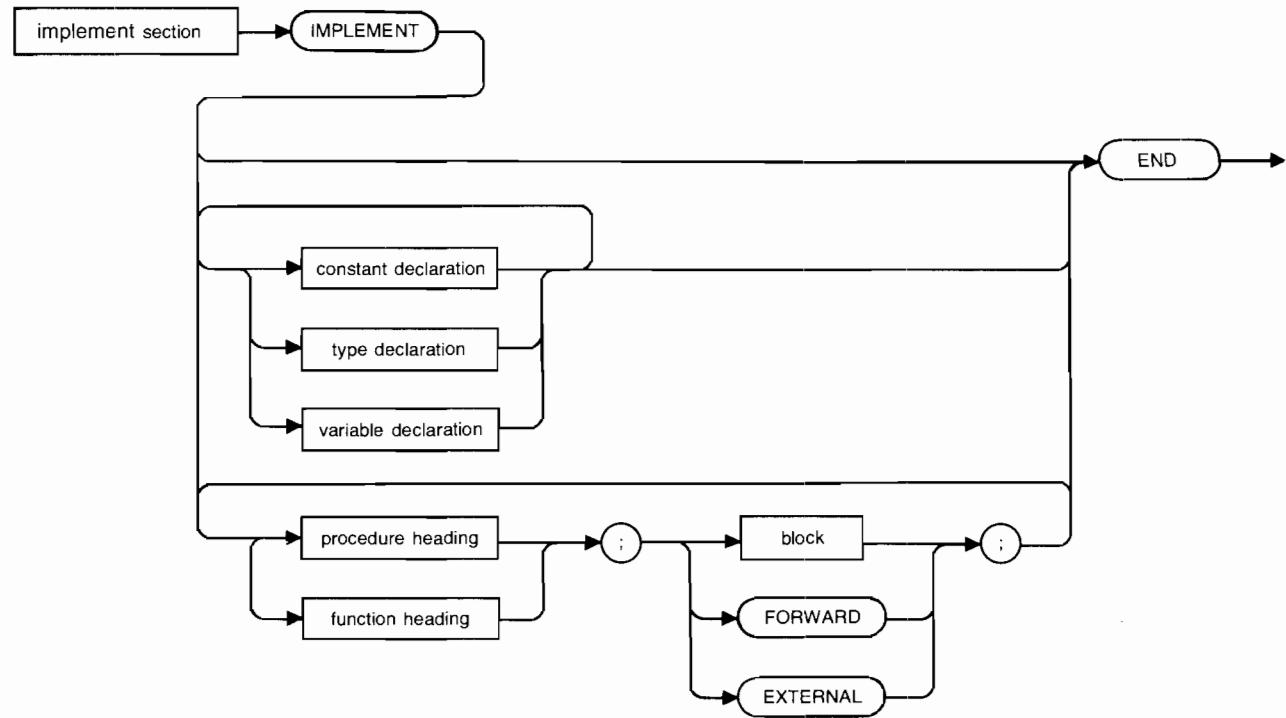
- Constants, variables, and all type except dangling pointer types must be completely specified in the export section.
- The name of a dangling pointer type, and the name of its base type are defined in the export section. The implement section contains the actual definition of the base type.
- The headings of exported procedures and functions appear in the export section. The bodies of the routines must appear in the implement section (except at \$STANDARD_LEVEL 'hp1000', see Accessing EXTERNAL Routines from a Module below).

The export section is required in the definition of a module, and it may not be empty. (A module which did not export anything would not be useful).

Implementation Restriction: HP Pascal allows data declarations (constants, types, and variables) to be intermixed with the procedure and function headings. Pascal/1000 requires all data declarations to precede any procedure or function headings.

Implement Section

Syntax



The implement section contains the details of the module that are to remain hidden from other modules, including importers of the module.

These details include:

- Constant definitions
- Type definitions
- Variable declarations
- Headings and bodies of unexported procedures and functions
- Headings (with or without the parameter list) and bodies of exported procedures and functions
- Base types of dangling pointer type

The implement section must include:

- A heading and body (or directive) for each exported procedure and function
- A base type definition for each dangling pointer type

The implement section of a module is required, but may be empty. It is common for a module to export only constants, types, and/or variables.

Module Usage

Where Modules May Be Declared

A module may be declared in either a main program unit or a module-list unit:

```

main program unit
  program heading
  global data declarations
  module-list
  import section
  main program procedures and functions
  main body of program

module-list unit
  module-list

```

MODULE and IMPORT declarations in the main program unit can be intermixed with the CONST, TYPE, and VAR sections and may be repeated as often as is required.

A module may not be declared in another module, nor in a subprogram or segment compilation unit.

Examples

{Main program unit}

```

PROGRAM p;
VAR i: INTEGER;
MODULE m1;
EXPORT VAR m1_i: INTEGER;
IMPLEMENT
END;

IMPORT
  m1, m2, m3;

BEGIN
  i := 0;
  m1_i := 0;
  m2_i := 0;
  m3_i := 0;
END.

```

{Module-list unit}

```

MODULE m2;
EXPORT VAR m2_i: INTEGER;
IMPLEMENT
END;

MODULE m3;
EXPORT VAR m3_i: INTEGER;
IMPLEMENT
END.

```

Where Modules May Be Imported

A module may be imported into a program or another module.

Examples

```
{Importing into          {Importing into
  a Program}           a Module}

PROGRAM p;
VAR i: INTEGER;
MODULE m1;
EXPORT m1_i: INTEGER;
IMPLEMENT
END;

IMPORT
  m1, m3;

BEGIN
  i := 0;
  m1_i := 0;
  initialize;
END.

{Importing into          MODULE m2;
  a Module}           EXPORT VAR m2_i: INTEGER;
IMPLEMENT
END;

MODULE m3;
IMPORT m2;
IMPLEMENT
  EXPORT
    VAR m3_i: INTEGER;
    PROCEDURE initialize;
BEGIN
  IMPLEMENT
    PROCEDURE initialize;
BEGIN
  m2_i := 0;
  m3_i := 0;
END;
END.
```

In Pascal/1000, modules can also be imported into subprograms and segments.

How Module Interfaces Are Found by the Compiler

When a program p or module m1 imports another module m2, the compiler must be able to access m2's exported information. This information is needed to check for the proper access and usage of m2's objects by p or m1.

Modules must be compiled before they can be imported. When the source file containing module m2 is compiled, the resulting relocatable file carries with it a description of m2's interface.

When p or m1 imports m2, the compiler then reads this interface information. The \$SEARCH\$ directive is used to tell the compiler the names of the relocatable files that should be searched for m2's interface information (see below).

If, however, the compiler has already seen the interface information for m2 while compiling the current source file, then the information is not re-read. This would be true if the compiler had already seen a declaration for m2 in the current source file, or if m2 had been imported in a previous module declaration in the current source file.

In summary, these are the ways the compiler can access interface information for m2 while processing an IMPORT list:

1. m2 is declared in the same source file prior to the current IMPORT list.
2. m2 has already been imported by a module declaration prior to the current IMPORT list.
3. m2 is declared in a separate source file, and that file has been compiled, and the compiler has access to its relocatable file. Access is determined by the search pattern, described below.

Module Interface Search Pattern

When the compiler processes an IMPORT section, it must find the modules named in the import list and read their interface specifications. A particular search pattern is followed, which is repeated for each module in the list.

- If the imported module has been previously declared or imported in the source text being compiled, the reference is to that module.
- If no module of that name has been seen, the compiler searches relocatable files for the module's interface. The files to be searched, and their search order, may be specified by a \$SEARCH\$ directive.
- If the module is not found in any of the above, an error (module not found) is issued.

A module which is imported may itself import other modules, which are listed in its import section. The compiler follows this implicit import chain all the way back to its root, to a module which imports no others. The search pattern above is applied recursively, to a maximum depth of 10 levels (this maximum depth can be increased by using the \$INCLUDE_DEPTH\$ directive, as described in Appendix C).

This point bears repeating in another way: to construct the search list, the programmer must not only include the files that define the modules being imported, but all of the files that define all of the implicitly-imported modules as well.

In the example below, note the following:

- When m2 imports m3, the compiler has not seen a definition for m3 yet, so it goes to the search list, and finds m3 in the code file MODS_B.REL.
- When m1 imports m2, the compiler has already seen m2, and so its interface is known, and no further searching need be done.
- When m1 imports m3, m3's interface is already known because m2 imported it.
- When m1 imports "unknown_module", the compiler has not seen it in the current source, nor can it find it in the search list (m3 is the only module in MODS_B.REL, which is the only file in the search list). Thus an error message is printed.

Example

File MODS_A.PAS contains:

```
$SEARCH 'MODS_B.REL'$  
MODULE m2;  
IMPORT m3;  
EXPORT VAR m2_i: INT;  
IMPLEMENT  
END;  
  
MODULE m1;  
IMPORT m2, m3, unknown_module;  
EXPORT  
  VAR m1_i: INT;  
IMPLEMENT  
  PROCEDURE initialize;  
  BEGIN  
    m1_i := 0;  
    m2_i := 0;  
    m3_i := 0;  
  END;  
END.
```

File MODS_B.PAS contains:

```
MODULE m3;  
EXPORT  
  TYPE INT = -32768..32767;  
  VAR m3_i: INT;  
IMPLEMENT  
END.
```

Module Access Rights

A module can access any of the following:

- Explicitly-Imported Objects

Module A is said to be explicitly import module B if:

A imports B

Module A can access exported constants, types, variables, procedures and functions. Also, module A "knows" the attributes of the objects it imports, whether or not those objects are explicitly imported. More on this later.

- Implicitly-Imported Objects

Module A is said to implicitly import module C if:

A imports B

B imports C

A does not import C

Module A cannot access any constant, type, variable, procedure or function in Module C. But Module A can "know" certain things about the objects that Module C exports. More on this later.

- EXTERNAL routines

Pascal routines that are not declared within a module, or routines that are written in another language, can be called once an external declaration is made for the routine in the implement section of that module. More on this later.

- Predefined Objects:

Predefined types

Predefined constants (minint, maxint, false, true)

Predeclared files (input and output). See note below.

Predefined Procedures and Functions

NOTE

The main program must mention input (and/or output) in the program heading if any module references input (and/or output). Otherwise the program will not load.

- Objects declared within the module

Any constant, type, variable, procedure or function that is declared either in the export or implement section can be accessed within that module. Normal Pascal scoping rules apply to such things as nested routines, record field identifiers, etc.

A module cannot access the following:

- Program objects

Labels, constants, types, and variables that are global to the program.

- Module objects

Constants, types, variables, and routines that are not exported from another module.

Explicit Import Access Rights

When A imports B, A's access rights to B's exported objects are as follows:

Constants

A constant exported from B can be used in A anywhere a constant expression is allowed. It can be used in

- constant expressions
- type definitions
- expressions
- CASE constant lists

Module A knows the type and value of the constant.

Types

A type exported from B can be used in A anywhere a type identifier is allowed. It can be used in

- type definitions
- structured constants
- variable declarations
- formal parameter lists

Module A knows the details of the exported type T:

- If T is a record, its field names and types are known.
- If T is an array, its index and component types are known.
- If T is a set, its base type is known.
- If T is a file, its component type is known.
- If T is an enumeration, its elements are known.
- If T is a subrange, its base type and bounds are known.
- If T is a pointer, then its base type BT is:
 - known if BT is defined in the export section, or BT is defined in a module that B imported,
 - unknown if BT is defined in the implement section. (see Dangling Pointers, below)

Variables

A variable exported from B can be used in A anywhere a variable is allowed. It can be used

- in an expression
- on the left-hand-side of an assignment statement
- as an actual VAR parameter

Module A knows the variable's type and its attributes.

Procedures (Functions)

A procedure (function) exported from B can be used in A anywhere that a procedure (function) identifier is allowed. It can be

- called from any routine in A
- passed as an actual PROCEDURE (FUNCTION) parameter

Module A knows the number and types of all the parameters in the procedure's (function's) parameter list, and function return types are known. No EXTERNAL declaration need be made in A for B's routines.

Implicit Import Access Rights

Module A is said to implicitly import module C if:

- A imports B
- B imports C
- A does not import C

Module A cannot access any constant, type, variable, procedure or function in Module C. But Module A can "know" certain things about the objects that Module C exports.

The following are A's access rights to C's exported objects:

- Constants
- Variables
- Procedures
- Functions

Module A cannot access any of these objects in Module C, nor does it know any of their attributes.

Types

An implicitly-imported type defined in Module C cannot be explicitly used in Module A. That is, it cannot be used to declare variables, nor to define structured constants, formal parameter lists, nor other types.

But Module A does know the details of an implicitly-imported type T:

- If T is a record, its field names and types are known.
- If T is an array, its index and component types are known.
- If T is a set, its base type is known.
- If T is a file, its component type is known.
- If T is an enumeration, its elements are known.
- If T is a subrange, its base type and bounds are known.
- If T is a pointer, its base type is known, unless the base type is defined in the implement part (see Dangling Pointers).

For example, Module C exports:

```
REC = RECORD
  a,b: INTEGER;
END;
```

and Module B imports Module C and exports:

```
VAR r: REC;
```

and Module A imports B.

REC, then, is an implicitly-imported type. A did not import C, but the attributes of REC are still known within A. The compiler knows that r has an integer field called a, and thus can allow, within A, a statement such as the following:

```
r.a := 5;
```

Accessing EXTERNAL Routines from a Module

Pascal routines that are not declared within a module, or routines that are written in another language, can be called from routines within a module.

Before any calls appear, there must be an EXTERNAL declaration for the routine that is to be called (see EXTERNAL routines in Chapter 4 to see how to make an EXTERNAL declaration).

The EXTERNAL declaration must appear in the implement section. The actual declaration of the routine must be in a separate compilation unit, and must not be within a module.

At STANDARD_LEVEL 'HP', an EXTERNAL routine cannot be exported (the actual routine must appear in the implement section if the heading appeared in the export section).

At STANDARD_LEVEL 'HP1000', an EXTERNAL routine can be exported (an EXTERNAL declaration can appear in the implement section if the heading appeared in the export section). This permits the actual routine to be written in another language (or to be a system service or library routine) while providing the type safety and convenience of the module mechanism. For example, a module to interface to EXEC or FMP routines can be written using this mechanism. However, since the compiler normally constructs a name for exported routines, the ALIAS mechanism may have to be used to specify the desired name of the external routine (see Creating Unique External Names in the Modules section of Chapter 8).

Example

```

MODULE show_external;

EXPORT
  TYPE
    INT = -32768..32767;

    TIME_INFO = RECORD
      tens: INT;
      secs: INT;
      mins: INT;
      hours: 0..23;
      day: 1..366;
      year: 1900..3000;
    END;

    PROCEDURE get_time (VAR t: TIME_INFO);

IMPLEMENT

  PROCEDURE system_time
  $ALIAS 'EXEC'$'
  (
    exec11: INT;
    VAR t: TIME_INFO;
    VAR year: YEARS);
  EXTERNAL;

  PROCEDURE get_time
  (VAR t: TIME_INFO);
  BEGIN
    system_time (11, t, t.year);
  END;
END.

```

Declarations

Dangling Pointer Access Rights

The base type of a pointer type may be "hidden" using a "dangling pointer", a pointer type whose base type is defined in the implement section.

Consider the two modules:

```
MODULE public;
EXPORT
  TYPE
    INFO_PTR = ^INFO;
    NAMES = STRING [30];
    ADDRESSES = STRING [30];
    INFO = RECORD
      name: NAMES;
      address: ADDRESSES;
    END;
IMPLEMENT
END.

MODULE private;
EXPORT
  TYPE
    INFO_PTR = ^INFO;
IMPLEMENT
  TYPE
    NAMES = STRING [30];
    ADDRESSES = STRING [30];
    INFO = RECORD
      name: NAMES;
      address: ADDRESSES;
    END;
END.
```

Normally, as in module public, when a pointer type is exported, its base type is exported along with it. This allows an importer to access both the pointer itself, and the objects it points to.

However, when the base type is not exported, as in module private, the importer is restricted in its use of the pointer type.

- Dangling pointer variables may be assigned or passed or compared to other variables of the same type.
- The objects of dangling pointer variables may not be accessed in any way.
- Dangling pointer variables may not be passed as arguments to the standard procedures new, dispose, mark, or release.

Example

```
MODULE user1; IMPORT public;
EXPORT PROCEDURE a;
IMPLEMENT
  PROCEDURE a;
  VAR p, q: INFO_PTR;
  BEGIN
    new (q); {ok}
    p := q; {ok}
    p^ := q^; {ok}
  END;
END.

MODULE user2; IMPORT private;
EXPORT PROCEDURE a;
IMPLEMENT
  PROCEDURE a;
  VAR p,q: INFO_PTR;
  BEGIN
    new (q); {ERROR}
    p := q; {ok}
    p^ := q^; {ERROR}
  END;
END.
```

Dangling Pointers

Dangling pointers may be used to purposely prohibit access to implementation details of Pascal data structures. Importers may assign, pass, and compare these pointers, but may not access the variables they point to. All such access must be done by routines in the module which are exported for that purpose.

The following are some of the reasons why this is sometimes useful:

- The architect of a module needs to supply the importers with a particular data structure, but wants the freedom to change its implementation as the module requirements change (needs it to go faster, needs more information in the data structure, etc).

If the full details of the data type are exported, importers may use (and thus rely on) attributes of that type. It may then be very difficult to change the data structure, because knowledge of its representation may be scattered throughout many modules.

If, on the other hand, only the pointer type, is exported the data structure may be changed at any time. No other module in the system need be changed. (Importers must be recompiled, however).

As an example, Joe exports a pointer to an exported record type which contains two fields, a Boolean named b, and an array of integers named a. He exports the pointer and its base type, and his importers begin accessing the fields of the record (e.g. `p^.a[5]`).

Joe later decides that the Boolean field is unnecessary, and that the data structure should be just the array. Now, every importer must change their source code to access the data in the new way (e.g. change `p^.a[5]` to `p^[5]`).

Had Joe exported just the pointer type and a routine for accessing the array, he could have made the change without affecting his importers.

- The architect of a module needs to control the access to, or maintain statistics about, the objects of a certain type.

For example, Joe creates a pointer to a symbol table record. He exports the pointer type, along with all the necessary routines for creating, disposing, and accessing (only the necessary) information from the symbol table.

Now Joe knows that there is only one place in the entire program where symbol table entries may be created, initialized, and accessed by field name. He has created routines for importers to retrieve and assign necessary information (like symbol name, etc), but has not provided any way to access the information in the record that he wants to remain unknown.

Also, Joe can accurately instrument symbol table access (how many entries are created, what kinds are created most often, etc.).

Example

```
MODULE symbol_table;

EXPORT
  TYPE
    NAMES = STRING [30];
    SIZES = (small, medium, large);

    SYMBOL_TABLE_PTR = ^SYMBOL_TABLE;

    PROCEDURE give_name (s: SYMBOL_TABLE_PTR; name: NAMES);
    PROCEDURE give_size (s: SYMBOL_TABLE_PTR; size: NAMES);
    FUNCTION name_of_symbol (symptr: SYMBOL_TABLE_PTR): NAMES;
    FUNCTION size_of_symbol (symptr: SYMBOL_TABLE_PTR): SIZES;

IMPLEMENT
  TYPE
    SYMBOL_TABLE = RECORD
      name: NAMES;
      size: SIZES;
    END;

    PROCEDURE give_name (s: SYMBOL_TABLE_PTR; name: NAMES);
    BEGIN ... END;
    PROCEDURE give_size (s: SYMBOL_TABLE_PTR; size: NAMES);
    BEGIN ... END;
    FUNCTION name_of_symbol (symptr: SYMBOL_TABLE_PTR): NAMES;
    BEGIN ... END;
    FUNCTION size_of_symbol (symptr: SYMBOL_TABLE_PTR): SIZES;
  BEGIN ... END;
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 and is defined from its point of declaration or definition.

The precise scope rules for HP Pascal are:

- a. The scope of an object extends over the whole of the program, procedure, function, module, 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 (or exported) 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: procedures and functions not declared within a module can access objects declared in the program level declaration part. Procedures and functions declared within a module can access objects declared in, or imported into, the module. (They may not access objects declared in the program level declaration part.) They may access the predefined files INPUT and OUTPUT provided they are mentioned in the main program heading.
- 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).

Declarations

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}

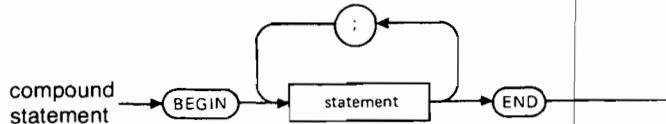
BEGIN {program levels}
  {can use global labels, constants, types, and variables}
  {can use proc1                                         }
END.
```

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.

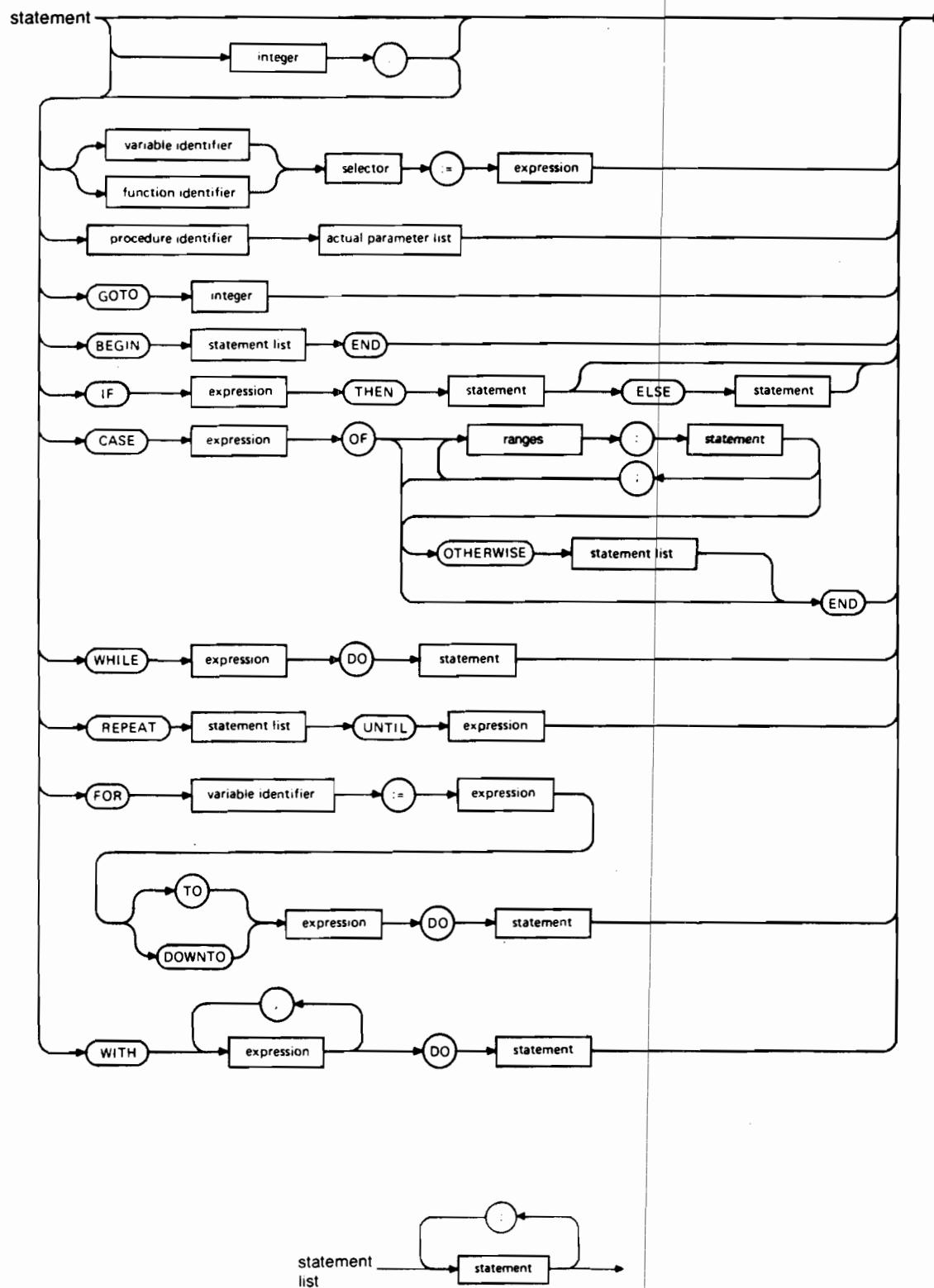
Statements

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).

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 contain other statements. There are no restrictions on the number of structured statements that may be nested, nor on the number of statements in a body.

Syntax:

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

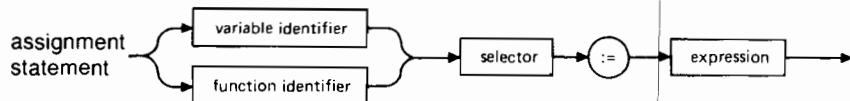
```
PROCEDURE show_labels;
LABEL 500, 501;
TYPE
  INDEX = 1..10;
VAR
  i: INDEX;
  target: INTEGER;
  a: ARRAY [INDEX] OF INTEGER;
BEGIN {show_labels}

  ...
  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 runtime 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. At least one assignment must be made in the function's body during each activation of the function.

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 (VMA/EMA) programs. Due to mapping constraints, an assignment may not assign a variable to or from the heap if its size is larger than 1024 16-bit words. The variable may be assigned one component at a time, provided each component is 1024 words or less.

Example

```

FUNCTION show_assign: INTEGER;

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

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

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

VAR
  s: INTEGER;
  a: TABLE;
  i: INDEX;
  r: REC;
  p1,
  p: ^INTEGER;

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 := p1; {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\$, \$NOABORT\$)
- e. location of the procedure's body (FORWARD, EXTERNAL)
- f. parameter addressing (\$HEAPPARMS\$)
- g. parameter conversion (\$FIXED_STRING\$, \$BASIC_STRING\$)

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

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
  writeln (output, 'Hi');
  actual (2, 4.0);
  inner;
  external_proc (2, 4.0);
  proc (2, 4.0);
END;    {outer}

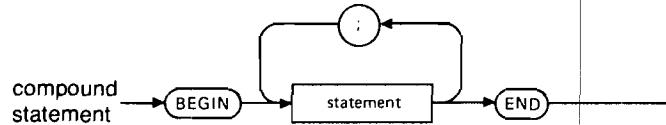
PROCEDURE forward_proc           {the actual declaration for a forward  }
                                {procedure                      }
BEGIN
  {call a routine with procedural
   parameters:
    external,
    forward,
    and actual procedures}
  outer (10, external_proc);
  outer (20, forward_proc);
  outer (30, actual);
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
  IF min > max THEN
    BEGIN
      error ('min is wrong ');
      min := 0;
    END;
  END;

  BEGIN
    BEGIN
      start_part_1;
      finish_part_1;
    END;

    BEGIN
      start_part_2;
      finish_part_2;
    END;
  END;

```

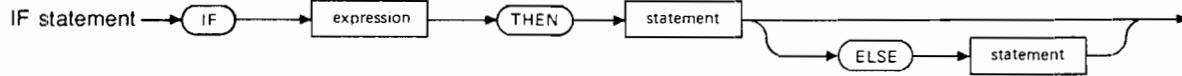
{This }
 {compound }
 {Compound } {statement }
 {statement is} {is }
 {part of IF } {the }
 {statement } {procedure's}
 {body }

{Nested compound statements }
 {for logically grouping statements}

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:

<pre>IF a = b THEN IF c = d THEN a := c ELSE a := e;</pre>	<pre>IF a = b THEN BEGIN IF c = d THEN a := c ELSE a := e; END;</pre>
--	---

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 in chapter 5 for more information.) Partial evaluation usually results in more efficient code, but it 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.

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');

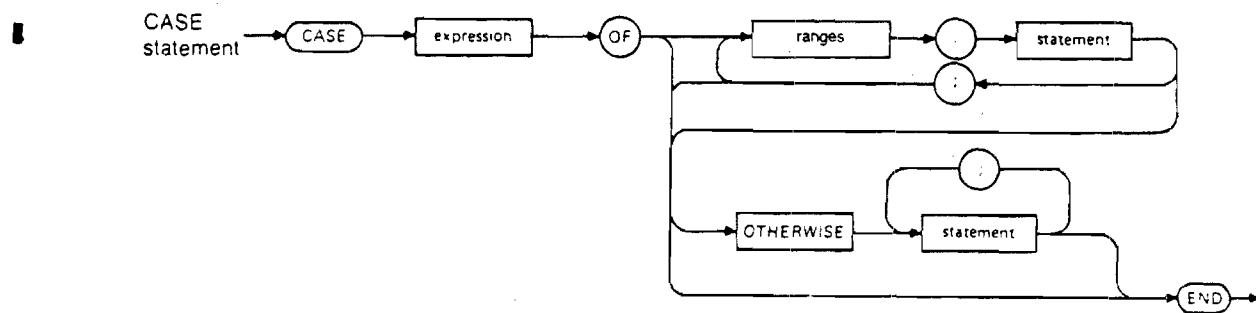
  IF s = 'RED' THEN              {This is similar to a  }
    i := 1                        {CASE statement for a  }
  ELSE IF s = 'GREEN' THEN        {'PAC' expression      }
    i := 2
  ELSE IF s = 'BLUE' THEN
    i := 3

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 does not 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.
 - 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}

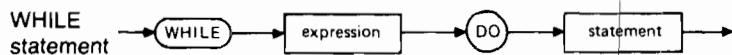
BEGIN {show_case}
  c := new_color (red, yellow);
END. {show_case}

```

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:

<pre>IF condition THEN BEGIN statement; WHILE condition DO statement END;</pre>	<pre>1: IF condition THEN BEGIN statement; GOTO 1; END;</pre>
---	---

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

```

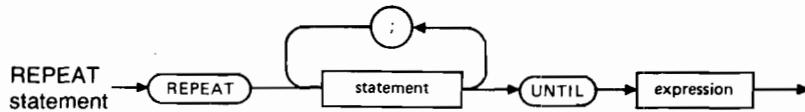
WHILE index <= limit DO BEGIN
  writeln (real_array [index]);
  index := index + 1;
END;

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 each 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

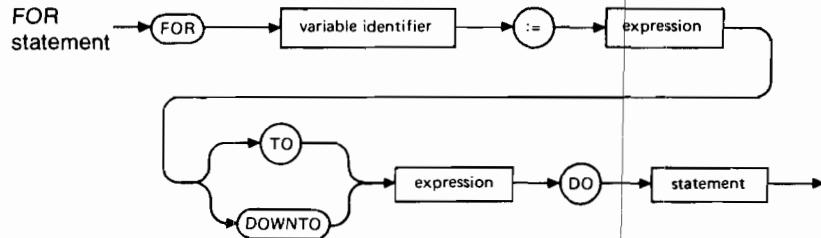
```

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.

Executable Parts

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 THEN
    x [i] := cos (i)
  ELSE
    x [i] := sin (i);

```

If a FOR statement is used in a section of a program where no range checking occurs (\$RANGE OFF\$) caution should be taken when mixing single and double integer expressions.

Example

```

VAR
  i : -32768..32767;
  initial, final :INTEGER;
  :
$RANGE OFF$
initial := 1;
final := 65546;
FOR i := initial TO final DO
  :

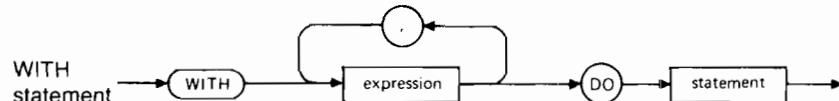
```

Only the first word of the double integers are used when the statement is executed since i is a single-word integer. Although the integer final equals 65546, the first word of its internal representation is identical to the internal representation of a single-integer whose value is 10. Thus the FOR statement will be executed ten times.

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:

<pre> WITH rec DO BEGIN field1 := e1; writeln (field1*field2); END; </pre>	<pre> BEGIN rec.field1 := e1; writeln (rec.field1 * rec.field2); END; </pre>
--	---

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

<pre> i := i1; WITH a[i] DO BEGIN writeln (f); or i:=i2; writeln (f) END; </pre>	<pre> p := p1; WITH p^ DO BEGIN writeln (f); p := p2; writeln (f) END; </pre>
---	---

produces the same effect as:

<pre> writeln (a[i1].f); or writeln (a[i1].f); </pre>	<pre> writeln (p1^.f); writeln (p1^.f); </pre>
--	--

The appearance of a function reference in the expression list of a WITH statement is an invocation of the function. Thus, a WITH statement cannot be used to set the results of a function that returns a record. For example, the function reference in the WITH statement of the following invokes the function.

```

FUNCTION f: RECORD_TYPE
BEGIN
  WITH f DO BEGIN {this is a recursive invocation, not}
  .                               {a reference to the function result.}
  .
  END;
END;

```

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. For further implementation considerations, see Chapter 8.

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 notes:

- Pascal/1000 does not prohibit GOTO's into a structured statement, but their use is highly discouraged.
- GOTO's 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 operator 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. Integer literals must be in the range of -2147483648 to 2147483647.

Syntax

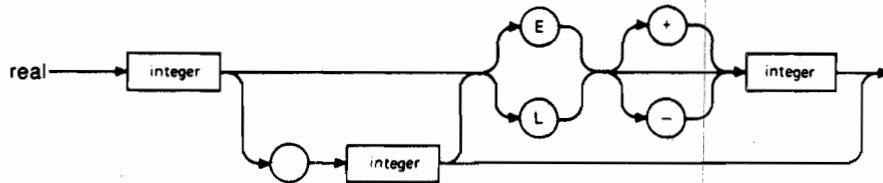


Example

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

Real literals must be in the range of:

-1.70141E+38 TO -1.4693683E-39
 0.0
 1.4693679E-39 TO 1.70141E+38

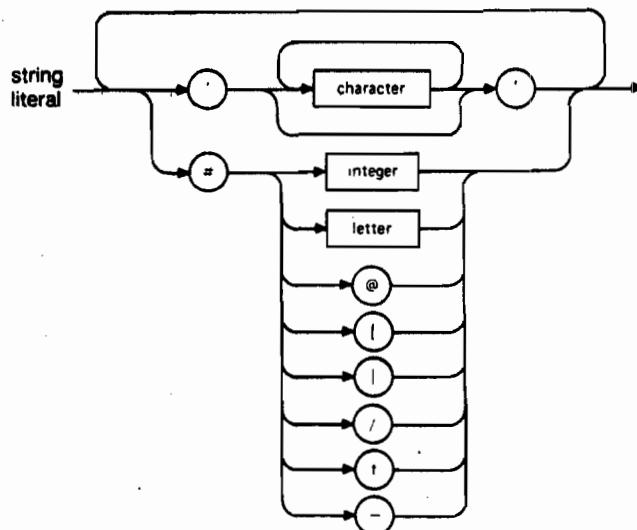
Constants of type longreal must be in the range of:

-1.70141183460469231L+38 TO -1.46936793852785946L-39
 0.0
 1.46936793852785938L-39 TO 1.70141183460469227L+38

Executable Parts

String Literals — sequence of ASCII printable characters enclosed in single quote marks, a single character after a sharp symbol (#), or some combination of the two. A string literal is type CHAR, PAC, or STRING, depending on the context.

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.

In HP Pascal, non-printing ASCII characters may be included in strings by using an extended string syntax employing the sharp sign (#). The sharp sign followed by a non-numeric character or its decimal value (in the range 0 to 255) is used to encode an ASCII control character.

For example, the sequence #g represents the ASCII control character 'control'g. The sequence #7 also can be used to represent 'control'g, since 7 is the decimal value of the control character. Two consecutive quote marks ("") specify the null or empty string literal.

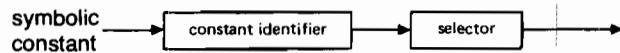
Examples

```
''          {this represents a null string}
''''        {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 side of an assignment statement, as an actual variable parameter, or as a FOR loop control variable. For example:

```

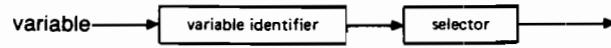
PROGRAM show_constants (output);

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          }
f1     {file variable            }
```

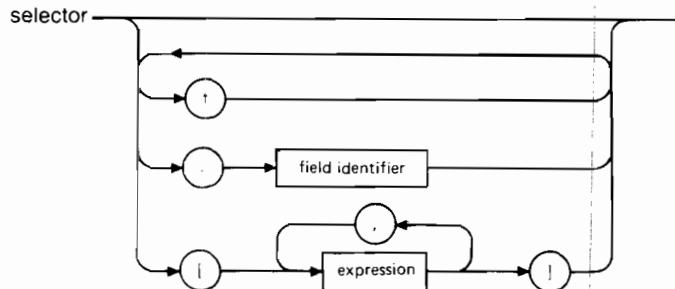
Selected variables:

```
a[i]    {indexed variable      (array component) }
r.f     {field variable        (record component)}
p^     {referenced variable    (pointer object)   }
f1^    {buffer variable         (file component)  }
m^.n[S] {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;
    g: REC_PTR;
    ff: INT_FILE;
  END;
  REC_ARRAY = ARRAY [1..3] OF REC;
  SMALL_STRING = STRING [5];
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;
  ss: SMALL_STRING;
  pt: ^TEXT;
FUNCTION func (fp: INTEGER): SMALL_REC; EXTERNAL;
FUNCTION pfunc: SMALL_REC_PTR; EXTERNAL;
  
```

Array and String 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 runtime, 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]
```

String components (of type CHAR) are selected using an integer (or subrange of integer) subscript. If the subscript is a constant expression, its value is checked at compile time to make sure its value is in the range 1..maximum-string-length (except VAR parameters of type STRING which cannot be checked). The value is always checked at runtime (whether or not RANGE is ON) to ensure that the component is within the current length of the string. The string denotation before the brackets may itself be a selected variable, constant, or function reference.

Examples

```
ss[i]
ss[20]
```

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 field.

Examples

```

WITH r[1], g DO BEGIN
  f := 5;
  s := 'C';
END;
c$ris
func(2).s

```

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 runtime, unless the RANGE compiler option is turned off, the value of the pointer is checked to make sure it is not 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^
r[1].q^
r[1].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.ffa^
pt^^

```

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:

- Number, order, and type of operands
- Result type
- 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	<, <=, <>, =, >=, >, IN

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:

$$(x + y * z) \quad \text{and} \quad (x + (y * z))$$

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

$$(x + y + z) \quad \text{and} \quad ((x + y) + z)$$

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.

Table 5-1. Pascal Operators

Operator	Meaning
+	numeric UNARY PLUS and ADDITION; set UNION; string CONCATENATION
-	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

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 as follows (with REM representing the remainder of i DIV j):

```
If i > 0 and j > 0 then i MOD j = i REM j
If i = 0 and j > 0 then i MOD j = 0
If i < 0 and j > 0 then (where k = ABS(i) REM j)
    If k = 0 then i MOD j = k
    else i MOD j = j - k
If Any i and j <= 0 then i MOD j = Runtime error, MOD by Invalid Value
i DIV j = trunc (i/j)
```

Both operands must be INTEGERS for both DIV and MOD.

Neither out-of-range values nor overflow are detected during the evaluation of an expression. If the final result is assigned or passed to a subrange type when RANGE checking is on, and is out-of-range, the out-of-range condition will elicit a run-time error. If an expression consists of only one-word integers and the result of the expression does not need to be a two-word integer; for example, it is not assigned, passed, or used in a context which implies a two-word INTEGER result, overflow of a one-word integer value will not be detected. If an expression consists of only one-word integers and the result of the expression needs to be a two-word integer, one-word integer overflow will not occur and a correct two-word result will be obtained as long as two-word integer overflow does not occur, which will not be detected.

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 3	+2
5 MOD (-3)	Runtime error, MOD by invalid value
-5 MOD 3	+1
-5 MOD (-3)	Runtime error, MOD by invalid value

Executable Parts

Boolean Operators

The Boolean operators perform logical functions on Boolean operands 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 manipulates 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.

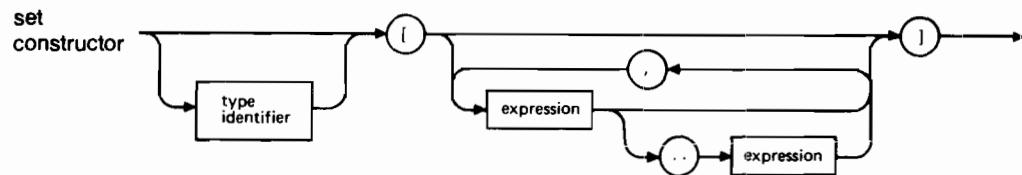
Narrowing of sets is also automatic for set assignments and actual value parameters. Compile- and run-time range checks are performed to verify that the narrowing of a set does not discard any elements. For example, in the assignment

```
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 runtime 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. Relational operators 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 Relational—The relational operators 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 relational operators 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/PAC Relational—Two PACS or string values can be compared using the operators =, <>, <, <=, >, or >=. If one of the PACS or strings is shorter than the other, it is padded on the right with blanks before the comparison. Two PAC or string values are equal only if their lengths (after blank padding) are equal and they contain identical character sequences. If two PAC or string values are equal through the length of the shorter value, the shorter value is considered to be less. If two PAC or string values are not equal, the result of the comparison is the result of comparing the first pair of unequal corresponding characters.

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 Pascal/1000, an additional restriction exists for HEAP 2 (VMA/EMA) programs: variables or components of variables, in the heap, that require 1025 or more 16-bit 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.
```

String Operator

One infix operator is defined in HP Pascal which manipulates two expressions having compatible string types and results in a third string.

Concatenation (+) —The result of this operator is the concatenation of the operator's two string operands. The operands may be string variables, string literals, function results of type STRING, or some combination of these types.

Examples Result

'abc' + 'def'	'abcdef'
'Pascal' + '/' + '1000'	'Pascal/1000'

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 B, 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 constantexpression. 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)
- hex
- binary
- octal
- strlen

Operands

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

Other operators, such as the relational, 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. Anywhere that a string literal can be used, a constant identifier which is equated to a string literal can be used.

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

REAL or LONGREAL constant expressions will elicit a warning at a STANDARD_LEVEL other than 'HP1000'. Constant expressions cannot include mixed-mode arithmetic:

```
CONST
  pi    = 3.0 + 0.1415926; {{correct for STANDARD_LEVEL 'HP1000')}
  badpi = 3    + 0.1415926; {{incorrect}}
```

Examples

```
CONST
  pi = +3.14159;
  pi_sqr = pi * pi; {correct with STANDARD_LEVEL 'HP1000'}
  num_symbols = 5;

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

CONST
  syms = SYM_ARRAY [succ(2) OF 'A',
                    abs (-2) OF 'B',
                    hex ('1') OF 'C'];

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, or
4. 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 both real types.
5. T1 and T2 are PAC types (PAC literals or variables). If either of T1 or T2 has a length less than that of the other, the shorter is temporarily blank filled to the length of the longer.
6. T1 and T2 are string types.

Assignment Compatibility

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 is REAL and T2 is LONGREAL. Loss of significant digits will occur beyond those representable in a variable of type REAL.
6. T1 and T2 are compatible ordinal types and the value of type T2 is in the closed interval specified by the type T1.
7. 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.
8. T1 and T2 are PACS and the length of T2 is less than or equal to the length of T1.
9. If T1 and T2 are string types then the length of the value of T2 must be less than or equal to the maximum length of T1. The length of the variable of type T1 is set to the length of the value of type T2.

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.

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.

- A VAR parameter of type STRING (no maximum length specified) is considered to be identical to any string type.

Chapter 6

Files

Introduction

Files allow a program to communicate with its environment, read information from external sources and create new information of its own for other programs or people to read.

To use a file from a Pascal program, the programmer needs to understand the following: (except as noted, these are all described later in this chapter).

- The distinction between a *logical* and a *physical* file
- How to associate a logical and a physical file
- Pascal file types
- How to open a file
- How to declare files (see Chapter 4)
- What operations are allowed on files
- Pascal/1000 input/output implementation considerations (see Chapter 8):
 - Default file types and sizes
 - Scratch file creation
 - Interactive I/O
 - I/O to File System versus FMGR files

Pascal File Types

Pascal allows the declaration and use of three different kinds of files:

- Text files
- Sequential files
- Direct-Access files

Text files are files which are used for reading and writing lines of characters.

Sequential files are used for reading and writing data values in a sequential manner. Values of almost any Pascal data type may be used.

Direct-access files are used for reading and writing data values in a non-sequential manner. Values of almost any Pascal data type may be used.

Each of these file types is described below. Each description contains:

- Declaration format
- Description of the file type
- Characteristics of the file type
- Routines that operate on the files
- Examples

Text Files

Declaration

```
VAR f: TEXT;
```

Description

Text files are files which are used for reading and writing lines of characters. Using text files it is easy to:

Read normal ASCII files, with all input conversions (like ASCII-to-INTEGER) performed automatically.

Write normal ASCII files, with all output conversions (like INTEGER-to-ASCII) performed automatically.

The predefined files input and output are text files.

Characteristics

Each text file f has the following characteristics:

Structure

A text file is a sequence of lines. Each line is a sequence of characters followed by an end-of-line marker. These markers are created by the standard procedure writeln. The component type of a text file is the type CHAR.

File buffer variable

The file buffer variable, f^, is of type CHAR. All characters that are read from or written to a file pass through f^.

Open state

f is always in one of the states: read-only, write-only, or closed.

Current position

The current position points to the component character in the file which is to be read or written to next. It may also point to an end-of-line marker.

Physical file

A physical file is associated with the file variable (when the file is not in the closed state). Any operations performed on the logical file will be translated to operations on the physical file.

(The following are Pascal/1000-specific characteristics)

Line-size

A text file has a maximum number of characters that may appear on each line. This number is 128 characters by default, but may be changed with the \$LINESIZE\$ directive.

Buffers

A text file may have several internal buffers to improve I/O performance. The default number of buffers is 1, but may be increased with the \$BUFFERS\$ directive.

Carriage control mode

Each text file, while it is in the read-only or write-only state, may be in the carriage-control mode (This mode is specified when the file is opened). Carriage control characters are automatically written (or skipped, if reading) while doing I/O to a file opened in this mode.

Shared state

Each text file, while it is in the read-only or write-only state, may be in the shared mode. (This mode is specified when the file is opened). This allows a particular physical file to be opened to more than one logical file from one or more executing programs.

Routines

append	open a file for extension.
close	close a file.
eof	determine end-of-file status.
eoln	determine end-of-line status.
get	read the next file component into the buffer variable.
linepos	determine the position in the current line of a text file.
overprint	overprint the current line with the next text file line.
page	page eject between two lines of a text file.
prompt	leave the cursor at the end of a line.
put	write the buffer variable to the next file component
read	read one or more values from a file.
readln	read values from a text file, and advance to the next line.
reset	open a file for reading.
rewrite	open a file for writing.
write	write one or more values to a file.
writeln	write values and a line marker to a text file.

Example:

```
PROGRAM copy (input); {This program copies the first line
VAR
      {of one text file to another}
      list: TEXT;
      line: STRING [80];
BEGIN
  reset (input);
  rewrite (list, 'COPY.DAT');
  readln (input, line);
  writeln (list, line);
END.
```

Sequential Files

Declaration

```
VAR f: FILE OF t;
```

Description

File f is a sequence of components, all of type t. The file is a sequential file if it has been opened with reset, rewrite, or append. (If opened with the procedure open, it would be a direct-access file instead.)

The type t may be any Pascal type, except another file type, or a structured type which contains a file type component (e.g. an array of files).

Characteristics

A sequential file f has the following characteristics:

Structure

A sequential file is a sequence of file records, all with values of the same type. The length of each record is the same, and is determined by the component type. (e.g. a FILE OF INTEGER is represented as a sequence of two-word records.)

File buffer variable

The file buffer variable, f^, is of type t, the component type from f's declaration. All values which are read from or written to f pass through f^.

Open state

f is always in one of the states: read-only, write-only, or closed.

Current position

The current position points to the component in the file which is to be read or written to next. The current position is automatically advanced as records are read or written, and cannot be moved explicitly to arbitrary records in the file.

Physical file

A physical file is associated with the file variable (when the file is not in the closed state). Any operations performed on the logical file will be translated to operations on the physical file.

(The following are Pascal/1000-specific characteristics)

Buffers

A text file may have several internal buffers to improve I/O performance. The default number of buffers is 1, but may be increased with the \$BUFFERS\$ directive.

Shared state

A sequential file, while it is in the read-only or write-only state, may be in the shared mode. (This mode is specified when the file is opened). This allows a particular physical file to be opened to more than one logical file from one or more executing programs.

Routines

append	open a file for extension.
close	close a file.
eof	determine end-of-file status.
get	read the next file component into the buffer variable.
put	write the buffer variable to the next file component.
read	read one or more values from a file.
reset	open a file for reading.
rewrite	open a file for writing.
write	write one or more values to a file.

Example:

This program reads REALs from a sequential file. As it reads each REAL, it copies it to another sequential file, and prints its value to the text file output.

```
PROGRAM copy (data, output);

VAR
  data,
  out: FILE OF REAL;

  x: REAL;
  i: 1..10;

BEGIN
  reset (data);
  rewrite (out);
  FOR i := 1 TO 10 DO BEGIN
    read (data, x);
    write (out, x);
    writeln ('Component Number ', i:2, ' is ', x);
  END;
END.
```

Direct-Access Files

Declaration

```
VAR f: FILE OF t;
```

Description

File f is a sequence of components, all of type t. The file is a direct-access file if it has been opened with the procedure open. (If it has been opened with reset, rewrite, or append, it would be a sequential file instead).

The type t may be any Pascal type, except another file type, or a structured type which contains a file-type component (e.g. an array of files).

Characteristics

A direct-access file f has the following characteristics:

Structure

A direct-access file is a sequence of file records, all with values of the same type. The length of each record is the same, and is determined by the component type. (For example a FILE OF INTEGER contains 2-wd records.)

Unlike other files, direct-access files may have components whose values are undefined. For example, if only the components 1 and 10 have been written to f, then components 2 through 9 are undefined.

File buffer variable

The file buffer variable, f^, is of type t, the component type from f's declaration. All values which are read from or written to a file pass through f^.

Open state

f is always in one of the two states: read-write or closed.

Current position

The current position points to the component in the file which is to be read or written to next. The current position is automatically advanced as records are read or written, and can be moved explicitly to arbitrary records using the seek, readdir, and writedir procedures.

Physical file

A physical file is associated with the file variable (when the file is not in the closed state). Any operations performed on the logical file will be translated to operations on the physical file.

(The following are Pascal/1000-specific characteristics)

Buffers

A text file may have several internal buffers to improve I/O performance. The default number of buffers is 1, but may be increased with the \$BUFFERS\$ directive.

Shared state

Each text file, while it is in the read-only or write-only state, may be in the shared mode. (This mode is specified when the file is opened). This allows a particular physical file to be opened to more than one logical file from one or more executing programs.

Routines

close	close a file.
eof	determine end-of-file status.
get	read the next file component into the buffer variable.
lastpos	determine the index of the last component in a file.
maxpos	determine the maximum number of components in a file.
open	open a file for direct access.
position	determine the current position in a direct-access file.
put	write the buffer variable to the next file component.
read	read one or more values from a file.
readdir	read one or more values from a direct-access file.
seek	position a direct-access file at a particular component.
write	write one or more values to a file.
writedir	write one or more values to a direct-access file.

Example:

This program creates a direct-access file, giving every other component, starting at 2, the value equal to its component number. (The value 2 is written into component 2, 4 into component 4, etc). It then verifies the contents in reverse order.

```

PROGRAM fill (data, output);

VAR
  data: FILE OF INTEGER;
  i: 1..10;
  j: INTEGER

BEGIN
  open (data);
  FOR i := 1 TO 10 DO BEGIN
    writedir (data, i*2, i*2);
  END;

  FOR i := 10 DOWNTO 1 DO BEGIN
    readdir (data, i*2, j);
    IF j <> i*2 THEN BEGIN
      writeln (output, 'Wrong value in component ', i*2);
    END;
  END;
END.

```

Associating Logical and Physical Files

A logical file is a file variable declared in a Pascal program. A physical file is a file which exists in the environment outside the program, and is controlled by the operating system.

Example:

Logical files

```
VAR {listing, data, and payroll are logical files}
    listing: TEXT;
    data: FILE OF REAL;
    payroll: FILE OF
        RECORD
            name: STRING[50];
            salary: REAL;
        END;
```

Physical files

'test.dat'	Disc file in the file system
'DATA::50'	Disc file on a FMGR cartridge
'6'	Printer
'1'	User's terminal
'0'	Bit bucket

During program execution, an association is made between a logical and a physical file, so that within the program, any operation performed on the logical file will cause the appropriate action to be performed on the physical file (see below).

A logical file can be used to access many different physical files, although the logical file can only be associated with one physical file at a time. Likewise, a physical file may be accessed by many different logical files in the same, or different programs. This, too, is usually done as one association at a time, although sharing physical files is possible.

When a program is ported to another system, the physical file names may need to change to a format acceptable to the new operating system. Logical file names in the Pascal program, however, need not change.

As described in the previous sections, there are three kinds of logical files: text, sequential, and direct-access files.

- In Pascal/1000, there are two kinds of physical files (devices and disc files), and there are several types of files that can be created, based on the kind and component type of the logical file (see below).

A logical file is associated with a physical file when the logical file is opened. The physical file selected depends on several things:

- Whether the physical file name parameter in the call to the file-opening routine was specified, and was a non-blank, non-null string or PAC expression.
- Whether the logical file is currently associated with a physical file.

- Whether the logical file is a program parameter (its name appeared in the file list in the program heading).

When opened, a logical file will be associated with one of the following kinds of physical files:

- Specified file name
- File with which it is currently associated
- File name in the run string
- Scratch file

The physical file name is chosen in one of the following ways. The numerals to the left of the rules are referred to in the following example program.

- 1) If the physical file name is *explicitly given* in the call to the file-opening routine, and the name is not null, and not all blanks, then this name is used.

If the physical file name is *not given*, or it is the null string, a blank character literal, or an all-blank PAC or string, then one of the following is done:

- 2) If the logical file is *currently associated* with a physical file, then that file is re-used.

If the logical file is *not currently associated* with any physical file, then one of the following is done:

If the logical file is *the nth program parameter*, then one of the following is done:

- 3) If the nth actual parameter in the program's run string was specified when the program was run or when the program was rescheduled and Pas.GetNewParms was called, then this name is used.

If the actual parameter is *missing* from the run string, then one of the following is done:

- 4) If the logical file is the predefined file input or output, then Logical Unit 1, the user's terminal, is used as a default.

- 5) If the file is not input or output, an error occurs.

If the logical file is *not a program parameter*, then one of the following is done:

- 6) If the file-open routine is *rewrite, open, or append*, a scratch file is created. Its name is determined by Pascal.

- 7) If the file-open routine is *rest*, then an error occurs.

Each of these cases is demonstrated in the example below.

Example:

This program shows the different ways logical and physical files can be associated. The file name chosen, and the rule used (from previous page) are shown to the right of the statement.

Also, for this example, assume the program has been scheduled with the run string:

```
C1> files address.txt files.tmp names.txt

PROGRAM files (ppf1, ppf2, input, ppf3, output);

VAR
  ppf1: TEXT;
  ppf2: TEXT;
  ppf3: FILE OF INTEGER;

  gf1: TEXT;
  gf2: FILE OF INTEGER;
  gf3: TEXT;

  s: STRING [20];

{Rule      File}
{No.       Name}

BEGIN
  rewrite (output);          {4:      '1'           }
  reset (input);            {3:      'names.txt'    }
  reset (ppf1);             {3:      'address.txt'  }

  rewrite (ppf2);           {3:      'files.tmp'    }
  writeln (ppf2, 'Testing'); {2:      'files.tmp'    }
  reset (ppf2);             {3:      'files.tmp'    }
  readln (ppf2, s);         {s='Testing'      }

  rewrite (ppf2, 'more.tmp'); {1:      'more.tmp'     }
  writeln (ppf2, 'Again');   {2:      'more.tmp'     }
  close (ppf2);              {3:      'files.tmp'    }
  reset (ppf2);             {3:      'files.tmp'    }
  readln (ppf2, s);         {s='Testing'      }

  rewrite (gf1);             {6:      scratch file name }
  rewrite (ppf3);            {5:      (error)        }
  reset (gf2);               {7:      (error)        }

END.
```

Opening Files

Pascal files must be opened before they can be accessed. HP Pascal provides four predefined procedures for opening files. The procedure used to open a file determines the access method of the file, and the operations allowed on the file.

A file opening operation is of the form:

```
<procedure name> (<logical file>, <physical file>, <options> )
```

<procedure name> is one of

reset	(open for sequential reading)
rewrite	(open for sequential writing)
append	(open for sequential writing, positioned past end-of-file)
open	(open for direct-access reading and writing)

<logical file> is the Pascal file variable.

<physical file> is the name of the file or device in the external RTE environment which is to be associated with the logical file. This parameter is optional, and must be a string literal, a string, or a PAC. (There are other ways to associate files. See *Associating Logical and Physical Files*.)

<options> is the string containing the various open options for the file. This parameter is optional, and must be a string literal, a string, or a PAC.

Examples:

```
reset <input>;  
rewrite <output>, 'test.dat';  
open <data_base>, 'personnel', 'SHARED';  
append <report>, '', 'NOCCTL';
```

In Pascal/1000 the <physical file> string may represent a disc file or peripheral device. A disc file may be either on a FMGR cartridge, or in the File System. The following are sample <physical file> strings:

'test.dat'	Disc file in the File System
'DATA::50'	Disc file on a FMGR cartridge
'6'	Printer
'1'	User's terminal
'0'	Bit bucket

In Pascal/1000 the following words are recognized in <options>:

'CCTL'	specifies that the file has carriage control. (Text files only.)
'NOCCTL'	specifies that the file has no carriage control. (Text files only.)
'SHARED'	specifies that the file may be open to more than one program, or to the current program more than once.
'EXCLUS'	specifies that the file may be open only to one program at a time, and only once to the current program.

Options may be in upper, lower or mixed case, and may be preceded or followed by blanks. Unrecognized options are ignored without error.

Defaults are:

	Non-text Files:	Text Files:
for reset:	'EXCLUS'	'EXCLUS, NOCCTL'
for rewrite:	'EXCLUS'	'EXCLUS, CCTL'
for append:	'EXCLUS'	'EXCLUS, CCTL'
for open:	'EXCLUS'	(not allowed)

If a file is already opened, an implicit close is done before the re-open. This will not destroy the association between the logical and physical files, as is the case on an explicit call to close.

Example:

```

PROGRAM p;

VAR
  f,g: TEXT;
  s: STRING [20];

BEGIN
  rewrite (f, 'FILE1');           {Open f, associate it with FILE1    }
  writeln (f, 'Hello');          {Write a line to it                  }
  reset (f);                     {Open f for reading, preserving the}
  readln (f,s);                 {file association                   }

  rewrite (g);                  {Open g to a scratch file           }
  writeln (g, 'Hello');          {Write a line to it                  }
  close (g);                    {Close it (destroys association)   }
  reset (g);                    {Error occurs since there is no    }
  readln (f, s);                {file name available               }

END;

```

More information on each of the file-opening routines can be found later in this chapter in the I/O Standard Procedures and Functions section.

Pascal/1000 Note:

In HP Pascal, <physical file> must be specified if <options> is specified. In Pascal/1000 (only at STANDARD_LEVEL 'hp1000') the <physical file> may be omitted, even if <options> is specified. Thus, the following is allowed

```
append (report,,'NOCCTL');
```

This feature is currently allowed for compatibility with previous compilers. Its use is not portable and is discouraged.

I/O Standard Procedures and Functions

This section describes the standard procedures and functions that are used for input and output operations, and opening and closing files.

In the following pages, each routine is described in alphabetical order with the format:

- Name of the routine
- Sample usage
- Parameter description
- Return value description, if the routine is a function
- Description of the routine
- Errors that may be encountered when calling the routine
- Limitations
- Example and/or illustration

A list of the I/O routines is shown in Table 6-1.

Table 6-1.
Input/Output Standard Procedures and Functions

append	open a file for extension.
close	close a file.
eof	determine end-of-file status.
eoln	determine end-of-line status.
get	read the next file component into the buffer variable.
lastpos	determine the index of the last component in a file.
linepos	determine the position in the current line of a text file.
maxpos	determine the maximum number of components in a file.
open	open a file for direct access.
overprint	overprint the current line with the next text file line.
page	page eject between two lines of a text file.
position	determine the current position in a direct-access file.
prompt	leave the cursor at the end of a line.
put	write the buffer variable to the next file component.
read	read one or more values from a file.
readdir	read one or more values from a direct-access file.
readln	read values from a text file, and advance to the next line.
reset	open a file for reading.
rewrite	open a file for writing.
seek	position a direct-access at a particular component.
write	write one or more values to a file.
writedir	write one or more values to a direct-access file.
writeln	write values and a line marker to a text file.

Append

Usage

```
append (f)
append (f, p)
append (f, p, o)
```

Parameters

- f Logical file: A file variable. f may be a TEXT or non-TEXT file.
- p Physical file: A string or PAC expression containing the name of the physical file to be associated with f.
- o Options: A string or PAC expression containing a list of file open options.

(The parameters p and o are described further in the Section called Opening Files.)

Description

Append is used to extend a file, i.e. write more records at the end of an existing file.

A call to append will do the following:

If f is currently associated with a physical file, that physical file is closed.

If the physical file associated with f currently exists, that file is opened for writing with sequential access, and positioned just beyond the end of the file.

If the physical file associated with f does not exist, the file is created, opened for writing with sequential access, and positioned at the beginning of the file.

(Further semantics of file association are discussed in the Section called Associating Logical and Physical Files)

After a call to append, the following are true:

f is in the write-only state.

f is positioned immediately past the last component in the file.

Eof (f) returns true.

The contents of f are undefined.

Any previous contents of the physical file are left unchanged.

Files

As long as the file is open, the following procedures and functions may be used with the file f:

write	Write data to the file.
put	
writeln	Write data to the file, if f is a text file.
prompt	
overprint	
page	
eof	Determine end-of-file status of f.
linepos	Determine current position in the current line, if f is a text file.
close	Close the file.

Errors

An error occurs if:

p contains an invalid file name.

f is a file in the program parameter list, and p is the default (null or blank) file name, and the corresponding file name in the run string is missing.

The program is denied access to p.

p is a Type 1 or Type 2 file (a file previously created for direct access).

p is a scratch file, and no scratch file name is available.

Illustration

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

```
{initial condition}

| P     A     S                         state: closed

append(examp_file);

| P     A     S                         current position
                                         ↓
                                         state: write-only
                                         examp_file*: undefined
                                         eof(examp_file): true
```

Close

Usage

```
close (f)
close (f, o)
```

Parameters

- f Logical file: A file variable. f may be a TEXT or a non-TEXT file.
- o Options: A string or PAC expression containing a file close option. Options recognized for Pascal/1000 are:
 - 'SAVE' Keep the file and its contents intact after closing.
 - 'PURGE' Purge the file after closing. (Ineffective if file was open 'SHARED')

The default option is:

'SAVE'	for non-scratch files
'PURGE'	for scratch files

Options may be in upper, lower or mixed case, and may be preceded or followed by blanks. Unrecognized options are ignored.

Description

Close is used to close a file, prohibiting further access to it.

A call to close will do the following:

If the file is currently open, any buffered data will be posted to the physical file, and the file will be closed, according to the default or specified options.

If the file is not currently open, no further action is taken (and no error occurs).

After a call to close the following are true:

The file is in the closed state.

The file may be reopened with reset, rewrite, open, or append.

Any other operation on the file, including references to f* and calls to eof, will cause an error.

Any association between the logical and physical files is destroyed.

Errors

An error occurs if:

The file system is unable to close f.

Eof

Usage

```
eof  
eof (f)
```

Parameters

- f Logical file:A file variable. f may be a TEXT or non-TEXT file. f is optional. If f is omitted, the predefined file input is assumed.

Return Value

A Boolean value of true (if f is at end-of-file), or false (if f is not at end-of-file).

Description

Eof is used to determine the end-of-file status of a file.

Eof returns a value of true if one of the following is true. It returns a value of false otherwise.

f is in the write-only state.

f is in the read-write state, and position (f) > lastpos (f) (i.e. the current position is greater than the highest-indexed component written to f)

f is in the read-only state, and the current position of f is beyond the last component of f.

Errors

An error occurs if f is not open.

Eoln

Usage

```
eoln  
eoln (f)
```

Parameters

f Logical file: A file variable. f must be a TEXT file. f is optional. If f is omitted, the predefined file input is assumed.

Return Value

A Boolean value of true (if f is at end-of-line), or false (if f is not at end-of-line).

Description

Eoln is used to determine the end-of-line status of a text file which has been opened in the read-only state.

Eoln returns a value of true if the current position of f is at an end-of-line marker.

The function references the buffer variable `f^`, which may cause an input operation to occur. For example, after a call to `readln`, a call to `eoln` will place the first character of the new line in the buffer variable.

Errors

An error occurs if:

- f is not a text file.
- f is not opened in the read-only state.
- `eof (f)` is true.

Get

Usage

```
get  
get (f)
```

Parameters

- f Logical file: A file variable. f may be a TEXT or a non-TEXT file. f is optional. If f is omitted, the predefined file input is assumed.

Description

Get is used to read the next component from a physical file into the buffer variable of a logical file which have been open in either the read-only state or the read-write state.

A call to get will do the following:

Advance the current position to the next component in the file.

If there is a component at this new position, then

The component is assigned to the buffer variable f*.

If there is no component at this new position, then

The contents of the buffer variable f* are undefined, and eof (f) will return a value of true.

Errors

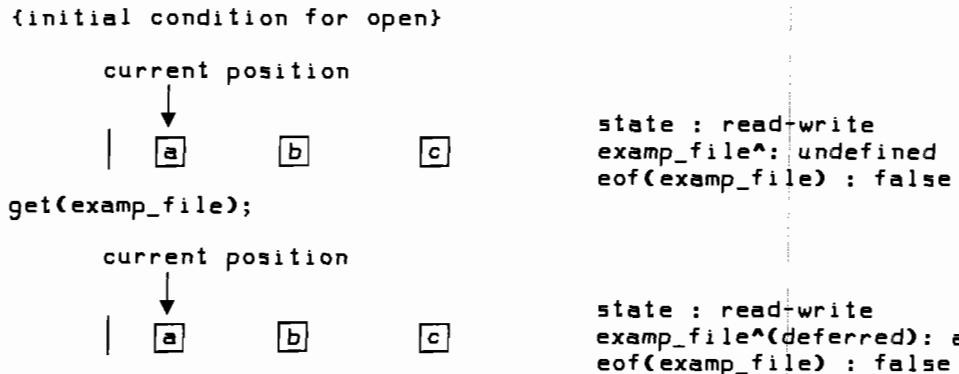
An error occurs if one of the following is true:

The file f is not opened either in the read-only state or the read-write state.

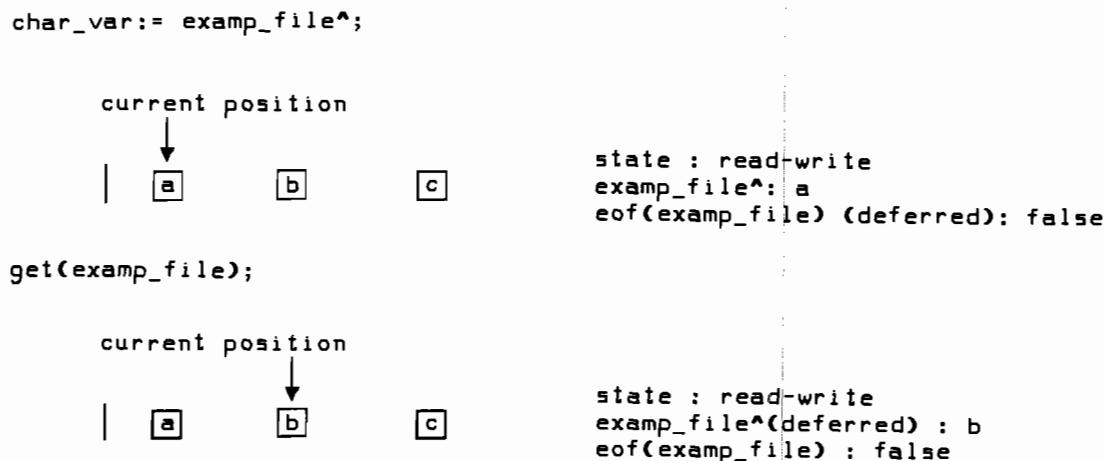
eof (f) was true before the call to get.

Illustration

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



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



Lastpos

Usage

```
lastpos (f)
```

Parameters

f Logical file: A file variable. f may not be a TEXT file.

Return Value

An integer value representing the last component written to a direct-access file.

Description

Lastpos is used to determine the index of the last (highest-indexed) component that has been written to the direct-access file f. If this value cannot be determined from information available from the file system, then lastpos will return the same value as maxpos.

Errors

An error occurs if one of the following is true:

f is not opened in the read-write state.

f is a text file

Limitations

In the Pascal/1000 system, lastpos will always return the same value as maxpos, since the index of the last component cannot be determined from the underlying file system.

Linepos

Usage

```
linepos (f)
```

Parameters

f Logical file: A file variable. f must be a TEXT file.

Return Value

An integer value containing the number of characters processed so far on the current line.

Description

Linepos is used to determine the number of characters read from or written to the current line in a text file. This count is one of the following quantities:

The number of characters read from f since the last call to reset or readln.

The number of characters read from f since the last call to read or get which moved the file's position past an end-of-line marker.

The number of characters written to f since the last call to rewrite, writeln, page, prompt, or overprint.

Zero if f is open in the read-only state, and eof (f) is true.

Errors

An error occurs if one of the following is true:

f is not a text file.

f is not open.

Maxpos

Usage

```
maxpos (f)
```

Parameters

f Logical file: A file variable. f may not be a text file.

Return Value

An integer containing the maximum number of possible components in f.

Description

Maxpos is used to determine the maximum number of components that the direct-access file f may ever contain.

The number returned represents the logical limits of the file, not the physical limits. That is, if maxpos (f) returns n, there is no guarantee that n components may actually be written to f. Nor is the existence of the nth component a guarantee that there is room to write all of the other components of the file.

Errors

An error occurs if one of the following is true:

f is not opened in the read-write state.

f is a text file.

Open

Usage

```
open (f)
open (f, p)
open (f, p, o)
```

Parameters

- f Logical file: A file variable. f may not be of type TEXT.
- p Physical file: A string or PAC expression containing the name of the physical file to be associated with f.
- o Options: A string or PAC expression containing a list of file open options.

The parameter p and o are described further in Section 6.4 on Opening Files.

Description

Open is used to open a file for direct access. Any component in the file may be directly read from or written to.

A call to open will do the following:

If f is currently associated with a physical file, that physical file is closed.

If the physical file associated with f currently exists, that file is opened.

If the physical file associated with f does not exist, the file is created.

The file is opened for direct access and positioned at the beginning.

(Further semantics of file association are discussed in the Section called Associating Logical and Physical Files.)

After a call to open, the following are true:

f is in the read-write state.

f is positioned at the beginning of the file.

Eof (f) returns false if the file is not empty.

Eof (f) returns true if the file is empty.

The contents of f* are undefined.

Any previous contents of the physical file are left unchanged.

Files

As long as the file is open, the following procedures and functions may be used with the file f:

- seek Set the current position of the file.
- write Write data to the file.
- writedir
- put
- read Read data from the file.
- readdir
- get
- position Determine current position of the file.
- maxpos Determine the maximum number of components that may be written to the file.
position (f) <= maxpos (f)
- lastpos Determine the index of the last (highest-indexed) component that has been written to the file. lastpos (f) <= maxpos (f)
- eof Determine end-of-file status of f. Returns true if the current position is greater than the last component. eof (f) = (position (f) > lastpos (f)).
- close Close the file.

Errors

An error occurs if:

- p contains an invalid file name.
- f is a file in the program parameter list and p is the default (null or blank) file name, and the corresponding file name in the run string is missing.

The program is denied access to p.

- p already exists, and is not a direct-access (Type 1 or Type 2) file.
- p is a scratch file, and no scratch file name is available.

Illustration

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

```
open(examp_file);
          current position
| 16      10      25  state: read-write
                      examp_file*: undefined
                      eof(examp_file): false
```

Overprint

Usage

```
overprint
overprint (p1, ..., pn)
overprint (f)
overprint (f, p1, ..., pn)
```

Parameters

f Logical file: A file variable. **f** must be a text file. **f** is optional. If **f** is omitted, the predefined file output is assumed.

p1 thru pn Write parameters: The expressions (and format specifications) whose values are to be written to the file **f**, followed by a special end-of-line marker.

p1 thru pn are optional. If omitted, just the end-of-line marker is written.

(See the description of write for more information on **p1 thru pn**).

Description

Overprint is used to mark the end of a line of output (line1) in a text file with a special end-of-line marker, and to begin a new line (line2). When the text file is printed, the special marker will cause line2 to be printed over line1. (After line1 is printed, a carriage return is performed with no line feed.)

The statement

```
overprint (f, p1, ..., pn)
```

is equivalent to the two statements

```
write (f, p1, ..., pn);
overprint (f);
```

A call to overprint will do the following:

If **p1 thru pn** are specified, they will be formatted and written to the file **f** as specified in the description of the procedure write.

A special end-of-line marker is written to the file which causes the following line to overprint the current line.

The current position of **f** is moved to just beyond this marker.

After a call to overprint, the following are true:

Eoln (**f**) is false.

Eof (**f**) is true.

The contents of **f** are undefined.

Linepos (**f**) is 0.

Files

Errors

An error occurs if one of the following is true:

f is not a text file.

f is not opened in the write-only state.

f is not opened with carriage control specified ('CCTL').

the field width specified in one of the write parameters is negative.

Limitations

In Pascal/1000 programs compiled with the HEAP 2 option, p1 thru pn may not be heap variables whose size is larger than 1024 words.

Page

Usage

```
page  
page (f)
```

Parameters

- f Logical file: A file variable. f must be a text file. f is optional. If f is omitted, the predefined file output is assumed.

Description

Page is used to cause a page eject operation between two lines of a text file.

A call to page will do the following:

A special character is written to the file which causes a page eject when the file is printed.

The current position of f is moved to just beyond this marker.

After a call to page, the following are true:

Eoln (f) is false.

Eof (f) is true.

The contents of f are undefined.

linepos (f) is 0.

Errors

An error occurs if one of the following is true:

f is not a text file.

f is not opened in the write-only state.

f is not opened with carriage control specified ('CCTL').

Position

Usage

```
position (f)
```

Parameters

f Logical file: A file variable. f may not be a text file.

Return Value

An integer containing the current position of the file f.

Description

Position is used to determine the index of the current component of the direct-access file f. This component is the one which will be affected by the next read or write operation. If the buffer variable is full, the result is the index of the component in the buffer.

The following are always true:

```
1 <= position (f) <= maxpos (f)  
1 <= lastpos (f) <= maxpos (f)  
eof (f) = (position (f) > lastpos (f))
```

Errors

An error occurs if one of the following is true:

- f is not opened in the read-write state.
- f is a text file.

Prompt

Usage

```

prompt
prompt (p1, ..., pn)
prompt (f)
prompt (f, p1, ..., pn)
```

Parameters

f Logical file: A file variable. f must be a text file. f is optional. If f is omitted, the predefined file output is assumed.

p1 thru pn Write parameters: The expressions (and format specifications) whose values are to be written to the file f, followed by a special end-of-line marker.

p1 thru pn are optional. If omitted, just the special marker is written.

(See the description of write for more information on p1 thru pn.)

Description

Prompt is used to mark the end of a line of output in a text file with a special end-of-line marker which, when output to an interactive device, will leave the cursor at the end of the line. The statement

```
prompt (f, p1, ..., pn)
```

is equivalent to the two statements

```

write (f, p1, ..., pn);
prompt (f)
```

A call to prompt will do the following:

If p1 thru pn are specified, they will be formatted and written to the file f as specified in the description of the procedure write.

A special end-of-line marker is written to the file which leaves the cursor at the end of the line. (No carriage return nor line feed operation takes place.)

The current position of f is moved to just beyond this marker.

After a call to prompt, the following are true:

Eoln (f) is false.

Eof (f) is true.

The contents of f are undefined.

Linepos (f) is 0.

Errors

An error occurs if one of the following is true:

- f is not a text file.
- f is not opened in the write-only state.
- f is not opened with carriage control specified ('CCTL').
- The field width specified in one of the write parameters is negative.

Limitations

In Pascal/1000 programs compiled with the HEAP 2 option, p1 through pn may not be heap variables whose size is larger than 1024 words.

Put

Usage

```
put  
put (f)
```

Parameters

- f Logical file: A file variable. f may be a TEXT or a non-TEXT file. f is optional. If f is omitted, the predefined file output is assumed.

Description

Put is used to write the buffer variable f^* into the current component in the physical file. f is a file which has been opened in either the write-only state or the read-write state.

A call to put will do the following:

Write the contents of the buffer variable f^* to the current component of f.

Advance the current position to the next component in the file.

After a call to put, the following are true:

Eof (f) is true.

The contents of the buffer variable f^* are undefined.

If f is a text file, then

Eoln (f) is false.

Linepos (f) after a call to put is one greater than linepos (f) before a call to put.

Errors

An error will occur if one of the following is true:

The file f is not opened in either the write-only or the read-write state.

Illustration

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

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

current position

1

state: write-only
`examp_file^`: 9
`eof(examp_file)`: true

```
put(examp_file);
```

current position

1

9

state: write-only
`examp_file^`: undefined
`eof(examp_file)`: true

Read

Usage

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

Parameters

f Logical file: A file variable. f may be a text or non-text file. f is optional. If f is omitted, the predefined file input is assumed.

vi thru vn Read variables: Variables which are to be assigned new values based on the contents of the file f. The parameters may be components of a packed structure.

If f is a non-text file, the component type of f must be assignment compatible with the type of each read variable.

If f is a text file, then v1 through vn are variables whose type is integer, integer subrange, real, longreal, Boolean, char, PAC, string, or an enumerated type. Conversions from ASCII to internal form are performed during the read operation.

Description

Read is used to read one or more values from an input file into the same number of variables.

A call to read will do the following:

A series of read operations will occur, one for each v in the parameter list.

If f is a non-text file, then

The value of the current component of f is assigned to v.

The current position of f is advanced by one.

Any subsequent reference to f^ will load f^ with the new current component.

If f is a text file, then

A sequence of characters is read from f and converted from ASCII to the internal form determined by the type of v (see Implicit Data Conversions below)

The current position of f is advanced by the number of characters used in the conversion operation.

Any subsequent reference to f^ will load f^ with the new current character from f.

```
read (f, v1, v2, ..., vn) is equivalent to read (f, v1);
                                read (f, v2);
```

...

```
                                read (f, vn)
```

```
read (f, v)      is equivalent to v := f^;
                  get (f)
```

If f is a non-text file or f is a text file and v is of type CHAR.

Implicit Data Conversions During a Read Operation

If f is a text file, its components are of type char. The read parameters v1 thru vn, however, may have other types. An implicit conversion is performed from the ASCII characters in the text to the internal form of the type of the read variable.

The type of each parameter may be an integer, integer subrange, real, longreal, char, PAC, string, Boolean, or an enumerated type. The conversions are performed as follows:

Integer

An integer is read by converting a character sequence which satisfies the syntax for integer literals. Preceding end-of-line markers and preceding blanks are skipped.

Real and Longreal

A real (or longreal number) is read by converting a character sequence which satisfies the syntax for real (or longreal) literals. A character sequence representing a real literal may be read into a longreal variable. Preceding end-of-line markers and preceding blanks are skipped.

Char

A character variable v is read by assigning the next character from the current line to v. If eoln (f) is true before the character is read, then v is assigned a blank, and the current position of f is moved to the first character of the next line.

PAC

A PAC variable v of length n is read by assigning the next n characters on the input line to v. If there are only m ($m < n$) characters on the line, then those m characters are assigned to the first m characters of v, and the rest of v is blank-filled.

String

A string variable v of maximum length n is read by assigning the next n characters on the input line to v, and the current length of v is set to n, i.e. strlen (v) = n.

If there are only m ($m < n$) characters on the line, then those m characters are assigned to the first m characters of v, and the current length of v is set to m, i.e. strlen (v) = m.

Enumerated

A variable v of an enumerated type is read by converting a sequence of characters that satisfies the syntax for an HP Pascal identifier, and matches one of the identifiers listed in the type definition for v. The appropriate value is then assigned to v. The letters making up the identifier may be in upper, lower, or mixed case. Preceding end-of-line markers and preceding blanks are skipped.

Boolean

A Boolean variable v is read as if it were any other enumerated type. That is, if file f has one line containing the characters:

 FALSE TRUE

then read (f, v1, v2), where v1 and v2 and Boolean, will assign false to v1 and true to v2.

Examples

The following table shows some examples of the results of calls to read with various sequences of characters for different types of v.

Sequence of characters in f following current position	Type of v	Result stored in v
(space)(space)1.850	real	1.850
(space)(end-of-line)(space)1.850	longreal	1.850
10000(space)10	integer	10000
8135(end-of-line)	integer	8135
54(end-of-line)36	integer	54
1.583E3	real	1583
1.583E+3	longreal	1583
(space)Pascal	string[5]	' Pasc'
(space)Pas(end-of-line)cal	string[9]	' Pas'
(space)Pas(end-of-line)cal	PAC, length=9	' Pas '
(space)Monday(space)	(Monday, Tuesday)	Monday

Errors

An error occurs if one of the following are true:

f is not opened either in the read-only state or the read-write state.

eof (f) was true before the call to read.

If f is a text file, then an error occurs if one of the following is true:

The read operation finds no non-blank characters (when reading variables whose type is other than CHAR).

A faulty sequence of characters is read for an integer, real, longreal, or enumerated value.

An integer is read which is outside the range allowed for the corresponding read variable.

An identifier is read for a variable of an enumerated type, and that identifier does not match any of the identifiers in the corresponding type definition.

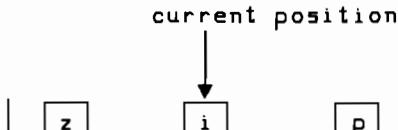
Limitations

In Pascal/1000 programs compiled with the HEAP 2 option, v1 thru vn may not be heap variables whose size is larger than 1024 words.

Illustration

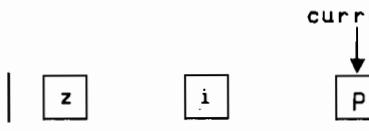
Suppose examp_file is a file of char opened in the read-only state. The current position is at the second component. To read the value of this component into char_var, we call read:

{initial condition}



state: read-only
examp_file^: i or undefined
eof(examp_file): false
char_var: old value. if any

`read(examp_file,char_var);`



state: read-only
examp_file^(deferred): p
eof(examp_file): false
char_var: i

Readdir

Usage

```
readdir (f, p, v1, ..., vn)
```

Parameters

f	Logical file: A file variable. f must be a non-text file.
p	Position: The index of the component of f that is to be read into v1. 1 <= p <= lastpos (f).
v1 thru vn	Read variables: Variables which are to be assigned the values in the components of f, starting at position p. The component type of f is assignment compatible with each variable in the list.

Description

Readdir is used to read one or more values from a direct-access file into a number of variables. Text files may not be used with readdir.

A call to readdir will do the following:

The current position of f is placed at component p.

For each variable v in the parameter list, a read operation is performed which does the following:

The value of the current component is assigned to v.

The current position of f is advanced by one.

Any subsequent reference to f^ will load f^ with the current component.

```
readdir (f, p, v1, v2, ..., vn) is equivalent to seek (f, p);
                                         read (f, v1);
                                         read (f, v2);
                                         ...
                                         read (f, vn)
```

Errors

An error occurs if:

f is not opened in the read-write state.

p is less than 1, or greater than lastpos (f).

Limitations

In Pascal/1000, programs compiled with the HEAP 2 option v1 thru vn may not be heap variables whose size is larger than 1024 words.

Illustration

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

```
{initial condition}
```

current position



```
state: read-write
examp_file^: undefined
eof(examp_file): false
int_var: old value
```

```
readdir(examp_file,3,int_var);
```

current position



```
state: read-write
examp_file^(deferred):10
eof(examp_file): false
int_var: 40
```

Readln

Usage

```
readln
readln (p1, ..., pn)
readln (f)
readln (f, p1, ..., pn)
```

Parameters

- f Logical file: A file variable. f must be a text file. f is optional. If f is omitted, the predefined file input is assumed.
- vi thru vn Read variables: Variables which are to be assigned new values based on the contents of the file f. Each variable may be of type integer, real, longreal, char, PAC, string, enumerated, or Boolean.
- v1 thru vn are optional.

Description

Readln is used to read one or more values from an input text file into a number of variables, and then advance to the beginning of the next line.

A call to readln will do the following:

A series of read operations will occur, one for each variable v in the parameter list, as specified in the description for the procedure read.

The current position of f is advanced to just beyond the next end-of-line marker. This will either be the position of the first character of the next line, or it will be the position of the end-of-file marker.

After a call to readln, the following are true:

eoln (f) is false.

linepos (f) is 0.

If the readln operation moved the current position past the last end-of-line marker in the file, then

eof (f) is true.

The contents of f^{*} are undefined.

If the readln operation moves the current position to the beginning of a new line, then

eof (f) is false.

A reference to f^{*} will load f^{*} with the first character of the new line.

```
readln (f, p1, ..., pn) is equivalent to read (f, p1, ..., pn)
                           readln (f)
```

Errors

An error occurs if one of the following is true:

f is not a text file.

f is not opened in the read-only state.

eof (f) was true before the call to readln.

Any of the text file errors for the procedure read (see Read) occurred.

Limitations

In Pascal/1000 programs compiled with the HEAP 2 option, p1 thru pn may not be heap variables whose size is larger than 1024 words.

Reset

Usage

```
reset (f)
reset (f, p)
reset (f, p, o)
```

Parameters

- f Logical file: A file variable. f may be a TEXT or non-TEXT file. f is optional. If omitted, the predefined file input is assumed.
- p Physical file: A string or PAC expression containing the name of the physical file to be associated with f.
- o Options: A string or PAC expression containing a list of file open options.

(The parameters p and o are described further in the Section called Opening Files.)

Description

Reset is used to open a file for reading.

A call to reset does the following:

If f is currently associated with a physical file, that physical file is closed.

If the physical file associated with f currently exists, that file is opened for reading with sequential access, and is positioned at the beginning of the file.

If the physical file associated with f does not exist, an error will occur.

(Further semantics of file association are discussed in the Section called Associating Logical and Physical Files.)

After a call to reset, the following are true:

f is the read-only state.

f is positioned at the beginning of the file.

If the file is not empty, then

Eof (f) returns false.

A reference to f^ will return the first component of the file.

Files

If the file is empty, then

Eof (f) returns true.
The contents of f* are undefined.

Any previous contents of the physical file are left unchanged.

As long as the file is open, the following procedures and functions may be used with the file f:

read	Read data from the file.
get	
readln	Read data from the file, if f is a text file.
eof	Determine end-of-file status of f.
eoln	Determine end-of-line status of f, if f is a text file.
linepos	Determine the number of characters that have been read from f, if f is a text file.
close	Close the file.

Errors

An error occurs if:

p contains an invalid file name.

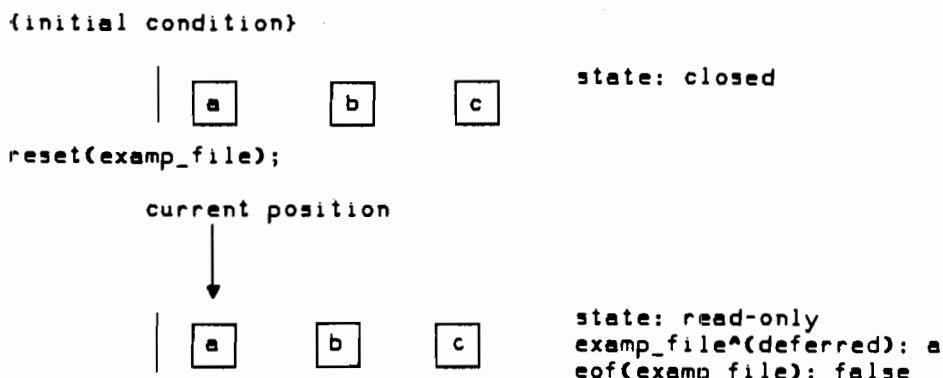
f is a file in the program parameter list, and p is the default (null or blank) file name, and the corresponding file name in the run string is missing.

The program is denied access to p.

p is '0'; Logical unit 0 cannot be opened for read-only.

Illustration

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



Rewrite

Usage

```
rewrite (f)
rewrite (f, p)
rewrite (f, p, o)
```

Parameters

- f** Logical file: A file variable. f may be a TEXT or non-TEXT file. f is optional. If f is omitted, the predefined file output is assumed.
- p** Physical file: A string or PAC expression containing the name of the physical file to be associated with f.
- o** Options: A string or PAC expression containing a list of file open options.

(The parameters p and o are described further in the Section called Opening Files)

Description

Rewrite is used to open a file for writing.

A call to rewrite will do the following:

If f is currently associated with a physical file, that physical file is closed.

If the physical file associated with f currently exists, that file is opened.

If the physical file associated with f does not exist, the file is created.

The file is opened for writing with sequential access, and positioned at the beginning of the file.

(Further semantics of file association are discussed in the Section called Associating Logical and Physical Files.)

After a call to rewrite, the following are true:

f is in the write-only state.

f is positioned at the beginning of the file.

Eof(f) returns true.

The contents of f* are undefined.

Any previous contents of the physical file are lost.

Files

As long as the file is open, the following procedures and functions may be used with the file f:

write	Write data to the file.
put	
writeln	Write data to the file, if f is a text file.
prompt	
overprint	
page	
eof	Determine end-of-file status of f.
linepos	Determine current position in the current line, if f is a text file.
close	Close the file.

Errors

An error occurs if:

- p contains an invalid file name.
- f is a file in the program parameter list, and p is the default (null or blank) file name, and the corresponding file name in the run string is missing.

The program is denied access to p.

p is a scratch file, and no scratch file name is available.

Illustration

Suppose examp_file is a closed file of char with three components. To discard these components and write sequentially to examp_file, we call rewrite:

```
{initial condition}
|   a     b     c           state: closed
|   ↓
rewrite(examp_file);
      current position
|   ↓
                                state: write-only
                                examp_file^ : undefined
                                eof(examp_file): true
```

Seek

Usage

`seek (f, p)`



Parameters

`f` Logical file: A file variable. `f` must be a non-text file.

`p` Position: The integer index of a component of `f`. $1 \leq p \leq \text{maxpos} (f)$

Description

Seek is used to position a direct-access file at a particular component. The position `p` may be any number between 1 and `maxpos (f)`. (Note: No check is made by Pascal to ensure that `p` is positive. A non-positive `p` is sometimes meaningful, and is dependent on the semantics of the underlying file system. For example, for FMGR files, a negative `p` can mean "backspace `p` records".)

A call to seek will do the following:

The current position of `f` is moved to component `p`.

After a call to seek, the following are true:

The contents of `f^` are undefined.

If $p \leq \text{lastpos} (f)$, then
`eof (f)` will return false.

A `read (f, v)` or `write (f, e)` call can be made.

If $p > \text{lastpos} (f)$, then
`eof (f)` will return true.
A `write (f, e)` call can be made.

If $p > \text{maxpos} (f)$, then
`eof (f)` will return true.

Errors

An error occurs if:

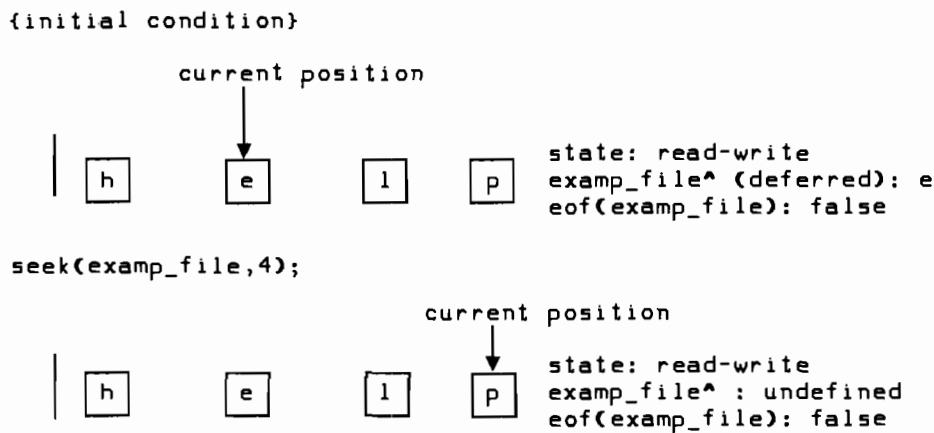
`A read or write is done after a seek (f,p), with $p > \text{maxpos} (f)$.`

`f is not opened in the read-write state.`

`f is a text file.`

Illustration

Suppose `examp_file` is a file of `char` with four components opened for direct access. The current position is the second component. To change it to the fourth component, we call `seek`.



Write

Usage

```
write (p1, ..., pn)
write (f, p1, ..., pn)
```

Parameters

f Logical file. A file variable. f may be a text or a non-text file. f is optional. If f is omitted, the predefined file output is assumed.

p1 thru pn Write parameters:

If f is a non-text file, then each parameter p is an expression whose value is to be written to f.

If f is a text file, then each parameter p consists of the subparameters e, w, and d in one of the following forms:

```
e
e:w
e:w:d
```

Subparameters:

e Expression: The char, PAC, string, integer, real, longreal, enumerated or Boolean expression to be written to f.

w Field Width: The integer number of characters to be written when writing e.

d Digits: The integer number of digits that are to appear to the right of the decimal point. (This subparameter is allowed only when e is real or longreal.)

Description

Write is used to write one or more values to a file.

A call to write will do the following:

A series of write operations will occur, one for each parameter p in the parameter list.

If f is a non-text file, then

Each parameter p must be assignment compatible with the component type of f.

The value of p is assigned to the current component of f.

The current position of f is advanced by one.

If f is a text file, then

Each parameter p is in the form above. Each e must be of one of the types char, PAC, string, integer, real, longreal, enumerated, or Boolean.

Each e is converted to an ASCII character sequence, and is then written to f. The character sequence produced depends on the values of the subparameters w and d, if specified. (See *Formatting of Output to Textfiles*, below.)

The current position of f is advanced by the number of characters written to f.

After a call to write, the following are true for both text and non-text files.

Eof (f) is true.

The contents of the buffer variable fⁿ are undefined.

```
write (f, p1, p2, ..., pn) is equivalent to write (f, p1);
                                         write (f, p2);
                                         ...
                                         write (f, pn)
```

Also, if either:

f is a non-text file, or

f is a text file and the parameter p is an expression e of type char, with w unspecified or w=1, then

```
write (f, e)           is equivalent to f^ := e;
                           put (f)
```

Errors

An error occurs if:

f is not opened in either the write-only state or the read-write state.

The current position of f is greater than maxpos, when f is a direct-access file.

In Pascal/1000, a runtime warning occurs if:

The number of characters written causes linepos (f) to be greater than the maximum line size of the text file f. (See LINESIZE compiler option.) In this case,

The warning is printed to the user's terminal.

A line-marker is written to f.

The character sequence is written on the next line.

e is a PAC or string whose length is greater than the line size of the text file f (see LINESIZE compiler option). In this case,

The warning is printed to the user's terminal.

If no characters have been written to the current line, (i.e. linepos (f) is 0), then a line-marker is written to f.

The characters that will fit on the line are written to f.

A line-marker is written to f.

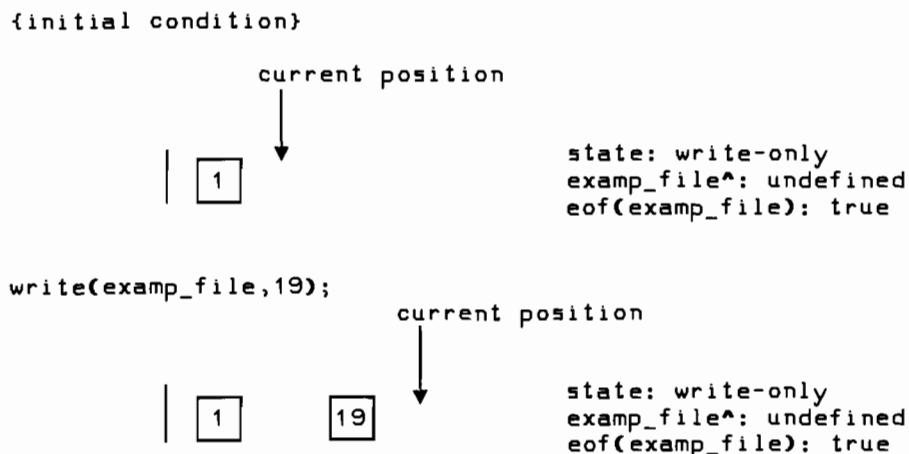
The left-over characters are then written on the next line.

Limitations

In Pascal/1000, programs compiled with the HEAP 2 option p1 through pn may not represent heap variables whose size is larger than 1024 words.

Illustration

Suppose examp_file is a file of integer in the write-only state and that we have written one number to it. To write another number, we call write again:



Formatting of Output to Textfiles

If f is a text file, its components are of type char. The write parameters p1 thru pn, however, may have other types. A format operation (a conversion from internal form to a sequence of ASCII characters) is performed for each parameter in the list. This character sequence is then written into the components of the text file f.

The following abbreviations are used in the descriptions below:

- p Write parameter
- e Expression
- w Field width ($w \geq 0$)
- d Digits past the decimal point to be printed ($d \geq 0$). (allowed only when e is real or longreal)
- c The ASCII character sequence representing e. (with no leading or trailing blanks added)
- n Number of characters in c.
- m Sign character: '-' if $e < 0$, no character if $e \geq 0$.
- s Exponent sign character ('+' or '-')
- x Digit sequence representing the integer part of c.
- y Digit sequence representing the fractional part of c.
- z Digit sequence representing the exponent part of c.
- E The character 'E' for reals, 'L' for longreals.

Default Field Widths for Output Formatting

The following table shows the default values for w.

Type of e	Default field width (w)
integer	12
real	13
longreal	23
enumerated	length of identifier
Boolean	length of identifier
PAC	length of PAC
char	1
string literal	length of string literal
string	current length of string

The type of each parameter expression may be a char, PAC, string, integer, real, longreal, Boolean, or an enumerated type. No conversion is done for expressions of type char. All others are converted as follows:

Integer

An integer expression e is converted to ASCII and written to f. If w is not specified, the default 12 is used.

The following is written to f:

if w > n:	(w-n blanks)(c)	(w characters are written)
if w <= n:	(c)	(n characters are written)

Real and Longreal

A real (or longreal) number is converted to ASCII and written to f. If w is not specified, the default 13 (23 for longreal) is used. If d is not specified, then e is written in floating-point format. If d is specified, then e is written in fixed-point format.

The following is written to f:

if w > n:	(w-n blanks)(c)	(w characters are written)
if w <= n:	(c)	(n characters are written)

Floating-point format (d is not specified, e:w)

The format of c is:

mx.yEsz

Fixed-point format (d is specified, e:w:d)

The format of c is:

if n <> 0:	mx.y
if n = 0:	mx

Exception: c will never contain more significant digits than are actually present in the internal representation of e. Thus if the values of e and d are such that too many significant digits would be printed, then e will be written using the floating-point format instead (as if e:w had been specified).

PAC

A PAC expression is written to f with no conversion. c is the full PAC expression, and n is its declared length. If w is not specified, the default n is used.

The following is written to f:

if w > n	(w-n blanks)(c)	(w characters are written)
if w <= n	(1st w chars of c)	(w characters are written)

Char

A character expression is written to f as if it were a PAC expression with a length of 1 (see above).

String Literal

A string literal expression is written to f as if were a PAC expression (see above) whose length is the number of characters in the literal.

String

A string expression e is written to f with no conversion.

n is the current length of e,

c is the n characters of e.

If w is not specified, the default n is used.

The following is written to f:

if w > n	(w-n blanks)(c)	(w characters are written)
if w <= n	(1st w chars of c)	(w characters are written)

Enumerated

An expression of an enumerated type is converted to the corresponding ASCII identifier, and is then written to f. The characters will appear in upper case. c is the identifier, n is the number of characters in c. If w is not specified, the default n is used.

The following is written to f:

if w > n	(w-n blanks)(c)	(w characters are written)
if w <= n	(1st w chars of c)	(w characters are written)

Boolean

A Boolean expression is written like any other expression of an enumerated type (see above). Thus, if b is a Boolean expression, then output will appear as follows:

write (f, b) 'FALSE' or 'TRUE'

write (f, b:1) 'F' or 'T'

Examples

```

PROGRAM show_formats (output);
VAR
  x: real;
  lr: longreal;
  george: boolean;
  list: (yes, no, maybe);
BEGIN
  writeln(999);           {default formatting}
  writeln(999:1);         {format defeated}
  writeln('abc');
  writeln('abc':2);       {string literal truncated}
  x:= 10.999;
  writeln(x);             {default formatting}
  writeln(x:25);
  writeln(x:25:5);
  writeln(x:25:1);
  writeln(x:25:0);
  lr:= 19.1111;
  writeln(lr);
  george:= true;
  writeln(george);        {default format}
  writeln(george:2);
  list:= maybe;
  write(list);            {default formatting}
END.

```

The output of this program is:

```

999
999
abc
ab
1.099900E+01
      1.0999001E+01
      10.99900
      11.0
      11
1.91110992431641L+001
TRUE
TR
MAYBE

```

Writedir

Usage

```
writedir (f, p, e1, ..., en)
```

Parameters

f Logical file: A file variable. f must be a non-text file.

p Position: The index of the component of f that is to be read into v1.
1 < = p < = maxpos (f)

e1 thru en Write expressions: Expressions which are to be written to the file f. The type of each expression e must be assignment compatible with the component type of f.

Description

Writedir is used to write one or more values to a direct-access file. Text files may not be used with writedir.

A call to writedir does the following:

The current position of f is placed at component p.

For each expression e in the parameter list, a write operation is performed which does the following:

The value of e is assigned to the current component of f.

The current position of f is advanced by one.

The contents of fⁿ become undefined.

```
writedir (f, p, e1, e2, ..., en) is equivalent to seek (f, p);
                                         write (f, e1);
                                         write (f, e2);
                                         ...
                                         write (f, en)
```

Errors

An error occurs if:

f is not opened in the read-write state.

p is less than 1, or greater than maxpos (f).

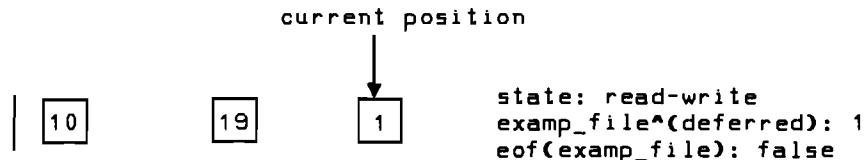
Limitations

In Pascal/1000 programs compiled with the HEAP 2 option, el thru en may not be heap variables whose size is larger than 1024 words.

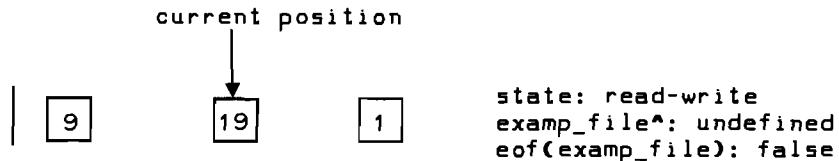
Illustration

Suppose file examp_file is a file of integer opened for direct access. The current position is the third component. To write a number to the first component, we call writedir:

```
{initial condition}
```



```
writedir(examp_file,1,4 + 5);
```



Writeln

Usage

```
writeln
writeln (p1, ..., pn)
writeln (f)
writeln (f, p1, ..., pn)
```

Parameters

f Logical file: A file variable. f must be a text file. f is optional. If f is omitted, the predefined file output is assumed.

p1 thru pn Write parameters: The expressions (and format specifications) whose values are to be written to the file f, followed by an end-of-line marker. p1 thru pn are optional.

(See the description of write for more information on p1 thru pn).

Description

Writeln is used to write data to a line of a text file, and then mark the end of that line.

A call to writeln does the following:

If p1 thru pn are specified, they will be formatted and written to the file f as specified in the description of the procedure write.

An end-of-line marker is written to the file.

The current position of f is moved to just beyond this marker.

After a call to writeln, the following are true:

Eoln (f) is false.

Eof (f) is true.

The contents of f^ are undefined.

Linepos (f) is 0.

```
writeln (f, p1, ..., pn)    is equivalent to write (f, p1, ..., pn);
                                writeln (f)
```

Errors

An error occurs if one of the following is true:

f is not a text file.

f is not opened in the write-only state.

the field width specified in one of the write parameters is negative.

Limitations

In Pascal/1000 programs compiled with the HEAP 2 option, p1 thru pn may not be heap variables whose size is larger than 1024 words.

Chapter 7

Standard Procedures and Functions

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 chapter contains descriptions of the standard dynamic allocation and deallocation procedures. Chapter 8 includes information about Pascal/1000's implementation of heap management, and describes a set of alternative short versions of heap procedures available to the user.

New

Usage

```
new (p);
```

Parameter 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 T = <type definition>; TPTR = ^T;	TYPE T = <type definition>; -OR- VAR P : ^T;
VAR 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 tab 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 specified 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);
```

Parameter 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.

Dispose causes any files in the variable pointed to by the pointer being disposed to be closed.

An error occurs if the value of the pointer parameter of dispose is nil or undefined.

Mark

Usage

```
mark (p);
```

Parameter 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.

Release

Usage

```
release (p);
```

Parameter 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).

Release does not cause files in the heap area being released to be closed.

String Procedures

Setstrlen

Usage

```
setstrlen (s, e)
```

Parameters s is a string variable.
 e is an integer expression. The value of e cannot be less than 0 or greater than the maximum length of s.

Description

The procedure setstrlen (s, e) sets the current length of s. If the new length is greater than the previous length of s, the components which were added to the accessible area of the string have undefined values. No blank filling occurs. If the new length is less than the previous length of s, previously defined components beyond the new length will no longer be accessible. A string to which this procedure has been applied is considered to have been given a value.

Example:

```
VAR
  alpha: STRING [80];

BEGIN
  .
  .
  alpha := 'abcdef'; {Alpha's current length =6}

  {Double the current length of alpha. Alpha [7] through
  {alpha [12] will not be defined. }

  setstrlen (alpha, 2 * strlen (alpha));

  {Make alpha [3] through alpha [80] not accessible.}

  setstrlen (alpha, 2)
  .
  .
END.
```

Strappend

Usage

```
strappend (s1, s2)
```

Parameters s1 is a string variable.
 s2 is a string expression.

Description

The procedure strappend (s1, s2) appends string s2 onto s1. The call passes s1 as an actual variable parameter to the procedure. The resulting length must be less than or equal to the maximum length of s1. The string s1 must have a value prior to appending to it.

Example:

```
VAR
  message: STRING [132]

BEGIN
  .
  .
  .
  message:= 'Now hear';
  strappend (message, ' ' + 'this!');
  .
  .
END.
```

Strdelete

Usage

```
strdelete (s, startpos, n)
```

Parameters s is a string variable.
 startpos is an integer expression representing the starting index of the deletion.
 n is an integer expression representing the number of characters to be deleted.

Description

The procedure strdelete (s, startpos, n) removes n characters from s, starting at s[startpos].

Example:

```
VAR
  uncensored, censored: STRING [80];

BEGIN
  .
  .           { 11111111 }
  .           { 12345678901234567 }
  uncensored := 'Attack at 6 a.m.!'
  strdelete (uncensored, 7, 10);
  censored := uncensored;      {Censored is 'Attack!' .}
  .
END.
```

Strinsert

Usage

```
strinsert (s1, s2, n)
```

Parameters s1 is a string expression.
 s2 is a string variable.
 n is an integer representing the offset in s2 where insertion will begin.

Description

The procedure strinsert (s1, s2, n) inserts string s1 into s2 starting at s2[n]. The resulting string may not exceed maximum length of s2.

Example:

```
VAR
  remark: STRING [80];

BEGIN
  .           { 11111111}
  .           {12345678901234567}
  remark := 'There is missing!';
  strinsert (' something', remark, 9);
  .
END.
```

Strmove

Usage

`strmove (nchars, source, sourcepos, dest, destpos)`

Parameters	<code>nchars</code> is an integer expression indicating the number of characters to be copied. <code>source</code> is a string expression or PAC variable. <code>sourcepos</code> is an integer expression indicating the offset in source from which copying will start. <code>dest</code> is a string or PAC variable. <code>destpos</code> is an integer expression indicating the offset in dest where copying will start.
------------	--

Description

The procedure `strmove (nchars, source, sourcepos, dest, destpos)` copies `nchars` from `source` [`sourcepos`] to `dest` [`destpos`].

If the destination is a string, it must have a value before having characters moved into it (`Setstrlen (dest, 0)` may be appropriate). The current length will be incremented if it is made longer. If `destpos = Strlen (dest) + 1`, a concatenation is performed.

The programmer may use `strmove` to convert PAC's to strings and vice versa. It is also an efficient way to manipulate subsets of PAC's.

Example:

```

VAR
  s: STRING [20];
  pac: PACKED ARRAY [1..15] OF CHAR;

BEGIN
  .      {111111}
  .      {123456789012345}
  pac := 'Hewlett-Packard';
  Setstrlen (s, 0);
  strmove (15, pac, 1, s, 1); {Converts a PAC to a string.}
  .
END.

```

Strread

Usage

```
strread (s, startpos, nextpos, v1, ..., vn)
```

Parameters s is string expression.
 startpos is an integer expression.
 nextpos is an integer or integer subrange variable.
 v1...vn are simple, string or PAC variables. Any number of v parameters may appear separated by commas.

Description

The procedure strread (s, startpos, nextpos, v1..., vn) performs a symbolic to internal conversion from the contents of string s into variables v1...vn. The conversion and rules for allowable arguments are the same as for Read from TEXT files (see Chapter 6). The string s is treated as a single line of a TEXT file. An error occurs if strread attempts to read beyond the current length of s. The operation starts at s[startpos]. After the operation nextpos will have the value 1 greater than the index of the last character that was read.

The call

```
strread (s, p, t, v1, ..., vn);
```

is equivalent to

```
strread (s, p, t, v1);
strread (s, t, t, v2);
.
.
.
strread (s, t, t, vn);
```

Example:

```
VAR
  s: STRING [80];
  p,t: 1..80;
  m,n: INTEGER;

BEGIN
  .      { 1111111111}
  .      {1234567890123456789}
  s := '    12 564    ';

  p := 1;
  strread (s, p, t, m);           {m will be 12 and t will be 8}
  strread (s, t, t, n);           {n will be 564 and t will be 13}
  .
END.
```

Strwrite

Usage

```
strwrite (s, startpos, nextpos, e1, ..., en)
```

Parameters s is a string variable.
 startpos is an integer expression.
 nextpos is an integer or subrange variable.
 e1...en are simple or string expression, or PAC variables. Any number of e
 parameters may appear separated by commas.

Description

The procedure strwrite (s, startpos, nextpos, e1,..., en) performs an internal to symbolic conversion from the values of expressions e1...en into the string s, starting at s[startpos]. After the operation, nextpos will have the value 1 greater than the index of the last character that was written. The conversion rules for allowable arguments, and formatting are the same as for Write on TEXT files (see Chapter 6).

The string s must have a value prior to writing into it with strwrite; Setstrlen (s, 0) may be appropriate. If startpos=strlen (s) + 1 (the normal case), a concatenation is performed.

An error occurs if strwrite attempts to write beyond the maximum length of s, or if startpos is more than one greater than the current length of s.

The call

```
strwrite (s, p, t, e1,..., en);
```

is equivalent to

```
strwrite (s, p, t, e1);
strwrite (s, p, t, e2);
.
.
strwrite (s, p, t, en);
```

Example:

```
VAR
  s: STRING [80];
  t: 1..80;
  f,g: INTEGER;

BEGIN
  .
  .
  f := 100;
  g := 99;
  setstrlen (s, 0);  strwrite (s, 1, t, f:3);      {s is now '100'; t is 4.  }
  strwrite (s, t, t, ' ', g:2); {s is now '100 99'; t is 7.}
  .
  .
END.
```

String Functions

Str

Usage

`str (s, startpos, len)`

Parameters s is a string expression.
 startpos is an integer expression indicating the index of the starting character.
 len is an integer expression indicating the length of the substring.

Description

The function `str (s, startpos, len)` returns that portion of the string expression `s` which starts at `s[startpos]` and is of length `len`. The result is of type string and may be used as a string expression.

An error occurs if the current length of `s` is less than the sum of `startpos` and `len` minus 1.

Example:

```

VAR
  i:          INTEGER;
  wish_list: STRING [132];
  granted:   STRING [5];

BEGIN
  .
  .
  .
  i := 13;    {11111111112222222222}
  wish_list := 'wish1 wish2 wish3 wish4 wish5';
  granted  := str (wish_list, i, 5); {selects the 3rd wish}
  .
  .
END.

```

Strlen

Usage

```
strlen (s)
```

Parameter s is a string expression or PAC variable.

Description

The function strlen (s) returns the current length of the PAC or string expression s as a non-negative integer. With a PAC literal (or named PAC literal) argument Strlen may be used in a constant expression. If s is a string expression and RANGE is ON, a library routine is called which validates the string (to the extent that such is possible) before returning the current length; if RANGE is OFF, the current length is extracted directly from the string without a string validation check.

Example:

```
CONST
    length = strlen ('constant'); {length is 8}

VAR
    variable: STRING [80];

BEGIN
    .
    .
    {123456}
    variable := 'string';
    IF strlen (variable) >= length THEN BEGIN
        .
        .
    END;
    .
END.
```

Strltrim

Usage

```
strltrim (s)
```

Parameters s is a strim expression.

Description

The function strltrim (s) returns a string consisting of s trimmed of all leading blanks.

Example:

```
VAR
  s: STRING [35];

BEGIN
  .   { 11111111112}
  .   {12345678901234567890}
  s := '      leading blanks';
  s := strltrim (s); {s is now 'leading blanks'}
  .
END.
```

Strmax

Usage

```
strmax (s)
```

Parameters s is a string variable.

Description

The function strmax (s) returns the maximum length of the string variable s as a positive integer. If s is a formal parameter, the result is the maximum length of the corresponding actual parameter. If s is a VAR parameter of the anonymous STRING type and RANGE is ON, a library routine is called which validates the string (to the extent that such is possible) before returning the maximum length; if RANGE is OFF, the maximum length is extracted directly from the string without a string validation check.

Example:

```
VAR
  s:      STRING [80];
  filled: BOOLEAN;

BEGIN
  .
  .
  .
  s := 'This is not a full string!';
  IF strlen (s) = strmax (s) THEN BEGIN
    filled := true;
  END;
  .
  .
END.
```

Strpos

Usage

```
strpos (s1, s2)
```

Parameters s1 is a string expression.
s2 is a string expression.

Description

The function strpos (s1, s2) returns the integer index of the position of the first occurrence of s2 in s1. If s2 is not found, zero is returned.

Example:

```
CONST
    separator = ' ';

VAR
    i:      INTEGER;
    names: STRING [80];

BEGIN
    .          {11111111222222223333}
    .          {1234567890123456789012345678901234}
    names := 'John Susan Dave Chris Ron Jeff Roy';
    i := strpos (names, separator);
    IF i <> 0 THEN
        strdelete (names, 1, i); {deletes first name}
    .
END.
```

Strrpt

Usage

```
strrpt (s, count)
```

Parameters s is a string expression.
 count is an integer expression indicating the number of repetitions.

Description

The function strrpt (s, count) returns a string composed of s repeated count times.

Example:

```
CONST
  one = '1';

VAR
  b_num: STRING [32];

BEGIN
  .
  .
  b_num := strrpt (one, strmax (b_num));
  .
  .
END.
```

Strrtrim

Usage

```
strrtrim (s)
```

Parameter s is a string expression.

Description

The function strrtrim (s) returns a string consisting of s trimmed of all trailing blanks.

Example:

```
VAR
  s: STRING [35];

BEGIN
  .   {      11111111122}
  .   {123456789012345678901}
  s := 'trailing blanks      ';
  s := strrtrim (s); {s is now 'trailing blanks'}
  .
  .
END.
```

Arithmetic Functions

There are eleven predefined arithmetic functions in Pascal/1000. Each of these functions is passed an arithmetic expression as a parameter and returns a numeric or Boolean 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.

Table 7-1. System Routines Called to Calculate Function Values

	INTEGER	REAL	LONGREAL
<code>abs</code>	—	—	—
<code>arctan</code>	<code>ATAN</code>	<code>ATAN</code>	<code>.ATAN</code>
<code>cos</code>	<code>COS</code>	<code>COS</code>	<code>.COS</code>
<code>exp</code>	<code>EXP</code>	<code>EXP</code>	<code>.EXP</code>
<code>ln</code>	<code>ALOG</code>	<code>ALOG</code>	<code>.LOG</code>
<code>odd</code>	—	—	—
<code>round</code>	—	<code>.FIXD*</code>	<code>.TFXD*</code>
<code>sin</code>	<code>SIN</code>	<code>SIN</code>	<code>.SIN</code>
<code>sqr</code>	<code>.DMP</code>	<code>.FMP</code>	<code>.TMYP</code>
<code>sqrt</code>	<code>SQRT</code>	<code>SQRT</code>	<code>.SQRT</code>
<code>trunc</code>	—	<code>.FIXD</code>	<code>.TFXD</code>

* = after adding or subtracting 0.5 as appropriate.

Errors within system arithmetic routines are routed through the Pascal run-time error handling mechanism where possible. Some errors may still be reported to the system log device via `ERR0`. Refer to the appropriate system routine documentation for information on the errors which can be generated by these routines.

Standard Procedures and Functions

Abs

Usage

abs (x)

Description

The function abs returns the absolute value of x.

Examples:

abs (-13)	returns 13
abs (-7.11)	returns 7.110000E+00

Arctan

Usage

arctan (x)

Description

The function arctan returns the value of 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

Cos

Usage

cos (x)

Description

The function cos returns the value of the cosine of x, where x is interpreted as being in radians. If x is outside of the range -8192 * pi..8192 * pi then a run-time error occurs, if the error is ignored, a value of zero is returned.

Examples:

cos (0.024)	returns 9.997120E-01
cos (1.62)	returns -4.918370E+00

Exp

Usage

```
exp (x)
```

Description

The function exp returns the value of 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. A run-time error will occur on overflow, if the error is ignored, 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)
```

Description

The function ln returns the value of the natural logarithm of x. If $x < 0$ then a run-time error occurs, if the error is ignored, a value of zero is returned.

Examples:

ln (43)	returns 3.761200E+00
ln (2.121)	returns 7.518877E-01

Odd

Usage

```
odd (x)
```

where x is an INTEGER (or a subrange of INTEGER).

Description

The function odd returns TRUE if x is odd, and FALSE otherwise.

Examples:

odd (6)	returns FALSE
odd (-32767)	returns TRUE
odd (32768)	returns FALSE
odd (1)	returns TRUE

Round

Usage

```
round (x)
```

Description

The 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.

Examples:

round (3.1)	returns 3
round (-6.4)	returns -6
round (-4.6)	returns -5

Sin

Usage

```
sin (x)
```

Description

The function sin returns the value of the sine of x, where x is interpreted as being in radians. If x is outside of the range $-8192 \cdot \pi .. 8192 \cdot \pi$ then a run-time error occurs. If the error is ignored, a value of zero is returned.

Examples:

sin (0.024)	returns 2.399769E-02
sin (1.62)	returns 9.987898E-01

Sqr

Usage

```
sqr (x)
```

Description

The function sqr returns the value of x squared, (x^2). If the value to be returned is greater than the maximum value for that type, the maximum value of the type is returned.

Examples:

sqr (3)	returns 9
sqr (1.198E3)	returns 1.435204E+06

Sqrt

Usage

```
sqrt (x)
```

Description

The function sqrt returns the value of the square root of x, ($x^{1/2}$). If $x < 0$ then a run-time error occurs. If the error is ignored, a value of zero is returned.

Examples:

```
sqrt (64)      returns 8.000000E+00
sqrt (13.5E12) returns 3.674235E+06
sqrt (-5)      returns 0 (if run-time error ignored)
```

Trunc

Usage

```
trunc (x)
```

Description

The function trunc returns an integer value 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.

Examples:

```
trunc (5.61)      returns 5
trunc (-3.38)     returns -3
trunc (18.999)    returns 18
```

Numeric Conversion Functions

There are three predefined numeric conversion functions in Pascal/1000:

1. Binary
2. Hex
3. Octal

Each of these functions is passed a string or PAC parameter and returns a two-word (32-bit) integer. The parameter is interpreted as the ASCII representation of a number in the base corresponding to the function name. The function returns the converted value. The exact bit representation is not guaranteed since the result can be assigned (or passed) to an instance of a smaller subrange if the value is in the subrange.

All characters must be legal digits in the base signified by the function name (leading and trailing blanks are ignored).

Since binary, hex, and octal return an integer value, which is represented in a 32 bit quantity, the programmer must specify all 32 bits if a negative result is desired. Alternatively, the programmer may negate the positive representation.

References to these functions may appear anywhere that an integer expression may appear within a Pascal program (specifically when the parameter is a string or PAC literal or an identifier equated with a string or PAC literal they may appear in constant expressions which are evaluated at compile time).

During conversion, a compile-time or run-time error will be reported on overflow of a two-word integer, or detection of an illegal character for the base, or if there is no value to convert.

Binary

Usage

`binary (x)`

Description

The function `binary` computes the binary value of `x`. The legal characters in `x` are '0' and '1'.

Examples:

<code>binary ('111')</code>	<code>returns 7</code>
<code>-binary ('100010')</code>	<code>returns -34</code>

Hex

Usage

`hex (x)`

Description

The function `hex` computes the hexidecimal value of `x`. The legal characters in `x` are '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'a', 'B', 'b', 'C', 'c', 'D', 'd', 'E', 'e', 'F', and 'f'.

Examples:

<code>hex ('16f')</code>	<code>returns 367</code>
<code>hex ('fFfFfF01')</code>	<code>returns -255</code>

Octal

Usage

`octal (x)`

Description

The function `octal` computes the octal value of `x`. The legal characters in `x` are '0', '1', '2', '3', '4', '5', '6', and '7'.

Examples:

<code>octal ('77')</code>	<code>returns 63</code>
<code>octal ('37777777701')</code>	<code>returns -63</code>

Ordinal Functions

Chr

Usage

`chr (x)`

where `x` is an integer expression.

Description

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 run-time error occurs if such an out-of-range result is assigned to a variable of type CHAR.

For any character `ch`, the following is true:

`chr (ord (ch)) = ch`

Examples:

Value of <code>x</code>	Value of <code>chr (x)</code>
63	'?'
100	' '
13	(carriage return)

Ord

Usage

```
ord (x)
```

where x is an expression of ordinal type.

Description

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);
```

thus,

```
ord (red)      returns 0
ord (blue)     returns 1
ord (yellow)   returns 2
```

Additional Ord Examples:

Value of x	Value of ord (x)
'a'	97
'A'	65
-1	-1
1000	1000

Pred

Usage

```
pred (x)
```

where x is an expression of ordinal type.

Description

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

Succ

Usage

```
succ (x)
```

where x is an expression of ordinal type.

Description

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

Packing and Unpacking Procedures

Pack

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;

Description

The standard procedure pack assigns the values of elements of an unpacked array to a packed array.

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 element 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:

$$\text{lb1} \leq i \leq \text{len1}-\text{len2} + 1$$

where len1 is the number of elements in the unpacked array, lb1 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 statement "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

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.
```

Description

The procedure `unpack` assigns the values of elements of a packed array to an unpacked array.

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:

$$\text{lb1} \leq i \leq \text{len1} - \text{len2} + 1$$

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
      BEGIN
        a[k] := z[j];
        IF j <> v THEN k := succ (k);
      END;
    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.

Program Control Procedures

Halt

The procedure Halt terminates the program.

Usage

```
halt (n)
```

where n is an integer expression.

Description

When halt (n) is executed, program execution terminates. If n is less than zero, a message containing the value of 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.

If n is greater than or equal to zero, no message is displayed.

The lower order sixteen bits of n are made available as the first of the five parameters returned to the scheduling program via PRTN. The high order sixteen bits of n are made available as the second of the five parameters returned to the scheduling program via PRTN. The remaining three parameters are returned containing zero.

The parameter n can be omitted (the parentheses are then omitted as well), which is effectively halt (0).

Halt Example:

```
CONST
  div_by_0 = -99;

VAR
  x,y: REAL;

BEGIN
  :
  x := 0.0;
  IF x <> 0.0 THEN BEGIN
    y := y/x;
  END
  ELSE BEGIN
    halt (div_by_0);
  END
  .
  .
END.
```

the message:

Pascal Halt: -99

will be displayed. The first parameter returned via PRTN will be -99 and the second will be -1 (the high order sixteen bits of the thirty-two bit representation of -99).



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 management, 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 chapter 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.

Allocations of Scalar Variables

Table 8-1 shows the allocations for variables of scalar, subrange, and pointer types. All allocations begin on word boundaries.

Allocations for Structured Variables

Table 8-2 shows the allocations for variables of array, record, file, string, and set 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	<p>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.</p> <p>Examples:</p> <table> <thead> <tr> <th>Subrange</th> <th>Allocation</th> </tr> </thead> <tbody> <tr> <td>0..8</td> <td>1 word</td> </tr> <tr> <td>-32768..32767</td> <td>1 word</td> </tr> <tr> <td>10..40000</td> <td>2 words</td> </tr> <tr> <td>-70000..-1</td> <td>2 words</td> </tr> </tbody> </table>	Subrange	Allocation	0..8	1 word	-32768..32767	1 word	10..40000	2 words	-70000..-1	2 words
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.										

Table 8-2. Allocations for Structured Variables

UNPACKED TYPE	SIZE								
ARRAY	<p>The size of an array allocation is the sum of the allocations of its elements: $(\text{product of cardinalities} \times (\text{allocation of one element}))$</p> <p>The elements are stored in row major order.</p>								
RECORD	<p>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.</p>								
FILE	<p>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 128).</p> <p>Allocation size for a text file (words): $30 + 128*b + (s+1) \text{ DIV } 2$</p> <p>Allocation size for a non-text file (words): $29 + 128*b + \text{allocation size of base type}$</p>								
SET	<p>Let n be the cardinality of the base type.</p> <table> <tr> <td style="padding-right: 20px;">n</td> <td>Allocation</td> </tr> <tr> <td>≤ 16</td> <td>1 word</td> </tr> <tr> <td>> 16</td> <td>1 word to represent n plus the number of words required to represent n bits, i.e.,</td> </tr> <tr> <td>≤ 32767</td> <td>$1 + (n+15) \text{ DIV } 16$</td> </tr> </table> <p>If the compiler cannot discern the base type, 17 words (1 word + 16 words to represent 256 elements) are allocated.</p> <p>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.</p>	n	Allocation	≤ 16	1 word	> 16	1 word to represent n plus the number of words required to represent n bits, i.e.,	≤ 32767	$1 + (n+15) \text{ DIV } 16$
n	Allocation								
≤ 16	1 word								
> 16	1 word to represent n plus the number of words required to represent n bits, i.e.,								
≤ 32767	$1 + (n+15) \text{ DIV } 16$								
STRING	<p>The size of a string allocation is two characters per word plus the string header size:</p> $(\text{num_of_char} + 1) \text{ DIV } 2 + 2 \text{ (words)}$								

Allocations 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*.)

Packed files are identical to unpacked files.

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	Size: Minimum number of bits necessary to represent the largest absolute value in the subrange (plus one if the subrange contains negative numbers). Alignment: Bit boundary
ENUMERATION	Size: Minimum number of bits necessary to represent the ordinal value of the maximum value of the subrange. 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. $n \leq 16$: Size: n bits Alignment: Bit boundary $16 < n \leq 32767$: Size: 1 word + m words where m is the number of words to hold n bits. Alignment: Word boundary
STRING	Size: (num_of_char + 1) DIV 2 + 2 (words) Alignment: Word boundary

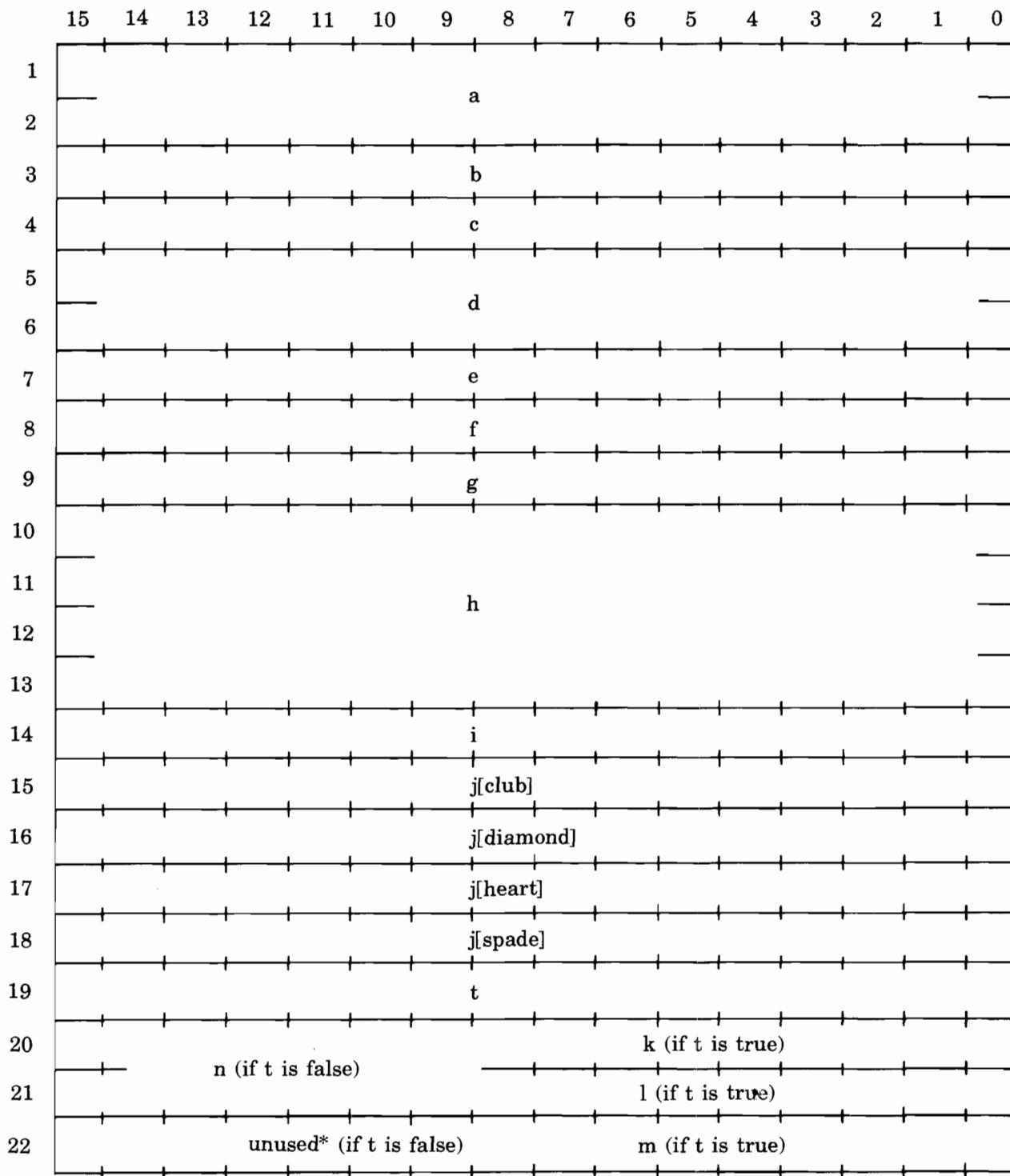
Examples of Packed and Unpacked Structures

Example 1: Assume the following:

```
TYPE
  SUIT = (club,diamond,heart,space);

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.

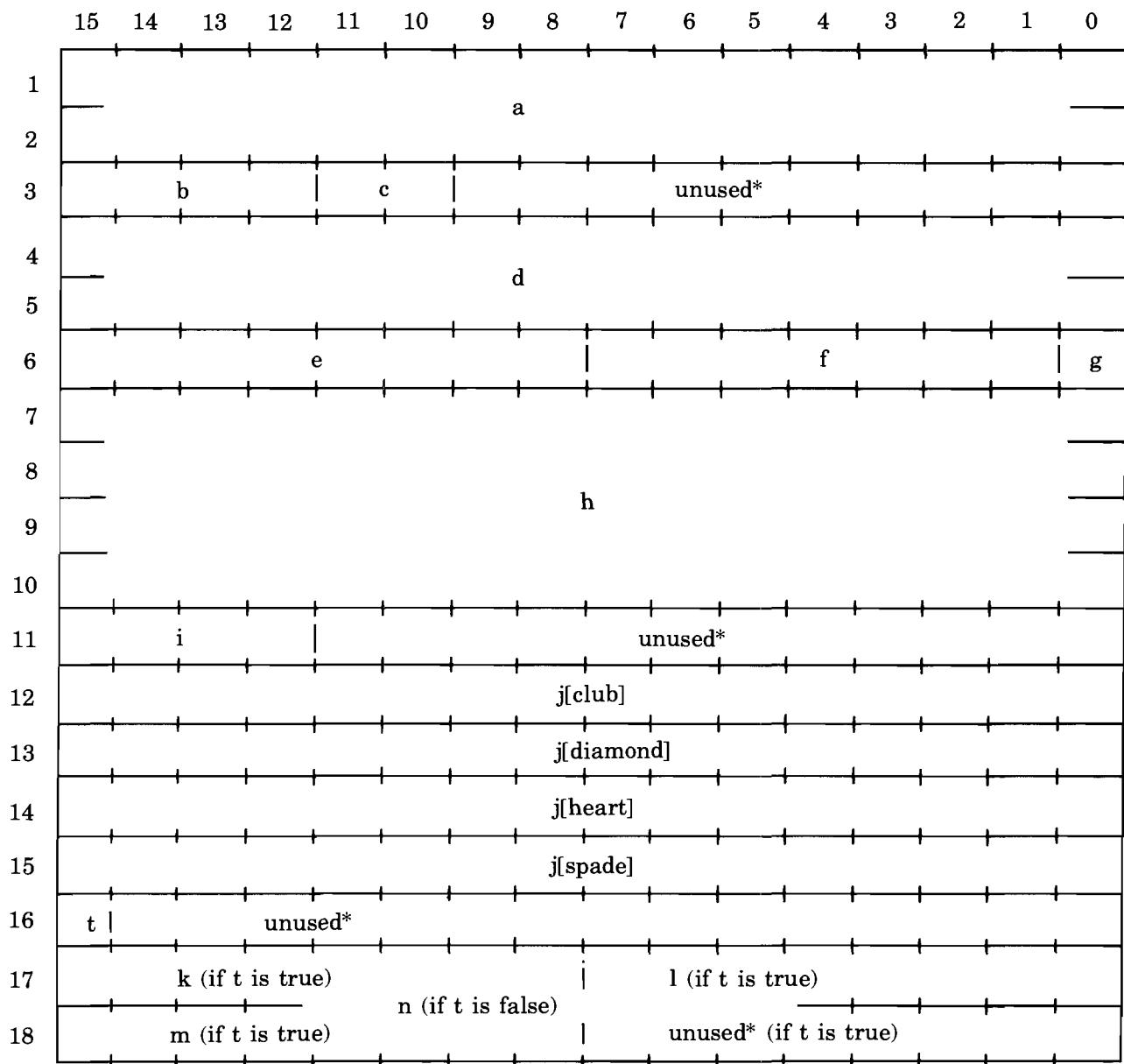
Implementation Considerations

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.

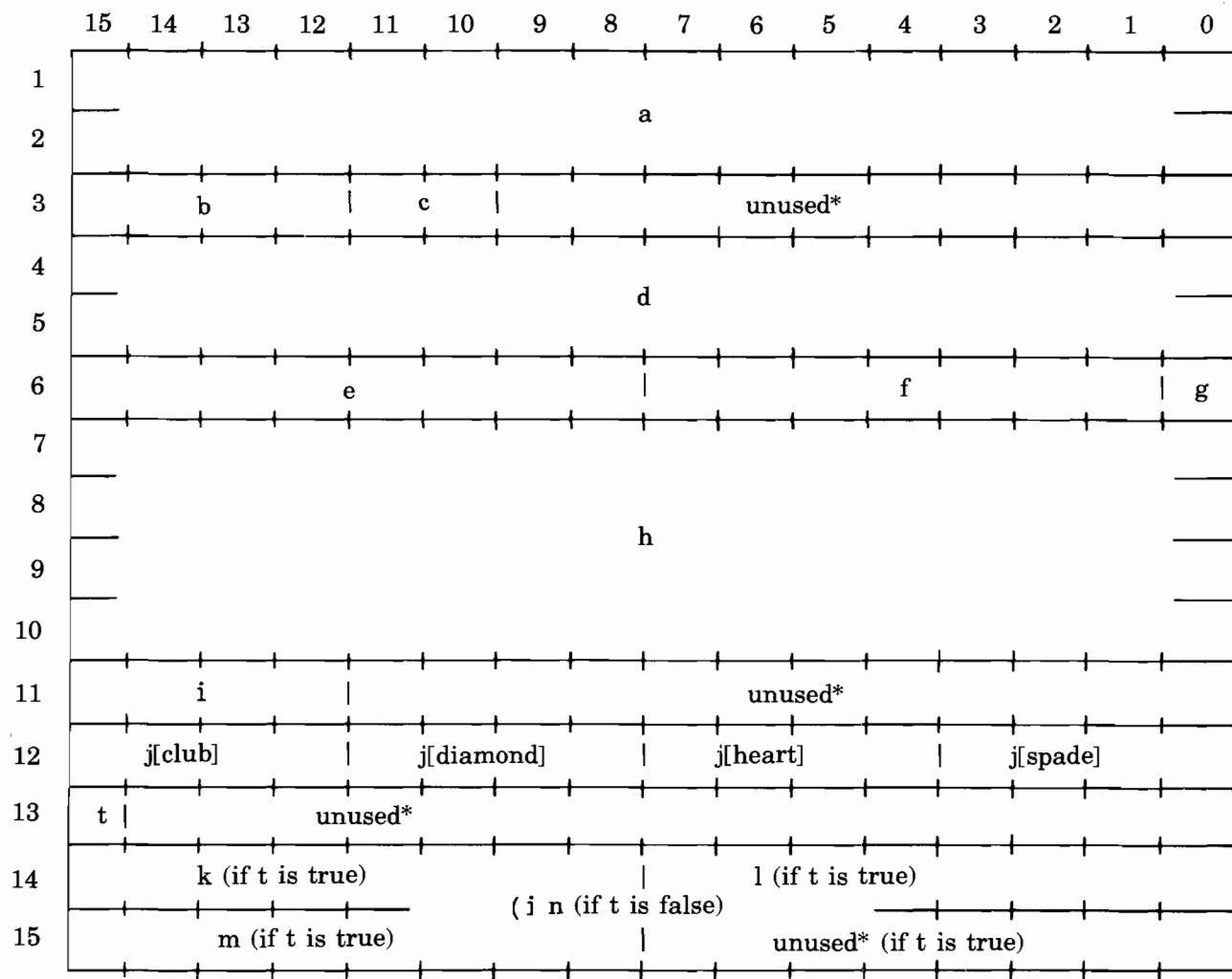
Implementation Considerations

Example 3: This is similar to the previous example, but field j is now 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 : 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.

Non-CDS Memory Configuration

Memory configuration, as discussed in this section, is the configuration of a partition in which a non-CDS Pascal/1000 program is running. Figure 8-1 illustrates the allocation of memory in a partition.

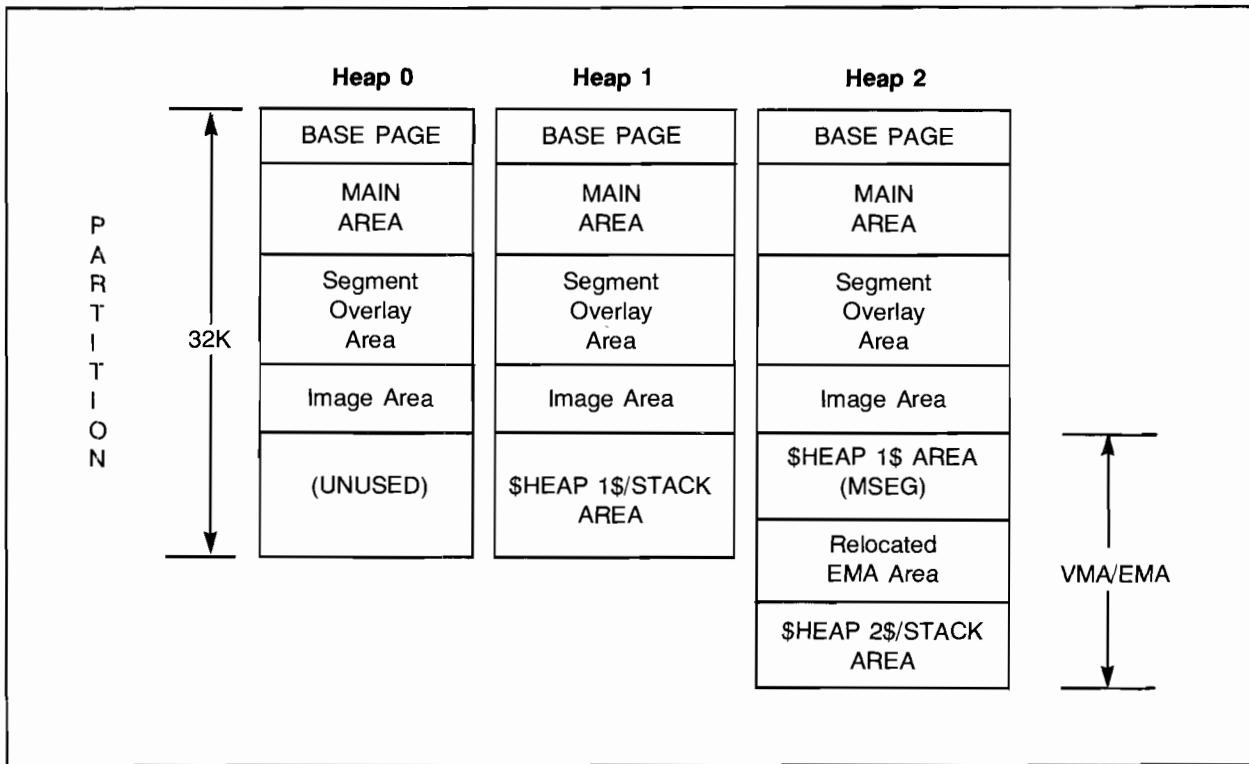


Figure 8-1. Pascal/1000 Non-CDS Memory Configuration

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 default \$HEAP 2\$/stack area is allocated in VMA/EMA, although the \$HEAP 1\$ area is initialized and is available in the users partition under certain circumstances. The stack starts above any relocated EMA area (if the program was loaded with LINK).

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 and module list 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 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 where then searched to supply the routines used in the main area.

```

PROGRAM main;
{main data declarations}

PROCEDURE proc1;
{proc1 local declarations}
BEGIN
{proc1 code}
END;

PROCEDURE proc2;
{proc2 data declarations}
PROCEDURE proc3;
{proc3 data declarations}
BEGIN
{proc3 code}
END;
BEGIN
{proc2 code}
END;

PROCEDURE fortn;
EXTERNAL;
{ftn is a FORTRAN routine}

BEGIN
{main code}
END.

$SUBPROGRAM$
PROGRAM subprogram;
{redeclaration of global
 declarations}
PROCEDURE sub1;
{sub1 data declarations}
BEGIN
{sub1 code}
END;
PROCEDURE sub2;
{sub2 data declarations}
BEGIN
{sub2 code}
END;
. {End of subprogram}

```

Main Area

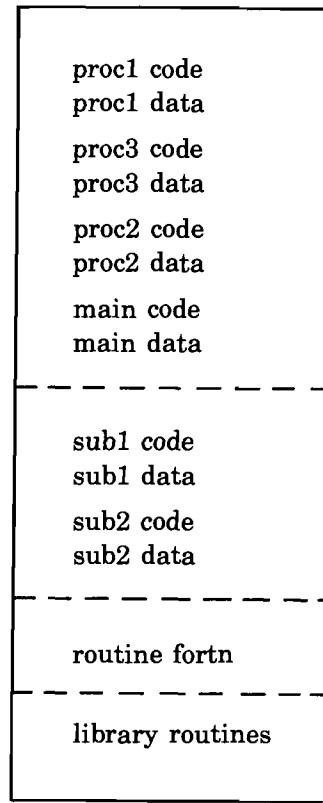


Figure 8-2. Main Area

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: default heap/stack area in logical memory; allocated at the end of the partition (default value)
- n = 2: default heap/stack area in VMA/EMA (although the \$HEAP 1\$ area is also initialized and available under certain circumstances)

Implementation Considerations

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 further in the Non-CDS Stack Management section in this chapter.

The stack and heap *grow* toward each other at 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.

CDS Memory Configuration

Memory configuration, as discussed in this section, is the configuration of the partitions in which a CDS Pascal/1000 program is running. Figure 8-3 illustrates the allocation of memory for a CDS program.

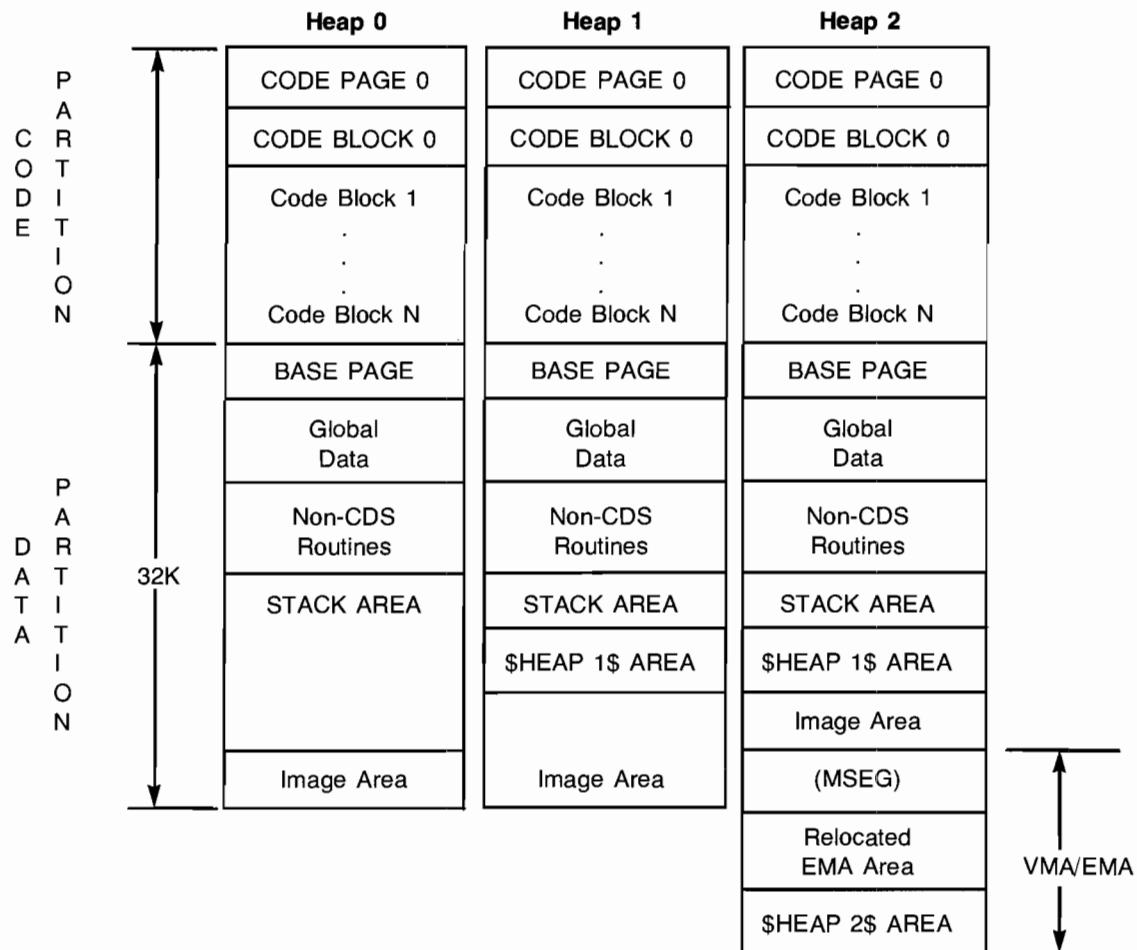


Figure 8-3. Pascal/1000 CDS Memory Configuration

The areas labeled in upper case are always allocated; those labeled in lower case are optional.

Implementation Considerations

The stack starts just above any global data and non-CDS code in the data partition and grows toward increasing addresses under hardware control. The \$HEAP 1\$ area (if present) begins below the image area (if present) and/or below the MSEG area and grows toward decreasing addresses under program control.

In the Heap 2 configuration, the default \$HEAP 2\$ area is allocated in VMA/EMA, while the hardware stack remains in the data partition. The \$HEAP 1\$ area is initialized and is available in the data partition under certain circumstances.

Code Partition

Code page zero contains information for use by the system in managing the execution of a CDS program. Information on code page zero is not normally accessible to the program.

One or more code blocks contain code segments which are defined (or defaulted) at load time. Each code segment contains one or more level-1 procedures or functions. There can be more code segments than there are code blocks, forcing some code segments to reside on disk until needed. When a procedure or function invocation refers to a routine in a code segment other than the one which is currently executing, the new segment is automatically made active if it is resident in a code block. If the segment is not resident, one of the code blocks is overlayed with the needed segment from the disk, and then made active.

For more information regarding the definition of code segment contents, refer to the RTE-A Link User's Manual.

For more information regarding the contents and layout of a code partition, refer to the RTE-A Programmer's Reference Manual.

Base Page

The base page is the first logical page of the data partition. It contains the communication area of the operating system, driver links, trap cells for interrupt processing, and operating system and user program links.

Global Data Area

The global data area of the data partition contains the main program unit's global variables, variables declared within modules, and string literals and structured constants declared with the CODE_CONSTANTS option OFF.

Non-CDS Routine Area

User written or library routines which are invoked by the CDS program but which do not themselves execute in the CDS environment reside in the data partition intermixed with objects in the global data area (the order of mixing is a function of how the program was loaded).

Stack Area

The hardware maintained procedure and function call stack begins just above any global data and non-CDS code in the data partition and grows toward increasing addresses under hardware control.

Local variables are allocated on the stack on procedure or function entry and removed on procedure or function exit.

The stack is discussed further in the CDS Stack Management section in this chapter.

Heap Area

When a CDS program is LINKed (or reLINKed) the stack and heap sizes can be specified. Pascal, however, ignores this specification, and merges the heap area into the stack area. It then removes stack area for the image area (if appropriate) and then establishes the top of the \$HEAP 1\$ area at the end of the remaining stack area. As dynamic variables are allocated on the \$HEAP 1\$ area, stack area is converted to heap area as required.

The heap is used for dynamic data and is described in detail in the Heap Management section in this chapter.

Image Area

If the Pascal/1000 program interfaces with the IMAGE/1000 subsystem then an area of the data partition must be set aside for use by IMAGE. The compiler option \$IMAGE n\$ allocates n words of IMAGE area above the \$HEAP 1\$ area, if present, or above the stack area if no \$HEAP 1\$ area is present.

Heap/Stack

The stack area provides storage for local variables of procedures and functions. The heap area contains the run-time heap of the program. The size of the heap area is determined by the \$HEAP n\$ compiler option,

where:

- n = 0: no heap area (not allocated)
- n = 1: default heap area in logical memory; allocated at the end of the data partition (default value)
- n = 2: default heap area in VMA/EMA (although the \$HEAP 1\$ area is also initialized and available under certain circumstances)

These three conditions are illustrated in Figure 8-3.

The stack and heap grow toward each other at 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 the hardware stack meets or attempts to cross the \$HEAP 1\$, image, or mseg area, the program will be aborted with a CS06 (stack overflow error). If the \$HEAP 1\$ heap meets or attempts to cross the hardware stack area or the \$HEAP 2\$ heap meets or attempts to cross the relocated EMA area (or VMA/EMA location zero if no relocated EMA area), the run-time error:

```
*** Pascal Error: Heap/Stack Collision In Line xxxx
```

is displayed and the program will abort.

Data Management

There are four classes of data in Pascal/1000 programs:

1. global data
2. local data
3. non-local data
4. 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 3). 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 statically allocated in non-CDS programs and is allocated on the stack on routine entry in CDS programs. It 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.

The non-local data of a routine includes local data of routines which statically enclose the routine, but does not include global data.

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).

Non-CDS Management

The run-time stack is used by Pascal/1000 programs to save copies of a routine'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.

The following library routines allow a user to initialize the stack and to retrieve and set information about the stack:

```
Pas.InitialHeap1
Pas.InitialHeap2
Pas.GetMemInfo1
Pas.GetMemInfo2
Pas.SetMemInfo1
Pas.SetMemInfo2
```

CDS Stack Management

The hardware stack is used by Pascal/1000 programs to allocate local data and to hold copies of constants stored in code space. On entry to a routine, local data is allocated and copies of the code space constants are made.

The currently active procedure or function stores its local variables at offsets from the current stack frame, the address of the current stack frame moves up with each procedure or function invocation and back down on each procedure or function exit. The address of the current stack frame is maintained in a hardware register designated the Q register. A hardware register designated the Z register keeps the stack from growing into the \$HEAP 1\$, image, or mseg areas. The Z register contains an address 264 words below the address of the top of the heap 1 area (or image area as appropriate). After each procedure or function invocation the hardware checks to see if the current stack frame extends beyond the address in the Z register, and if it does, the program is aborted with a CS06 (stack overflow) error. The 264 word offset is used because the check for overflow is done after a new stack frame is established, and the hardware can modify any of the first 264 words of the new frame before checking for overflow.

Non-local data is accessed by indirect reference through a table called a *display table*, which contains the addresses of the stack frames for the most recent pending (active but not yet completed) instance of each routine in which the current routine is statically nested.

The following library routines permit a user to retrieve and modify certain information about the stack.

```
Pas.InitialHeap1  
Pas.InitialHeap2  
Pas.GetMemInfo1  
Pas.GetMemInfo2  
Pas.SetMemInfo1  
Pas.SetMemInfo2
```

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 refer to the \$HEAP 1\$ heap. The differences between \$HEAP 1\$ and \$HEAP 2\$, as well as CDS considerations, 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.

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. Should there be insufficient space to allocate the new data space at the top of the heap, an attempt will be made to coalesce free blocks, on the current mark level, into a space large enough for the data space. Should this attempt fail to produce a large enough space, a run-time error will occur. The routine that coalesces free blocks is user callable to permit preventive coalescing (see Appendix F).

At no time is garbage collection of allocated data areas performed.

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 to 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 cause unpredictable results. An attempt to dispose of any such variable will cause 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:

```
Pas.InitialHeap1    (initialize_heap)  
Pas.GetMemInfo1   (get_heap_stack_info)  
Pas.SetMemInfo1   (set_heap_stack_info)
```

NOTE

Pas.InitialHeap2, Pas.GetMemInfo2 and Pas.SetMemInfo2 are used with the compiler option \$Heap 2\$.

The parameter for these routines must be of the record type INFO_REC in Figure 8-4. The heap management routines will be described in terms of the data structures and identifiers in Figure 8-4. The three library routines above can be made available in the user's program by including the code from Figure 8-4 in the program 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 'Pas.GetMemInfo1'
  (VAR heap_info : INFO_REC);
EXTERNAL;

PROCEDURE set_heap_stack_info
  $ALIAS 'Pas.SetMemInfo1'
  (heap_info : INFO_REC);
EXTERNAL;

PROCEDURE initialize_heap
  $ALIAS 'Pas.InitialHeap1';
EXTERNAL;

```

Figure 8-4. 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 8-5. Heap diagrams have higher addresses at the bottom, lower addresses at the top. The heap grows toward lower addresses.

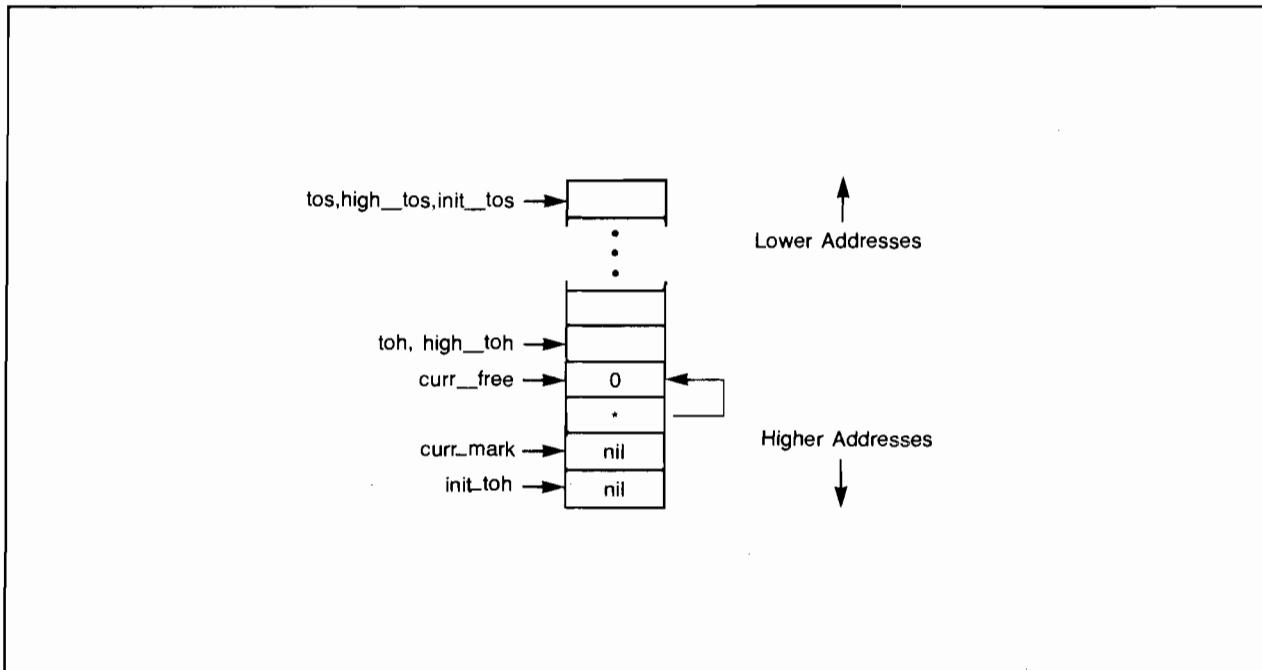


Figure 8-5. Heap/Stack Area After Initialization

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.

Implementation Considerations

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 8-6 and 8-7.

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, attempt to coalesce the free list at the current mark level. If a large enough space results, allocate the variable at the end of the free space. If insufficient space was recovered, 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.

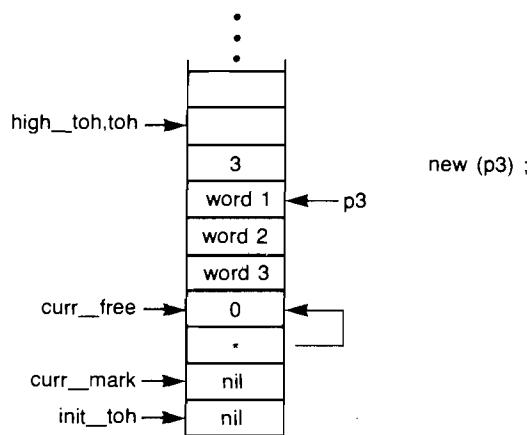


Figure 8-6. Allocation of a 3-word Variable

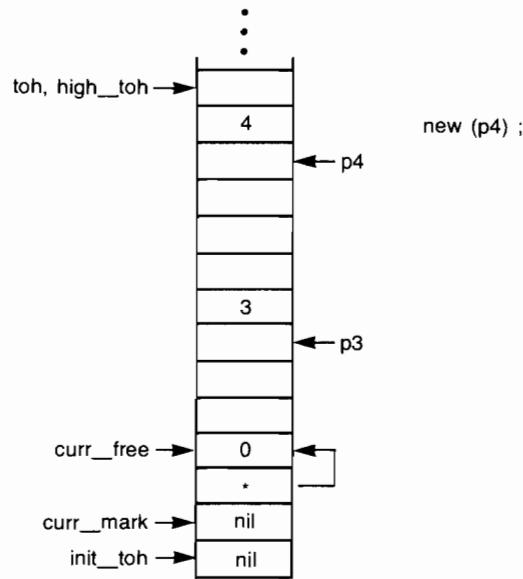


Figure 8-7. Allocation of a 4-word Variable

Dispose

See Figure 8-8.

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.

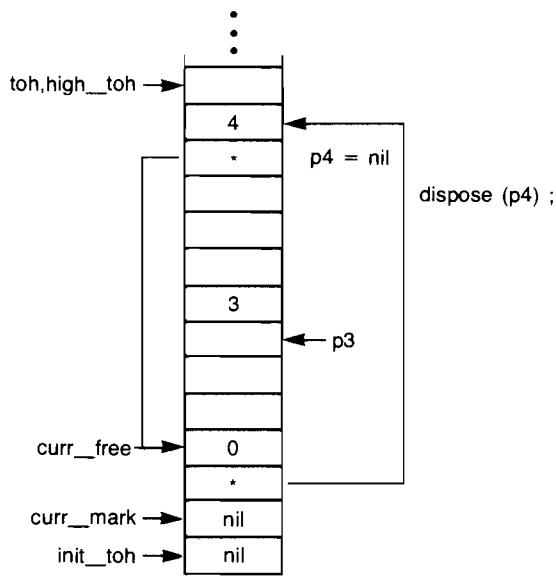


Figure 8-8. Disposing a 4-word Variable

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 8-9.

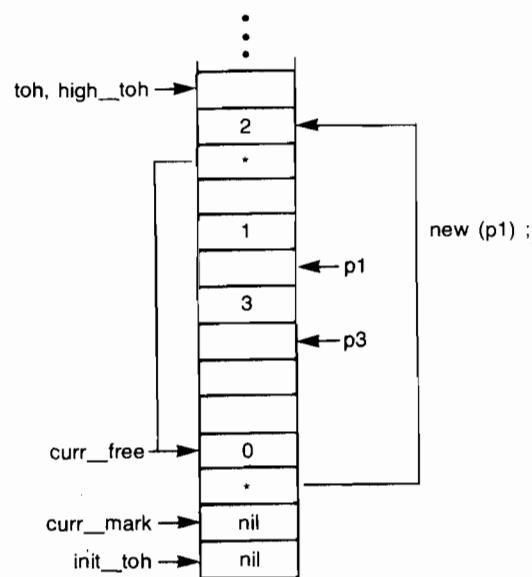


Figure 8-9. Allocation of a 1-word Variable

Implementation Considerations

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, 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 8-7. This would apply if the statement above were new (p4).

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 $n+1$, then the remaining word is not used and cannot be allocated. This would apply if the statement in Figure 8-9 were new (p3).

Mark

See Figure 8-10.

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.

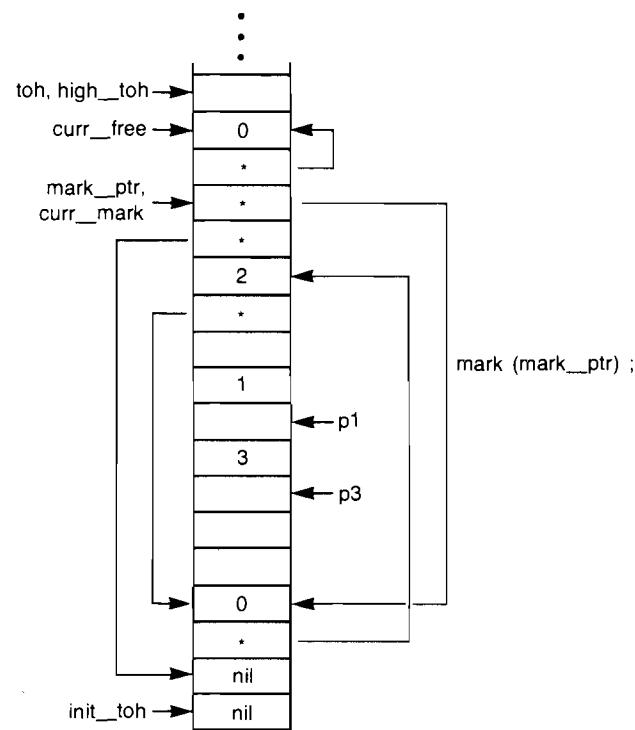


Figure 8-10. Creating a New Mark Region

Remember that new will only search the free list in the current mark region. The mark block 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 8-11.

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.

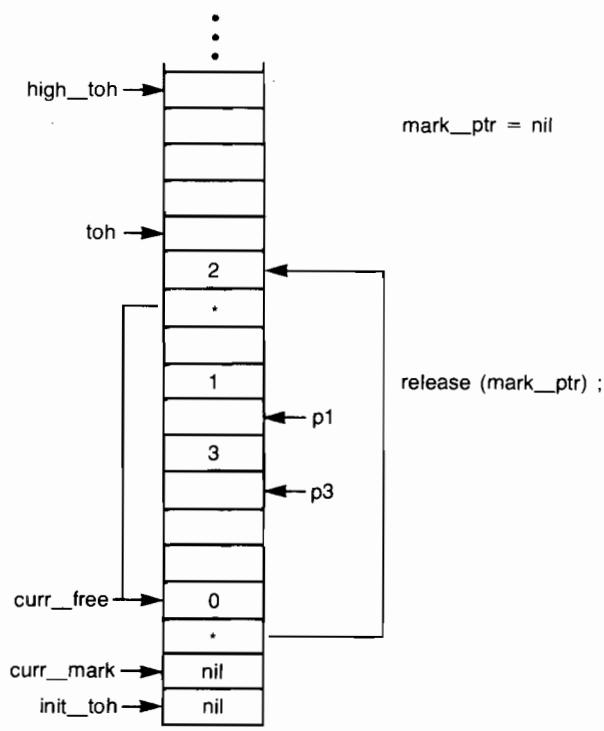


Figure 8-11. Releasing a Mark Region

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 8-4 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
  $ALIAS 'Pas.GetMemInfo2'$*
  (VAR heap_info : INFO_REC);
  EXTERNAL;

PROCEDURE set_heap_stack_info
  $ALIAS 'Pas.SetMemInfo2'$*
  (heap_info : INFO_REC);
  EXTERNAL;

PROCEDURE initialize_heap
  $ALIAS 'Pas.InitialHeap2'$;
  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.

CDS Heap Management

For programs running in the CDS environment, the following additional considerations apply.

The routines which get and set heap/stack information operate differently. Refer to Appendix D for further information.

When a dynamic variable is allocated at the top of the \$HEAP 1\$ area, the Z register is moved down appropriately to protect the \$HEAP 1\$ area from the hardware stack.

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 only a data space, without a 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 compiler option HEAP_DISPOSE must be turned OFF.

Figure 8-12 shows the definition of type INFO_REC as used by the short routines in callls to:

```
get_heap_stack_info
set_heap_stack_info
initialize_heap_stack_info.
```

Heap initialization is not necessary and is not performed by these procedures.

```
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 8-12. Definitions Used by Short Heap Management Routines

Implementation Considerations

For the short version of NEW (p):

1. If $\text{toh} - (\text{size of new variable}) \leq \text{tos}$ then report:

***** Pascal Error: Heap/Stack Collision In Line xxxx**

and abort the program. Otherwise, set toh to the value of

$\text{toh} - (\text{size of new variable})$

and set p to the value of

$\text{toh} + 1$

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 $p < \text{toh}$ 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 $p < \text{toh}$ 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.

Programs

Mixed Heap Programs

Programs can contain compilation units compiled under any of the \$HEAP\$ options under certain conditions and restrictions.

1. If any compilation unit in a program is \$HEAP 2\$ and
 - a. is compiled for either the non-CDS or the CDS environment and uses the heap (via new, dispose, mark, or release).

and/or

- b. is compiled for the non-CDS environment and does recursion.

then the main program must be a \$HEAP 2\$ program to ensure that the \$HEAP 2\$ heap/stack area is present and properly initialized. A main program compiled with the \$HEAP 2\$ option has both the \$HEAP 1\$ and \$HEAP 2\$ areas present, initialized, and accessible from units compiled with the appropriate option. A main program compiled with the \$HEAP 1\$ option is only guaranteed to have the \$HEAP 1\$ area present and initialized.

2. If modules are used as the only unit of separate compilation in addition to the main program, parameter passing and heap access are handled automatically.
3. If any compilation unit calls an EXTERNAL procedure or function, the following additional constraints apply.
 - a. If VAR parameters are passed from a \$HEAP 0\$ or \$HEAP 1\$ unit to a \$HEAP 2\$ unit, the actual parameters in the \$HEAP 2\$ unit must be declared with \$HEAPPARMS OFF\$.
 - b. If VAR parameters are passed from a \$HEAP 2\$ unit to a \$HEAP 0\$ or \$HEAP 1\$ unit, the formal parameters in the EXTERNAL declaration must be declared with \$HEAPPARMS OFF\$.
 - c. If any pointer types or variables of (or containing) pointer types are declared globally outside of any procedure or function and not within a module, then all compilation units which share those global declarations must be compiled with the same \$HEAP\$ setting to ensure correct global alignment.
 - d. Variables of (or containing) pointer types must not be passed as parameters or returned as function results via EXTERNAL procedures or functions between a \$HEAP 0\$ or \$HEAP 1\$ compilation unit and a \$HEAP 2\$ compilation unit and vice versa.
4. Parameters can be passed between \$HEAP 0\$ and \$HEAP 1\$ units without any additional constraints.

Modules

Relocating and Searching Modules

If a module compilation unit is to be searched during linking, it should be indexed (with LINDX) before being searched.

Overlay Considerations

In a non-CDS program, module units should usually be relocated or searched into the main program. If they are to be relocated or searched into overlay/segments, use extreme care. If the module static data area is placed in an overlay/segment then it does not have a global lifetime, it only maintains its values as long as the overlay/segment is loaded and has not been overlaid.

Mixed Heap Modules

Modules can be imported into one another regardless of the \$HEAP\$ option settings used to compile the modules.

Parameters are passed correctly between such modules.

Modules compiled \$HEAP 0\$ or \$HEAP 1\$ are assumed to have \$HEAPPARMS OFF\$ for all exported VAR parameters when imported into a \$HEAP 2\$ module.

Modules compiled with different \$HEAP\$ option settings can use the heap and will access the heap area appropriate to the setting of the \$HEAP\$ option. Exported pointer types and pointer variables will have a size and heap area access appropriate for the module which exports them, not for the module which imports them.

If a program uses any \$HEAP 2\$ modules (either directly importing them or importing modules which import them, etc.) that use the heap, the main program must be compiled with \$HEAP 2\$ option to ensure that the \$HEAP 2\$ area is initialized. If the main program is compiled with the \$HEAP 2\$ option, both the \$HEAP 1\$ and \$HEAP 2\$ areas exist, are initialized, and can be accessed by program units compiled with the appropriate setting of the \$HEAP\$ option. If the main program is compiled with the \$HEAP 0\$ or \$HEAP 1\$ option, the \$HEAP 2\$ area will not be initialized.

Using Files Within Modules

Modules which declare file variables or variables which contain files (whether exported or hidden), must be made known to the main program or to a segment/overlay unit to ensure that the files are properly initialized.

They can either be directly imported or imported into another module which in turn is directly imported (or recursively is itself imported into a module which in turn indirectly imported, etc.).

If a program consists of only a main and a series of module-list units, this will automatically be the case.

Such importation is necessary if the main or segment unit with which the \$SUBPROGRAM\$ will be relocated, does not itself import, either directly or indirectly, the modules imported by the \$SUBPROGRAM\$.

Overlay Considerations

If a non-CDS program contains SEGMENT/overlay units and these SEGMENT/overlays are to contain modules, there are additional constraints.

- The SEGMENTED option must be specified in the main to prevent the main program from trying to initialize files relocated in overlays and to prevent overlays from trying to initialize files relocated in the main.
- If a module is to be relocated into an overlay, it must be imported into the SEGMENT unit that comprises the basic overlay, or files in the module will not be properly initialized.
- Since modules relocated into SEGMENT/overlay units only maintain their data values while the overlay is loaded and has not been overlaid, files in such modules should be closed before loading another overlay.
- File variables in modules relocated with (and imported into) a SEGMENT/overlay will be reinitialized each time the overlay is loaded.

Creating Unique External Names

Routines in modules, whether exported or hidden within the implement section, must have unique external names for use by loaders/linkers and Symbolic DEBUG. The compiler tries to form the longest possible external name (16 chars max) in an attempt to guarantee uniqueness.

An external name for a module's routine is created by a concatenation separated by '.' of part/all of the module and routine names. If the module/routine name is seven characters or less, its entire name is used. Up to seven characters of the module name or ALIAS will be used, followed by a dot. The remaining characters (at least eight and as many as 14) will be taken from the routine name or ALIAS (see below).

The ALIAS option has additional capability when applied to a routine being declared in a module (whether exported or implemented). If the ALIAS string does not begin with a +, the ALIAS will be used exactly as specified as the external name (overriding the construction rules noted above). This permits non-constructed externally visible names to be used, but risks a potential external name duplication if not used carefully. If the ALIAS string begins with a +, the characters following the + will be used in the construction rules noted above to form the end of the constructed name.

Module name :	AA	ANT	LONG_NAME
Routine name :	BB	HIPPOPOTAMUS	LONGER_NAME
External name:	AA.BB	ANT.HIPPOPOTAMUS	LONG_NA.LONGER_N

In some cases, long names will require the user to use the ALIAS option to change a routine name (or the module name) so that its external name is unique. To avoid conflict with Pascal generated externals, the module name (or ALIAS) PAS should not be used.

Strings

Passing Strings to Other Languages

Pascal strings cannot be passed directly to any other language, nor can strings from other languages be passed directly to Pascal.

Strings may be passed to languages and subsystems which support the fixed string format (FTN7X and the new file system access routines) through the use of the FIXED_STRING option (see Appendix C).

Strings may be passed to BASIC/1000C through the use of the BASIC_STRING option (see Appendix C).

Routines which expect PAC should not be passed *strings* (i.e., FMGR, EXEC, etc.). The standard procedure Strmove can be used to move string data to a PAC which can be passed to such routines, or the library routine Pac.StringData can be used to obtain a pointer to the PAC part of a string.

Using Strings of Unknown Length

Some string expressions have a maximum length whose value may not be determinable at compile-time. These include expressions which involve the standard function Strrpt with a variable repeat count or unknown length string argument, or a VAR parameter of type String (such an expression will be referred to below as an ULSE, which stands for Unknown Length String Expression).

When assigned to local or global string variables, or passed as value parameters, the maximum length of a ULSE is implied by the maximum length of the variable or parameter, and is known at compile time.

When used in a comparison or when assigned to a VAR parameter of the anonymous STRING type, the maximum length of a ULSE cannot be determined. In these cases, the maximum length is presumed to be 256 characters. If a larger maximum length is required, the expression must be assigned to a local or global variable of the desired maximum length and that variable used in place of the original expression.

I/O

Program Heading Files

The program headings of SUBPROGRAM and SEGMENT units must contain all of the file identifiers in the same order that are specified in the main program unit.

The program headings of SUBPROGRAM and SEGMENT units may only omit the file identifiers, and, in fact, must omit them, if the SUBPROGRAM or SEGMENT is an autonomous library which declares no global string literals, no global structured constants, and no global variables.

The files INPUT and OUTPUT need not be declared in the program headings of SUBPROGRAM and SEGMENT units, in order to be used in such units. If they are so used, the main program unit must declare them in its program heading or the program will not load.

Standard Files

The standard files INPUT/OUTPUT are allocated their own data areas and are not part of the overall global data area. They may be omitted from or appear in any position in the program heading of a SUBPROGRAM or SEGMENT unit without effecting global storage allocation.

The standard files INPUT/OUTPUT cannot be accessed from any procedure or function in a main program unit, nor from the body of the main program, unless they are specified in the program heading of the main program.

The standard files INPUT/OUTPUT can be accessed from any procedure or function in a SUBPROGRAM or SEGMENT compilation unit or within a module. These files need not be specified in the program heading of such units, but they must not have been redefined.

If any SUBPROGRAM, SEGMENT, or module uses INPUT/OUTPUT, they must appear in the program heading of the main program or undefined external errors will occur at load time.

Automatic Association

If INPUT/OUTPUT appear in the program heading of the main program unit, but no corresponding run string parameter is present when the program starts, the LU number returned by LOGLU will be used as the file to be associated with INPUT/OUTPUT.

Direct-Access Files

Three predefined functions are available in HP Pascal to provide position information with respect to a currently open direct access file.

1. Position — the record number of the current position of the file pointer in the file.
2. Maxpos — the record number of the last record that could ever be in the file.
3. Lastpos — the record number of the last record that has ever been written to the file.

HP Pascal specifies that if lastpos cannot be determined for a particular implementation that lastpos should return the same value as maxpos. In Pascal/1000 this is the case, and lastpos returns the same value as maxpos.

Maxpos is the record number of the last record that could ever be in the file (it is a function of the maximum file size divided by the record size), but it does not guarantee that all records from 1 to maxpos can be written to the file (due to media limits of free space limits on the media). Direct access files can be sparse, in that record N can be written without writing records prior to record N. The file system uses disk space only for records which have actually been written, which is why it is not guaranteed that all logically possible records can be written. If a non-existent record is read, zeros will be returned and no error will be reported, as long as the record number is less than or equal to that returned by maxpos.

Files in the Heap

Dynamic variables can be files or data types which contain files (arrays or records).

Such files are initialized when the dynamic variable is created with new and are closed when the dynamic variable is destroyed with dispose. Note that the release procedure does not close any files which may exist in the heap area which is being released, only dispose closes files in the heap.

In a compilation unit compiled with the \$HEAP 2\$ option, files in the heap must have a total size of 1024 words or less. This size includes the Pascal file overhead (14 to 16 words), the DCB (144 words), optional \$BUFFERS\$ (128 words each), the file buffer variable (size of the file's component type), and for text files, the line buffer size (controlled by \$LINESIZE\$). Attempts to use files in the heap which are larger than 1024 words will elicit syntax error 398.

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, extendable, random access file.
2	A fixed length, user defined record length, 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 contains absolute binary code.
>7	Same as 3, but contents are user defined.

For more information, refer to the RTE-6/VM Programmer's Reference Manual or the RTE-A Programmer's 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 to be associated with the logical file, the file is created using the type and size specified. If the type is missing, it will default to 3, and if the size is missing, it will default to 24 blocks or 50 records (whichever is larger).

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 associate a physical file with a type other than 1 or 2, an error will occur.

If the physical file does not exist, and a file name, instead of an LU number, is to be associated with the logical file, the file is created using the type and size specified. If the type is missing it will be determined by the component size, and if the size is missing, it will default to 24 blocks or 50 records (whichever is larger).

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

Notes:

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

```
*** Pascal I/O Error On File xxxx
File Cannot Be Type 1 Or 2
```

2. With 24 blocks or the number of blocks necessary to hold 50 records, whichever is larger.
3. If the size of the component type (which is the record length) is 128 words, the type is 1, otherwise, the type is 2.
4. The following error will occur:

```
*** Pascal I/O Error On File xxxx
File Must Be Type 1 Or 2
```

Text Files

Any FMP file type can be used as a text file. Type 1 and 2 files can be opened as text files for reading if the line-size of the logical file is compatible with the record size of the physical file; writing to such files will cause a run time error.

If the physical file does not exist, and a file name, instead of an LU number, is to be associated with the logical file by rewrite or append, the file is created using the type and size specified. If the type is missing, it will default to a 3, and if the size is missing, it will default to 24 blocks.

Interactive File I/O

The following table illustrates the differences in program operation when input is taken from an interactive device (terminal) with respect to programs loaded with the FMGR only library (PASCAL_FMGR.LIB or PASCAL_FMGR_ALT.LIB or \$PLIB or =PLIB) and programs loaded with the file system library (PASCAL.LIB or \$PLIBN or PASCAL_CDS.LIB or \$PLIBC).

	FMGR only library	File system library
reading n characters (where n is odd)	n+1 characters are seen by the program (last character is blank)	n characters are seen by the program.
carriage return with no preceding characters	interpreted as end-of-file.	interpreted as an empty line.

Note that Cntl-D is interpreted as end-of-file in both modes.

Closing Files With PURGE

When a file which was opened with the 'SHARED' option is closed with the 'PURGE' option, the actual behavior is described by the following table:

File Location	Pascal Library	Open Once	Error	Exists
File System	File System	No	-8	Yes
File System	File System	Yes	None	No
FMGR	File System	No	None	No
FMGR	File System	Yes	None	No
FMGR	FMGR	No	None	Yes
FMGR	FMGR	Yes	None	Yes

Legend

File Location

- File System — the actual file is on a file system volume
- FMGR — the actual file is on a FMGR cartridge

Pascal Library

- File System — program was loaded with one of
 - PASCAL.LIB
 - \$PLIBN
 - PASCAL_CDS.LIB
 - \$PLIBC
- FMGR — program was loaded with one of
 - PASCAL_FMGR.LIB
 - \$PLIB
 - PASCAL_FMGR_ALT.LIB
 - =PLIB

Open Once

- Yes — the actual file is open shared only once
- No — the actual file is open shared more than once

Error

- error reported on the close with 'PURGE'

Exists

- Yes — the actual file still exists after the close with 'PURGE'
- No — the actual file does not exist after the close with 'PURGE'

Naming Restrictions

No level-1 routine name (or ALIAS) or program name (or ALIAS) or module name (or ALIAS) should be A or B or begin with @. MACRO errors will occur if such names are used.

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 (non EMA_VAR)	direct	LDA g
Global (EMA_VAR)	long,I	JSB .LBPR DEF dd LDA B,I
Local	direct	LDA L
Non-local (non-CDS)	direct	LDA n1
Non-local (CDS) and local if offset > 1023 from Q (CDS)	display	LDA display+N ADA =Doffset LDA A,I
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.

display — variable is accessed indirectly through a display table.

short,I — variable is accessed indirectly through a 1-word address.

long,I — variable is accessed indirectly through a 2-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 it is known that all of the actual parameters passed to a VAR formal parameter, will not be in VMA/EMA, turning HEAPPARMS OFF will save access time and code space (see HEAPPARMS in Appendix C).
- If an actual PAC parameter and its corresponding formal parameter differ in length then the actual parameter is converted into a variable of the type of the formal parameter. This is less efficient than passing a PAC 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 :R :M	none none none	short short short	none a1 none	short,I short,I short,I	none rad none
	Func result	none	short	none	short,I	none
	Heap 2	Value:1 :2	cvtl	short	ctl,rc	direct
			ctl	short	ctl,rc	direct
	Var:1N :1R :2N :2R :MN :MR	none none none none f1wa f1wa	stlm stlm stl stl short short	c2wa c2wa,a2 c2wa c2wa,a2 none m2wa,a2	long,I long,I long,I long,I short,I long,I	none rad none rad none rad
	Func Result:N :R	none	short	none	short,I	
		none	short	m2wa,a2	long,I	

CAUTION

* = non-CDS environment only. For programs running in the CDS environment, the :N (non-recursive) cases should be used for both recursive and non-recursive routines, the :R (recursive) cases apply only to recursive routines in the non-CDS environment.

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
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)
 a1 — 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 0 : PROGRAM pack;
2 0 : TYPE
3 0 :   RANGE      = 0..3;
4 0 :
5 0 : {Declare Packed and Unpacked Array Types}
6 0 :   INDEX      = 1..16;
7 0 :   ARR        = ARRAY [INDEX] OF RANGE;
8 0 :   PACKED_ARR = PACKED ARRAY [INDEX] OF RANGE;
9 0 :
10 0 : {Declare Packed and Unpacked Record Types}
11 0 :   REC = RECORD {with eight one-word fields}
12 1 :     first,
13 1 :     second,
14 1 :     third: RANGE;
15 1 :     ar: ARRAY [1..5] OF RANGE;
16 0 :   END;
17 0 :
18 0 :   PACKED_REC = PACKED RECORD {with eight two-bit fields}
19 1 :     first,
20 1 :     second,
21 1 :     third: RANGE;
22 1 :     ar: PACKED ARRAY [1..5] OF RANGE;
23 0 :   END;
24 0 :
25 0 : VAR
26 0 :   a: ARR;           {16 words }
27 0 :   pa: PACKED_ARR;  { 2 words }
28 0 :   r: REC;          { 8 words }
29 0 :   pr: PACKED_REC; { 1 word  }
30 0 :   i: INDEX;        { 1 word  }
31 0 :   x: RANGE;        { 1 word  }
32 0 :
33 0 1 BEGIN $LISTCODE, RANGE OFF$           {Variable offsets for unpcked structures}
34 0 1 x := a [i];
35 0 1 {Variable offsets for packed structures}
36 0 1 x := pa [i];
37 0 1 {Constant offsets for unpacked structures}
38 0 1 x := a [7];
39 0 1 x := r.second;
40 0 1 x := r.ar [5];
41 0 1 x := pa [7];
42 0 1 x := pa [8];
43 0 1 x := pr.second;
44 0 1 x := pr.ar [5];
45 0 1 $LISTCODE OFF$                         { }
46 0 1 END.

```

```
00057 * Line #35
00057 LDA .400
00060 ADA Pas.1+27
00061 LDA A.
00062 STA Pas.1+28
00063 * Line #37
00063 CCA
00064 ADA Pas.1+27
00065 STA Pas.1+29
00066 JSB Pas.BitExtract1
00067 DEF *+6
00070 DEF .300+2
00071 DEF Pas.1+16
00072 DEF Pas.1+29
00073 DEF .300+1
00074 DEF .300+1
00075 STA Pas.1+28
00076 * Line #39
00076 LDA Pas.1+6
00077 STA Pas.1+28
00100 * Line #40
00100 LDA Pas.1+19
00101 STA Pas.1+28
00102 * Line #41
00102 LDA Pas.1+25
00103 STA Pas.1+28
00104 * Line #43
00104 LDA Pas.1+16
00105 LSR 2
00106 AND =D3
00107 STA Pas.1+28
00110 * Line #44
00110 LDA Pas.1+16
00111 AND =D3
00112 STA Pas.1+28
00113 * Line #45
00113 LDA Pas.1+26
00114 LSR 12
00115 AND =D3
00116 STA Pas.1+28
00117 * Line #46
00117 LDA Pas.1+26
00120 AND =D3
00121 STA Pas.1+28

0 Errors detected.
0 Warnings issued.
48 Source lines read.
125 Words of program generated.
```

Heap 1 vs. Heap 2 (Non-CDS)

Under Heap 2 the access times are slower and more code is emitted to access:

- dynamic variables.
- VAR parameters, except for non-recursive 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 VMA/EMA. Stack access is done with double integer arithmetic and VMA/EMA addressing routines.

VMA access time is the same as that of EMA, if the heap is entirely resident in the working set. The amount of degradation in performance as the working set gets smaller is dependent on the way the particular application accesses the heap.

Shareable EMA access time is the same as EMA access time.

There can be much more memory available for heap and stack with Heap 2.

Heap 2 — (EMA) amount of EMA space in EMA partition (up to 1.8 Mb).

(VMA) amount of VMA space associated with the program (up to 128 Mb).

Heap 1 — amount of memory left between the end of the program and the end of the partition (64 kb
— op sys — program size).

EMA partitions (RTE-6/VM) are generally scarcer than normal partitions (RTE-A dynamically allocates partitions, so this is less of a concern). 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.

Heap 1 vs. Heap 2 (CDS)

Under Heap 2 the access times are slower and more code is emitted to access:

- dynamic variables.
- VAR parameters, except for routines with \$HEAPPARMS OFF\$

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.

Heap 2 Heap routines are larger than the Heap 1 versions.

VMA access time is the same as that of EMA, if the heap is entirely resident in the working set. The amount of degradation in performance as the working set gets smaller is dependent on the way the particular application accesses the heap.

Shareable EMA access time is the same as EMA access time.

There can be much more memory available for heap and stack with Heap 2.

Heap 2 — (EMA) amount of EMA space in EMA partition (up to 1.8 Mb).

(VMA) amount of VMA space associated with the program (up to 128 Mb).

Heap 1 — memory is taken from the hardware stack area as needed. It is at most (64 kb – base page – global data area – non-CDS routine area – stack area currently in use – image area if any). Allocation of heap area reduces the available hardware stack area and hence the depth to which routines can be called.

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
- string 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)
- string length and indexing

Some expressions/operations, notably some involving sets and strings are always range checked, whether or not RANGE is ON. See Appendix D for information on disabling run time checks on some string operations.

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 Statement

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 Statement

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 Statement

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 in the non-CDS environment are more costly in terms of space and execution time than non-recursive routines. In the CDS environment there is no space or time penalty for recursion.

Space Considerations

Recursive routines in the non-CDS environment are allocated a few more words of space which are used to save the environment of a current activation.

Extra code from the run-time library is loaded with the program to handle recursion in the non-CDS environment. (This code is always the same size, regardless of the number of recursive routines in the program.)

In the CDS environment, no extra space is allocated and no library routines are loaded to handle recursion.

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 in both the non-CDS and CDS environments.

Time Considerations (CDS)

There is only one routine activation method in the CDS environment, the PCAL instruction.

PCAL execution time is a function of the number of parameters to the routine and whether or not the called routine is resident in the currently active segment.

Refer to the appropriate processor reference manual for PCAL timing information.

Time Considerations (Non-CDS)

Routine activation methods in the non-CDS environment 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 Sequences (Non-CDS)

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 DIRECT calling sequence is not available in the CDS environment.

The savings are 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 as shown in the table below.

Table 8-7. Overhead Times for Routines with .ENTR vs. \$DIRECT\$ Calling Sequences
(in microseconds, Non-CDS)

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

NOTE: These times are for firmware .ENTR only. The times for software .ENTR (found on M, L, and XL CPUs) are much greater.

.ENTR overhead =

$$10.2 + (\text{number of parameters} * 2.5) \text{ microseconds code space} = 2 \text{ words}$$

\$DIRECT\$ overhead =

$$1.5 + (\text{number of parameters} * (4.1 + \text{indirect levels} * 3.4)) \text{ code space} = 1 + (\text{number of parameters} * 5) \text{ words}$$

FMP vs. Pascal I/O

FMP and Pascal I/O are 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:

- Use the procedure Pas.DcbAddress1 or Pas.DcbAddress2 to obtain a pointer to the DCB to use in the FMP calls.
- 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. This module is called either PASCAL_ERR.REL or %PRERS for programs in the non-CDS environment, and PASCAL_CERR.REL or %PRERC for programs in the CDS environment. This module must be relocated in the program before the appropriate library is searched.

To save space in your loaded program, short versions of the heap management routines can be used by specifying the compiler option \$HEAP_DISPOSE OFF\$. They are smaller because they do 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

This section refers to segmentation (single or multi-level) in the non-CDS environment.

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 B) for details.

Putting Globals in the Heap

Large or infrequently-used globals can be put in VMA/EMA with the EMA_VAR option (see Appendix C). This makes more room in the 32KW program partition in the Non-CDS environment or the 32KW data partition in the CDS environment.

Structured Constants

Using structured constants can save time and space in several ways:

In the non-CDS environment 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.

In the CDS environment, execution time is spent copying a structured constant from code space to data space on routine entry if the CODE_CONSTANTS ON option is used. The cost in execution time may be acceptable since data space is only in use when the routine is active. If CODE_CONSTANTS OFF is used, data space is permanently allocated to the constant, for the duration of the program, whether or not the routine is active (however no execution time is spent copying the constant on routine entry).

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 (in the Non-CDS environment), so that you do not pay a large space penalty for the existence of both a variable and a structured constant.

Structured constants can improve on CASE statements which serve to map one data type onto another. For example, the functions below are equivalent:

Implementation Considerations

```
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.

Pascal programs can be developed either under the File System (FS Mode), or under the old FMGR System (FMGR Mode). The following sections describe program development using the two different modes.

Relevant Files

Installation instructions and aids in tuning the compiler can be found in the following Pascal/1000 Configuration Guide.

92833-90003 Pascal/1000 Configuration Guide for RTE-6/VM and RTE-A.

The following is a list of files used in compiling, loading, and running Pascal programs. (Not all are needed for all installations.)

(FS)	(FMGR)	Description
Compiler programs		
/PROGRAMS/PASCAL.RUN	PASCL	Pascal monitor
/PROGRAMS/PASCOMP.RUN	PCL	Pascal compiler
Std FS Libraries		
/LIBRARIES/PASCAL.LIB	\$PLIBN	Pascal FS Library
/LIBRARIES/PASCAL_ERR.REL	%PRERS	Short Error Library
/LIBRARIES/PASCAL_TRA.REL	%TRCAS	Trace Library A
/LIBRARIES/PASCAL_TRB.REL	%TRCBS	Trace Library B
/LIBRARIES/PASCAL_TRC.REL	%TRCCS	Trace Library C
/LIBRARIES/PASCAL_LH2.REL	%PLDH2	LOADR/MLLDR dummies
/LIBRARIES/SHSLB.LIB	\$SHSLB	Short Heap/Stack Lib
CDS FS Libraries		
/LIBRARIES/PASCAL_CDS.LIB	\$PLIBC	Library (New files)
/LIBRARIES/PASCAL_CERR.REL	%PRERC	Short Error Library
/LIBRARIES/PASCAL_CTRA.REL	%TRCAC	Trace Library A
Std FMGR Libraries		
/PASCAL/PASCAL_FMGR.LIB	\$PLIB	Pascal FMGR Library
Altered Std FMGR Libraries		
/LIBRARIES/PASCAL_FMGR_ALT.REL	=PLIB	Altered Pascal FMGR Library
/LIBRARIES/PASCAL_ERR_ALT.REL	=PRERS	Altered Short Error Library
/LIBRARIES/SHSLB_ALT.LIB	=SHSLB	Altered Short Heap/Stack Library
Other Programs		
/PROGRAMS/ALTER.RUN	ALTER	Convert relocatable to IVB format
Other Files		
/SYSTEM/PASCAL.ERR	"PERRS	Syntax Error List
/SYSTEM/ALTER.DAT	*	Name file for program ALTER

*not supported

Compiling a Program

In order to run a Pascal program, a programmer must first compile the source file into a relocatable file, and then load the relocatable file into a runnable program. This section describes how to invoke the compiler to produce a relocatable file.

The Pascal compiler is executed by issuing the following command:

(FS Mode)

```
Pascal <source> [<list>] [<relocatable>] [<opt>] [<options>]
```

(FMGR Mode)

```
:RU,PASCL,<source> [,<list>] [,<relocatable>] [,<opt>] [,<options>]
```

where each item in the run string is one of the constructs listed after it, below:

<source>

- the name of the file containing Pascal source code (TEST.PAS)
- the source file name without the suffix (TEST implies TEST.PAS)
- the FMGR name of the source file (&TEST)

<list>

- the name of the program listing file (TEST.LST)
- the listing file name without the suffix (TEST implies TEST.LST)
- the FMGR name of the listing file ('TEST')
- the default list file name specifier ('-')
- no-listing specifier ('0')
- empty (implies '0')

<relocatable>

- name of the relocatable code file (TEST.REL)
- the relocatable file name without the suffix (TEST implies TEST.REL)
- the FMGR name of the relocatable file (%TEST)
- the default relocatable file name specifier ('-')
- no-code-generation specifier ('0')
- empty (implies '-')

<opt>

- name of the file containing compiler options (TEST.COP)
- the option file name without the suffix (TEST implies TEST.COP)
- the FMGR name of the option file (*TEST)
- default option file name specifier ('-')
- no-option-file specifier ('0')
- empty (implies '0')

<options>

- compiler options (DEBUG=ON, RANGE=OFF)
- empty

Any of the above *files* may be disc files or devices.

Source File

The source file is the file containing the Pascal source code. When Pascal is executed, the source file must be specified. The file may be a disc file or a device.

Source files must have the *.Pas* extension if they reside in a File System directory, but the extension need not be specified in the runstring for Pascal.

If the file name specifies a disc file, the file:

1. must exist, and
2. (FS Mode): may be either in a directory or on a FMGR cartridge. If it is in a directory, it must have a type extension of *.Pas*
(FMGR Mode): must be on a FMGR cartridge, and
3. must be of type 3 or 4, and
4. must have a readable security code; that is, either
 - a. the existing file security code is negative and is the identical security code specified in <source>, or
 - b. the file security code is non-negative. In this case, the security code in <source> need not be specified, and if specified, need not match that of the existing file.

If the file name specifies a device, the device:

1. must exist, and
2. must be readable

Listing File

The listing file is the file generated by the compiler that contains:

1. a listing of the program,
2. syntax errors detected by the compiler,
3. symbol tables of the program procedures, variables, etc. (if the TABLES option was specified),
4. intermixed listing of Pascal source and generated machine code (if LIST_CODE was specified),
5. a compilation summary (if STATS was specified)

Compilations usually result in purging the existing listing and creating a new one. Certain checks are performed before purging listing files to avoid accidentally purging unrelated files.

An existing listing file will be purged if the following are true: (if any are false, an error is generated, the file is not purged, and the compilation is aborted)

1. <list> is an ASCII file (type 3 or 4).
2. (FS Mode): <list> has no extension, or the extension *.Lst*. <name> or <name>.Lst may be specified; <name> implies <name>.Lst
(FMGR Mode): <list> has the default list name derived from <source>
3. <list> does not specify the same file as <source>
4. no FMP error is encountered while trying to purge the file

If not specified, or if LU 0 is specified, the listing is suppressed, except for the heading and source lines containing errors, which will be printed at the user's terminal.

'Pascal Test' is equivalent to 'Pascal TEST 0'

The list file is opened exclusively, prohibiting access to it during compilation.

Assembly File

The assembly file is the file generated by the compiler that contains the assembly code for the program. This file is automatically passed to the assembler, which creates the program's relocatable file.

Most of the time compilations will result in creating an assembly file, and then purging it after the assembler has created the relocatable. If the assembly file exists beforehand, then checks are performed before purging assembly files to avoid accidentally purging unrelated files.

An existing assembly file will be purged if the following are true: (if any are false, an error will be generated, the assembly file will not be purged, and the compilation will be aborted):

1. the assembly file is an ASCII file (type 3 or 4)
2. the file is not the same as <source> or <list>
3. no FMP error is encountered while trying to purge the file

The assembly file that the compiler creates is purged after the assembler has executed, unless:

1. the assembler terminated abnormally, or
2. the KEEPASMB compiler option is ON

If the assembly file is not purged, then the following message will appear on the user's terminal:

```
Pascal: Assembly source kept in file <name>
```

Relocatable File

The relocatable file is the file generated by the assembler, which contains the relocatable object code for the program. In order to run the program, this file must first be loaded by one of the system loaders.

Most of the time compilations will result in purging the existing relocatable and creating a new one. Certain checks are performed before purging relocatable files to avoid accidentally purging unrelated files.

An existing relocatable file is purged if the following are true: (if any are false, an error will be generated, the file will not be purged, and the compilation will be aborted).

1. the file is a relocatable (type 5)
2. (FS Mode): <relocatable> has an extension of *.Rel*, or no extension <name> or <name>.Rel may be specified; <name> implies <name>.Rel
(FMGR Mode): <relocatable> has the default name, derived from <source>
3. <relocatable> is not the same file as <source> or <list>
4. no FMP error is encountered while trying to purge the file

If LU 0 is specified, the relocatable file is not created (nor is any existing relocatable file purged). Code generation, in fact, will not be done at all, which will result in a relatively fast syntax check of the program.

If not specified, then '-' is implied for the relocatable file.

These five commands are equivalent:

```
Pascal test
Pascal test 0 -
Pascal test 0 test.rel
Pascal test.pas 0 test.rel
Pascal test 0 test
```

NOTE

Some syntax errors are not detected/reported if code generation has been suppressed. Errors numbered 300 through 350 will not be reported, other errors may not be detected as well. Code generation can be suppressed by specifying 0 (zero) as the relocatable file name or by turning the CODE option OFF. Code generation is automatically suppressed following a procedure or function which contains any syntax errors.

Option File and Runstring Compiler Options

Compiler options can be specified for a given compilation without editing the source by either including them in an option file, or by specifying them directly in the runstring. Neither the option file nor runstring options are required. Options may appear with or without the \$ delimiters. Since CI normally replaces blanks with commas, an "=" can be used between option names and their arguments on the runstring, or the options and their arguments can be quoted to preserve blanks using the CI quoting mechanisms.

The only option that is specifically not allowed in the option file or on the runstring is the INCLUDE option.

WORK and RESULTS are options that, if specified, can only appear in the option file or on the runstring.

Example

(FS Mode)
Pascal test.pas 1 - opt.cop results='1'

(FMGR Mode)
:RU,PASCL,&TEST,1,-,*OPT,RESULTS '1'

where opt.cop and *OPT are files containing the lines

\$LINES 75, RANGE DNS
\$HEAP 1\$

If the option file <opt> is specified, the following must be true:

1. <opt> must exist
2. <opt> must be an ASCII file (type 3 or 4)
3. FS Mode: <opt> must have an extension of .Cop, or no extension <name> or <name>.Cop may be specified; <name> implies <name>.Cop If '-' is specified when compiling Test.Pas, then <opt> must be Test.Cop

FMGR Mode: <opt> may be any FMGR file name If '-' is specified when compiling &TEST, then <opt> must *TEST

4. <opt> must not be the same file as <source>, <list>, <relocatable> or the assembly file
5. <opt> must have a readable security code; that is, either
 - a. the existing file security code is negative and the identical security code was specified in <opt>, or
 - b. the existing file security code is non-negative. In this case, the security code in <opt> need not be specified, and if specified, need not match that of the existing file.

If <opt> specifies a device, that device:

1. must exist, and
2. must be readable

The option file options and runstring options do not *override* any options specified in the source file, but instead are applied in a certain order. The order in which options take effect is:

1. option file,
2. runstring options,
3. source file

Thus, runstring options may be used to:

1. specify options that are not specified in the option or source files,
2. *override* options that were specified in the option file, subject to the normal rules for respecifying options. That is, certain options may be specified any number of times, while others may be specified only once. For example, if the option file TEST.COP contains the line:

`$DEBUG, HEAP 2$`

then

```
Pascal TEST 0 - TEST.COP DEBUG=OFF {ok, can respecify DEBUG}
Pascal TEST 0 - TEST.COP HEAP=1    {not ok, can't respecify HEAP}
```

Use of Default File Names

A - may be used to specify a default file name for either the listing, relocatable, or option file. The compiler will construct a name for the file based on the source file name and the table below:

File:	(FS Mode)	(FMGR Mode)	Type	Size
	Extension	1st character		
Source	Pas	&	na	na
Listing	Lst	'	3	24
Relocatable	Rel	%	5	24
Assembly	Mac	^	3	24
Option	Cop	*	na	na

The name of the file is constructed as follows:

File name: (FS Mode)

The file name will be the same as the source file name; the extension will be as defined in the table above.

(FMGR Mode)

The file name will be the same as the source file name, with the initial character replaced as defined in the table above. If the source file name does not begin with '&', then each character of the name is shifted one space to the right, and the initial character is defined in the table.

Directory: (FS Mode only)

The file will be placed in same directory as the source file, unless overridden with a directory specification (e.g. /dir/-).

Cartridge: (FMGR or FS Mode)

The file will be placed on the same cartridge as the source file, unless overridden with a cartridge specification (e.g. -::CR) or a directory specification (e.g. /dir/-).

Security:

The security code is 0 unless it is specified.

Type:

The listing and assembly files will be type 3, and the relocatable file will be type 5.

Size:

The listing, assembly, and relocatable files will be created with a size of 24 blocks, unless overridden.

Other rules:

If the default specification is simply - without any cartridge or directory reference, then the default will be expanded from the source name with either:

- a. the extension changed if the source is in a directory,
(e.g. test.pas - test.lst)
- b. the prefix changed (or added) if the source is on a FMGR cartridge,
(e.g. &test::CR - 'test::CR)

If the specification contains directory or sub-directory specifications, then an attempt will be made to open the directory in which the file does or will exist. One of the following will occur:

- a. If the directory can be opened, then the default will be expanded using the appropriate extension,
(e.g. test.pas /dir/- /dir/test.lst)
- b. If the directory cannot be opened, and there are no apparent sub-directories, then the default will be expanded into the FMGR form. (Note that the resulting name will be truncated to 6 characters),
(e.g. test.pas /no/- /no/'t)
- c. If the directory cannot be opened, and there are apparent sub-directories, then an error will be reported,
(e.g. test.pas /not1/not2/- ERROR)

How to Use Pascal/1000

For example, the following pairs of runstrings are equivalent:

```
Pascal test1
Pascal test1.pas 0 - 

Pascal test1 - - -
Pascal test1.pas test1.lst test1.rel test1.cop

Pascal /tests/test1.pas - -
Pascal /tests/test1.pas /tests/test1.lst /tests/test1.rel

Pascal /source/test1.pas /list/- /obj/-
Pascal /source test1.pas /list/test1.lst /obj/test1.rel

Pascal test1.pas - -:sc:::300
Pascal-test1.pas test1.lst:0::3:24 test1.rel:sc:5:300

PASCL,&TEST::C1,-,-:SC:C2:5:100,-
PASCL,&TEST::C1,'TEST::C1:3:24,%TEST:SC:C2:5:100,[TEST::C1
```

Alternate Compiler Designation

You may want to use more than one version of the compiler on your system (for example, a CDS and non-CDS version, or an old and new revision, etc). This can be accomplished by naming the alternate versions of PASCOMP.RUN as PASCOMP<some string>.RUN, and then entering the command:

```
Pascal +<some string> <source> <list> <relocatable> ...
```

For example, a CDS version (PASCOMPC.RUN) and a non-CDS version (PASCOMP.RUN) can be executed on the same system:

```
Pascal test 0 - {Schedules PASCOMP.RUN}
Pascal +c test 0 - {Schedules PASCOMPC.RUN}
```

NOTE

This only works for revision 2401, or later, compilers.

Compiler Workspace

The RTE-6/VM and RTE-A Pascal compilers keep their entire symbol table and other work areas in VMA. When the VMA area is exhausted, the compiler aborts with an appropriate message. These error messages, as well as details on adjusting compiler workspace area and performance are discussed in the Pascal Configuration Guide.

The Listing

The listing produced by the Pascal/1000 compiler for the source program below appears on the next page. (The program contains errors.)

```
$DEBUG$  
PROGRAM sample (output);  
  
CONST  
    number_of_disks = 4;  
  
PROCEDURE move (n : INTEGER;  s, d, i : CHAR);  
BEGIN  
    IF (n > 0) THEN BEGIN  
        move (n-1, s, i, d);  
        writeln (' Move disk', n:3,' from ', a, ' to ', b);  
        move n-1, i, d, s);  
    END;  
END END;  
  
BEGIN  
    writeln (' Tower of Hanoi solution for',  
            num_disks:3, ' disks:' );  
    writeln;  
    move (number_of_disks, 'A','C' 'B');  
END.
```

Pascal/1000
Ver. 2/2401Thu May 17, 1984 9:19 am
SAMPLE Page 1

```

1 0 : $DEBUG
2 0 : PROGRAM sample (output)
3 0 :
4 0 : CONST
5 0 :     number_of_disks = 4;
6 0 :
7 0 :
8 1 : PROCEDURE move (n : INTEGER; s, d, i : CHAR);
9 1 1 BEGIN
10 1 2     IF (n > 0) THEN BEGIN
11 1 2         move (n-1, s, i, d);
12 1 2         writeln (' Move disk', n:3, ' from ', a, ' to ', b) ^104 ^104
0 ****
13 1 2         move (n-1, i, d, s);
14 1 2     END;
15 1 1 END END;
12 **** ^14,6
16 0 :
17 0 : BEGIN
18 0 :     writeln (' Tower of Hanoi solution for',
19 0 :                 num_disks:3, ' disks:');
15 **** ^104
20 0 :     writeln;
21 0 :
22 0 :     move (number_of_disks, 'A','C' 'B');
23 0 : END.

```

5 Error detected (first 12/last 19).
 0 Warnings issued.
 23 source lines read.
 0 Words of program generated.

Errors in this compilation:

6: Illegal symbol
 14: ';' expected
 104: Identifier not declared

NOTE

Some syntax errors are not detected/reported if code generation has been suppressed. Errors numbered 300 through 350 will not be reported, other errors may not be detected as well. Code generation can be suppressed by specifying 0 (zero) as the relocatable file name or by turning the CODE option OFF. Code generation is automatically suppressed following a procedure or function which contains any syntax errors.

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 runstring, without the security code. 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 next two numbers on the line are nesting levels: the first is the procedure nesting level, and the second is the statement nesting level (BEGIN...END, REPEAT...UNTIL, etc). These can be useful in detecting missing BEGIN, and END symbols in programs with syntax errors.

Lines containing errors are followed by error lines. Lines 12, 15, and 19 in the previous example contained errors. For a detailed explanation of error lines refer to the Errors section in this Chapter.

The next few 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 (and data for CDS programs) that were generated.

At the end of the listing, the numbers and descriptions of the errors encountered during the compilation are listed.

The compiler options STATS, TABLES, CODE_OFFSETS, and CODE_INFO can be used to send additional information to the list file (refer to Appendix D).

AUTOPAGE can be used to automatically page eject the listing file after each procedure body. LINE_INFO can be turned OFF to suppress the nesting level information printed on each line. LIST can be turned ON or OFF to control the printing of any information to the list file. PAGE can be used to page eject the listing file. LIST_CODE can be used to display the generated code.

Loading a Pascal Program

Loading a Pascal program consists running the loader, named LINK, with Pascal relocatable files, Pascal and other libraries, and various options to LINK. For simple programs, the parameters (files and options) can be called out in the runstring to LINK. For other programs, a link command file can be constructed with the editor and passed to LINK.

The relocatables used in the load process are 1) the main program relocatable, and 2) all other relocatables containing any routine that may be called during program execution.

The libraries used may be implicitly-searched libraries (those that have been generated into the system), and/or explicitly-searched libraries (those that are given to LINK explicitly).

The Pascal libraries used in the load process are 1) a primary Pascal library, and 2) some special-function relocatables that may be used in certain situations.

The primary Pascal library used depends upon the environment in which the program is to run, and is one of the following files:

One and only one of these should be searched when linking a particular Pascal program. It is recommended that you generate into your system the one that you will use the most often.

With RTE-6, you would normally generate Pascal.Lib into your system. With RTE-A, you would normally call out Pascal.Lib in the non-CDS library section, and Pascal_CDS.Lib in the CDS library section.

- Cds/FS Library: Pascal_Cds.Lib {aka \$PLIBC}

This library is used with programs that:

- have been compiled with \$CDS ON\$
- need to access File System and/or FMGR files

- Std/FS Library: Pascal.Lib {aka \$PLIBN}

This library is used with programs that:

- have been compiled with \$CDS OFF\$
- need to access File System and/or FMGR files

- Std/FMGR Library: Pascal_FMGR.Lib {aka \$PLIB}

This library is used with programs that:

- have been compiled with \$CDS OFF\$
- need to access only FMGR files

It provides no capability beyond Pascal.Lib, but will result in a smaller loaded program. It is provided for those programs which do not need access to the new file system, and would be too large if loaded with the new file system software.

- Std/FMGR/Altered Library: Pascal_FMGR_ALT.Lib {aka =PLIB}

This library is used with programs that:

- have been compiled with \$CDS OFF\$
- need to access only FMGR files
- have been Altered (run through ALTER)

It is provided for those who want to develop RTE-IVB or RTE-IVE programs on an RTE-A or RTE-6/VM system.

The special-function relocatables fall into three categories:

Cds versions of these relocatables should be used with CDS programs.

Std versions should be used with non-CDS programs.

Alt versions should be used with ALTERed relocatables (relocatables that have been run through the program ALTER, for loading onto an RTE-IVB or IVE system).

Some of the following are no longer necessary (*'d files), but are provided for compatibility with relocatables created by previous versions of the compiler.

- Short-error handler: used to reduce code space of the loaded program. It prints brief instead of full runtime error messages.
 - Cds version: Pascal_Cerr.Rel (aka %PRERC)
 - Std version: Pascal_Err.Rel (aka %PRERS)
 - Altered: Pascal_Err_Alt.Rel (aka =PRERS)
 - Tracers: used to trace procedure invocation and exit (Trace A) and show a procedure traceback when a runtime error occurs (Trace B and Trace C).
 - Cds version:
 - Trace A Pascal_Ctra.Rel (aka %TRAC)
 - Std versions:
 - Trace A Pascal_Tra.Rel (aka %TRCAS)
 - Trace B Pascal_Trb.Rel (aka %TRCBS)
 - Trace C Pascal_Trc.Rel (aka %TRCCS)
 - Short Heap-Stack Lib: a smaller, faster heap manager that can be used when the full functionality of dispose (free-list management, etc.) is not needed. This is for relocatables created by pre-2401 compilers. The short heap manager in the newer compilers is selected at compile time by the compiler option \$HEAP_DISPOSE OFF\$
- * • Std version: Shslb.Lib (aka \$SHSLB)
 * • Altered: Shslb_Alt.Lib (aka =SHSLB)

Loading a Simple Pascal Program

The Pascal program, in the file "hello.pas",

```
PROGRAM hello (output);
BEGIN
  writeln ('hello');
END.
```

can be compiled, loaded, and executed with the following command sequence:

```
CI> pascal hello
CI> link hello.rel
CI> hello
```

This will create either a:

Std version of hello, if the compiler's code generation default is \$CDS OFF\$, and Pascal.Lib is generated into the system, or a

Cds version of hello, if the compiler's code generation default is \$CDS ON\$, and Pascal_CDS.Lib is generated into the system.

If the compiler and library are set up to create Std programs by default, a CDS version of the program can be made by using the commands:

```
CI> pascal hello 0 -,,cds
CI> link hello.rel pascal_cds.lib
CI> hello
```

If the compiler and library are set up to create CDS programs by default, a Std version of the program can be made by using the commands:

```
CI> pascal hello 0 -,,cds=off
CI> link hello.rel pascal.lib
CI> hello
```

For loading more complex programs (segmented, EMA/VMA, shared EMA, etc), consult the LINK User's Manual for either RTE-A or RTE-6.

Load-Time Errors

There are several errors that may occur while loading that are unique to Pascal. These are:

Undefined entry point Pas.<number>

The entry points Pas.1, Pas.2, etc. are used for the proper operation of separate compilation. If LINK reports one of these as being undefined, the following table may be helpful in determining the cause. The problem is usually due to objects being referred to from subprograms or modules that are assumed to be, but are not, in the main program. (Note that these symbols are reserved by Pascal and are subject to change.)

SYMBOL	USE	ACTION
Pas.1	Global variable data area.	Make sure all compilation units use the same global declarations.
Pas.2	Global string constants.	Same as above.
Pas.3	Global structured constants.	Same as above.
Pas.4	Predefined file <i>input</i> .	Declare <i>input</i> in main's program heading, or check for inadvertent use of <i>input</i> , e.g. <i>readln (x);</i> <i>readln (input, x);</i>
Pas.5	Predefined file <i>output</i> .	Declare <i>output</i> in main's program heading, or check for inadvertent use of <i>output</i> , e.g. <i>write (x);</i> <i>writeln;</i> <i>write (output, x);</i>
Pas.nnn	Statement label defined at the global level.	Make sure subprograms and main program agree on the name and number of global LABELs.
<module_name>.1 <module_name>.2 <module_name>.3 <module_name>.4 <module_name>.5	Module globals String constants Structured consts File initialization File closing	Undefined errors with these names are caused by changing a module without recompiling all modules that import it. Recompile all modules that import <module_name>, either implicitly or explicitly.

Duplicate entry point: Pas.CDSCONFLICT

This happens when both CDS and non-CDS Pascal routines are relocated into the same program. Pascal routines must be all CDS or all non-CDS.

Undefined entry point: Pas.CDS (or Pas.NONCDS)

This happens when the wrong Pascal library is searched for a particular relocatable, i.e. either searching the CDS library for a Std relocatable, or vice-versa. (If you did not explicitly search any Pascal library, then the Pascal library generated into the system does not match your relocatable.) To fix this error, link the program again searching the proper Pascal library.

Running a Pascal Program

After a program has been successfully loaded, it is in executable form, and can be run by entering the command:

```
CI> <program name> <program parameters>
```

where:

<program name> is, by default, the name appearing in the PROGRAM heading of the main program.

<program parameters> are identifiers, numbers, characters, etc. that are to be passed to the program. The first parameters are the names of the file you wish to associate with the logical files appearing in the program heading (see Chapter 6). Other parameters may be passed to the program and *picked up* explicitly with calls to Pas.SParameters or Pas.Parameters from within the Program (see Appendix F).

Example

If the program heading is:

```
PROGRAM test (input, output);
```

then the program can be executed with the commands:

```
CI> test 1 1
```

Input will be read from, and output will be written to LU 1, the user's terminal.

```
or CI> test infile outfile
```

Input will be read from the file *infile* and output will be written to the file *outfile*.

The predefined files "input" and "output" are treated specially. If no parameter is given, then LOGLU (usually LU 1) is assumed. For the program above, then, these two commands are equivalent:

```
CI> test  
CI> test 1 1
```

Run-Time Errors

When the program is executed, an error may be detected by Pascal, FMP, or RTE which will cause an error message to be printed to the terminal, and the program to abort. Pascal-detected errors are described in Appendix A. Other errors are described in appropriate RTE Programmer's Reference Manual and User's Guide.

Debugging a Pascal Program

Several tools are available for determining program failure:

Debug/1000

Procedure Tracing

Trace Library A

Procedure Traceback

Trace Library B

Trace Library C

Pas.TraceBack

Mixed Listing

These are described in the section of Debug/1000.

Debug/1000

The most useful tool in debugging Pascal programs is the Debug/1000 product. To be used with Pascal, the program must be compiled with the \$DEBUG\$ option, and loaded with the +DE option.

Debug/1000 allows you to interactively single-step, set breakpoints and conditional breakpoints, examine and modify program variables, and profile the program. (Refer to the Debug/1000 Reference Manual for further capabilities and information.)

Example: File test.pas contains:

```
$DEBUG ON$  
PROGRAM test (input, output);  
  
VAR  
  i,j: INTEGER;  
  
FUNCTION add (first, second: INTEGER): INTEGER;  
BEGIN  
  add := first + second;  
END;  
  
BEGIN  
  readln (i, j);  
  writeln (i:1, ' + ', j:1, ' = ', add (i,j):1);  
END.  
  
CI> Pascal test  
CI> Link test.rel +de  
CI> Debug test  
  
DEBUG> s          {Single step from readln to writeln line}  
10 20          {Enter two numbers to add}  
DEBUG> d i j    {Display values of i and j}  
DEBUG> m i 40    {Modify value of i to be 40}  
DEBUG> s i        {Step Into the function add}  
DEBUG> d first second {Display values of first and second}  
DEBUG> s          {Execute the addition and assignment}  
DEBUG> d add      {Display value of the function result (60)}  
DEBUG> p          {Proceed; allow program to continue}
```

Pascal/Debug Limitations

There are several operations that are not currently supported through Debug/1000. These are:

Debug/1000 cannot display/modify:

- Variables in VMA
- Set variables (the entire set may be displayed, and picked apart by hand)
- Record fields by name (the record itself may be displayed, and the fields apart by hand)
- Constants
- Bit fields within packed arrays and records
- File Buffer Variable (e.g. input^)

Procedure Tracing

By using the Trace A library, procedure invocations and exits can be displayed, as they occur, to a specified Logical Unit.

The program must be compiled with the \$TRACE <Logical Unit Number>\$ compiler option, and then loaded with one of the two relocatables:

(Std programs) Pascal_Tra.Rel

(CDS programs) Pascal_CTra.Rel

Example

File test.pas contains the program:

```
$TRACE 1, CDS OFF$  
PROGRAM test;  
  
PROCEDURE here;  
BEGIN  
END;  
  
BEGIN  
    here;  
END.  
  
CI> pascal test  
CI> link test.rel /libraries/pascal_tra.rel  
CI> test
```

The following information will then be written to the terminal.

```
>      1 Enter: TEST  
>      2 Enter: HERE  
>      2 Exit: HERE  
>      1 Exit: TEST
```

Procedure Traceback

When a program encounters a run-time error, a procedure traceback may be displayed on the terminal to help give you an indication of where the program was, and what it was doing when it aborted. The traceback capability is handled differently for CDS and Std programs:

Std Programs

For Std programs, a traceback is only produced when the program is compiled with the \$TRACE <Logical Unit Number>\$ compiler option, and then loaded with Trace Library B or C. The difference between B and C is N, the number of entries kept in the traceback buffer, and then displayed at program termination. Trace B can be used to minimize the program size, if there is not enough room to load with Trace C.

NAME	N	FILE NAME
Trace Library B	20	/Libraries/Pascal_Trb.Rel
Trace Library C	100	/Libraries/Pascal_Trc.Rel

CDS Programs

CDS programs will always display a procedure traceback when a runtime error occurs. This is done with a call to the library routine *Pas.TraceBack* prior to program termination. There is no limit on the number of routines displayed, as there is in the Std traceback. All routines in the current call chain, back to the main program, are displayed.

Mixed Listing

The LIST_CODE compiler option is used to get a *mixed listing* of a Pascal program. The assembly code generated by the compiler is intermixed in the listing with the corresponding Pascal source lines. This is sometimes useful when low-level debugging must be done, or when you need to know the relative efficiency of different algorithms or constructs.

Pascal/1000
Ver. 2/2401

Fri May 18, 1984 10:57 am
POKE Page 1

```

1 0 : $RANGE OFF$  

2 0 : PROGRAM poke  

3 0 :   (output);  

4 0 :  

5 0 : TYPE  

6 0 :   VAL = 0..7;  

7 0 :  

8 0 :   REC = PACKED RECORD  

9 1 :     a,  

10 1 :     b: VAL;           {2 three-bit fields}  

11 0 :   END;  

12 0 :  

13 0 : VAR  

14 0 :   r: REC;  

15 0 :  

16 1 : PROCEDURE stuff (i: VAL);  

17 1 1 BEGIN  

18 1 1 $LIST_CODE$  

19 1 1   r.a := i;  

20 1 1   r.b := 6;  

21 1 1 $LIST_CODE OFF$  

22 1 1 END;  

00000 .700 EQU *  

00000 * ----- Line #19  

00000   LDA Pas.1  

00001   RRR 13  

00002   AND -D-8  

00003   IOR .200  

00004   RRL 13  

00005   STA Pas.1  

00006 * ----- Line #20  

00006   LDA Pas.1  

00007   AND -D-7169  

00010   IOR -D6144  

00011   STA Pas.1  

23 0 :  

24 0 1 BEGIN  

25 0 1   stuff (5);  

26 0 1 END.  

      0 Errors detected.  

      0 Warnings issued.  

      26 Source lines read.  

      344 Words of program generated.

```

Interfacing Pascal Routines to Other Routines

This section contains information about using Pascal with other subsystems and routines on the HP 1000.

- Calling Non-Pascal Routines from Pascal
- Calling Pascal Routines from Non-Pascal Routines
- Calling FMP Routines from Pascal
- Pascal and FORTRAN
- Pascal and IMAGE
- Calling EXEC from Pascal

Calling Non-Pascal Routines from Pascal

Whenever a routine written in another language is called from a Pascal routine, there must be an EXTERNAL declaration for that routine, including name and parameter list, included in the compilation units which reference that routine.

There are several compiler options that can be used in conjunction with these EXTERNAL declarations to ensure the proper interface with the actual routine. (These are discussed further in Appendix C.)

- ALIAS '<string>':** allows a program to call a routine whose name is not a legal Pascal identifier. For example PROCEDURE get_parm \$ALIAS 'Pas.Parameters'\$...
- DIRECT:** for non-CDS procedures, specifies that the actual or external routine has a *direct* calling sequence instead of a *.ENTR* calling sequence. (DIRECT is allowed for externals in CDS programs for calling existing routines, but DIRECT is ignored for routines declared with CDS on.)
- ERROREXIT:** specifies that an external routine's calling sequence accommodates an error return.
- NOABORT:** specifies that an external routine has a no-abort error return (like EXEC).
- FIXED_STRING ON:** converts a Pascal string parameter to FORTRAN's fixed-string format. (This is necessary to call FMP routines, which are written in FORTRAN.)
- BASIC_STRING ON:** converts a Pascal string parameter to BASIC/1000C format.
- HEAPPARMS OFF:** converts a two-word-addressable VAR parameter to a one-word-addressable parameter. This is necessary when calling, for example, system routines which cannot handle two-word addresses.

Calling Pascal Routines from Non-Pascal Routines

Routines written in other languages can call routines written in Pascal. There are several different procedure-calling and parameter-passing mechanisms supported on the HP 1000. Thus care should be taken to ensure that the generated calling sequence matches the procedure interface set up by Pascal.

Pascal routines have one of the following interfaces, depending on the settings of various compiler options:

Std normal, or *.ENTR* calling sequence (\$CDS OFF\$)

Std direct (\$CDS OFF, DIRECT\$)

CDS normal (\$CDS ON\$)

Calling FMP Routines from Pascal

Most routines in the FMP pacakge of RTE are written in FORTRAN. They are provided in both CDS and non-CDS forms, and may be called from either CDS or non-CDS Pascal programs, with identical EXTERNAL declarations. The subroutine interfaces are the normal CDS or non-CDS interfaces (i.e. no direct calling sequences, etc).

Like other system routines, FMP routines expect only one-word addresses. Therefore, any VAR parameters must be declared with \$HEAPPARMS OFF\$ in compilation units compiled with the \$HEAP 2\$ option.

Since the routines are written in FORTRAN, the strings that FMP expects are in a different format than Pascal strings. In order to pass string parameters to FMP, the compiler option FIXED_STRING must be turned ON for the EXTERNAL declaration. For string parameters returned by FMP, there is one additional requirement. Prior to the call, the current string length of the actual parameter must be set to length of the largest string than FMP can build for the particular parameter. If this isn't done, FMP cannot properly return the parameter. FMP will insert characters at the beginning of the string, and blank-pad the rest of the string out to the length set before the call. FMP will not blank-pad anything past 64 characters, however.

A good example is FmpRunProgram, since it has both input and output string parameters.

```

PROGRAM whpa; {Runs WH program with run string RU,WH,PA}

TYPE
  INT      = -32768..32767;
  RUN_STRING = STRING [256];
  PROG_NAME = STRING [5];
  PARAMETERS = ARRAY [1..5] OF INT;

VAR
  wh_str: RUN_STRING;
  wh_parms:PARAMETERS;
  wh_name: PROG_NAME;
  wh_err: INT;

FUNCTION FmpRunProgram
$FIXED_STRING ON, HEAPPARMS OFF$

  (run_str:          RUN_STRING;{String passed to program, including name}
   VAR return_parms: PARAMETERS;{Parameters passed back from program}
   VAR sched_name:   PROG_NAME) {Name of program constructed by FMP}
   : INT;                      {Error value}

$FIXED_STRING OFF, HEAPPARMS ON$
EXTERNAL;

BEGIN
  {Must first set length of returned parameter}
  wh_str := 'ru,wh,pa';
  wh_name := StrRpt (' ', 5);
  wh_err := FmpRunProgram (wh_str, wh_parms, wh_name);
END.

```

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.

Strings:

The internal representation of strings is different between Pascal and FORTRAN. Pascal provides an option for automatic conversion to FORTRAN string format, but no facility exists for converting FORTRAN strings to Pascal format.

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. This example shows accessing of FORTRAN common by a Pascal subprogram. Pascal programs can access system common through user callable Pascal library routines (see Appendix F).

```

FTN,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: Common initialized to:', 3F10.2)
20 FORMAT('STEP 2: Changed in Pascal to: ', 3R10.2)
END

$SUBPROGRAMS$
PROGRAM pasc;

TYPE
  COMMON = RECORD
    first,
    second,
    third : REAL;
  END;

PROCEDURE pas
  (VAR com: COMMON);

BEGIN
  WITH com DO BEGIN
    first := 10.0;
    second := 20.0;
    third := 30.0;
  END;
END;
.

```

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.

```
$IMAGE 5000$
PROGRAM foo;

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 ilevel: 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: BOOLEANA);
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 Programmer's Reference Manuals for RTE-6/VM and RTE-A. 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
  exec11 = 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 (exec11, 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*.

Error returns can be handled for EXEC (and other routines) by doing one of the following:

1. Declare the EXTERNAL for EXEC with the ERROREXIT compiler option, and set the no-abort bit on the service request code parameter. This will cause Pas.ErrorExit to be called when the EXEC call fails. The error is then routed through the normal Pascal runtime error processor. If the program has supplied its own Pas.ErrorCatcher, then the error can be handled there. If not, a message will be printed, and the program will abort.
2. Declare the EXTERNAL for EXEC with the NOABORT compiler option, set the no-abort bit on the service request code parameter, and place an executable statement for handling the error return immediately following the EXEC call. This statement may be any Pascal statement, including a BEGIN...END statement.

On a normal return, this statement is skipped, and the statement following it is executed.

On an error return, this statement is executed, and control then falls into the statement following it.

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;
  exec7        = 7;
  exec7_na     = 7 + no_abort_bit;

TYPE
  INT = -32768..32767;
  ASCII_WORD = PACKED ARRAY [1..2] OF CHAR;

VAR
  a_reg: ASCII_WORD;
  b_reg: ASCII_WORD;

PROCEDURE abreg
  (VAR a_reg: ASCII_WORD;
   VAR b_reg: ASCII_WORD);
  EXTERNAL;

PROCEDURE exec_error;

BEGIN
  writeln ('**ERROR IN EXEC CALL - ERROR CODE IS:',a_reg,b_reg);
  halt (1);
END;

PROCEDURE suspend1 {No error exit; EXEC will handle error}
  $ALIAS 'EXEC'$
  (icode: INT);
  EXTERNAL;

PROCEDURE suspend2 {Error exit;      Pascal will handle error}
  $ALIAS 'EXEC', ERROREXIT$
  (icode: INT);
  EXTERNAL;

PROCEDURE suspend3 {No-abort;       User will handle error}
  $ALIAS 'EXEC', NOABORT$;
  (icode: INT);
  EXTERNAL;

BEGIN
  :
  suspend1 (exec7);           {error handled by EXEC}
  :
  suspend2 (exec7_na);       {error handled by Pas.ErrorCatcher}
  :
  suspend3 (exec7_na);       {error handled by exec_error}
  BEGIN
    abreg(a_reg, breg);      {get registers here because a call}
    exec_error;               {to a Pascal routine can destroy them}
  END;
  :
END.

```

ABREG Calls

To ensure that the values of the registers are returned correctly these guidelines should be followed:

All compilation units:

The use of a simple variable of an integer subrange type that occupies one-word is recommended. The use of fields of unpacked statically declared records is safe. The use of any object which involves array element selection, pointer dereferencing, or a function call, is unsafe. Calling a Pascal procedure which in turn calls ABREG, is unsafe.

CDS compilation units:

In addition to the above considerations, the objects passed should be local to the procedure or function which is calling ABREG or they should be global to the entire program. The use of any non-local and non-global object is unsafe.

Appendix A

Error Messages and Warnings

This appendix covers the errors which can be detected during the compilation or execution of a Pascal/1000 program. These are Program errors, I/O errors, FMP errors, EMA errors, Segmentation errors, and compile-time errors.

Program Errors

These errors occur when the program detects a value that it cannot process or when a problem is detected in the management of the heap area. The format of the error message is:

*** Pascal Error: "Error Message"

Message Number	Error Message
<0	Pas.ErrorExit Called At xxxx With AA-BB
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
5	MOD By Invalid Value In Line xxxx
6	String Underflow In Line xxxx
7	String Overflow In Line xxxx
8	String Bad Index In Line xxxx
9	Invalid String 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
96	Overflow Of Two Word Integer In Line xxxx
97	Illegal Char For Base In Line xxxx
98	No Value To Convert In Line xxxx
99	Insufficient Image Space In Line xxxx

The following pages describe the errors and give corrective actions for each error.

Error Messages and Warnings

Table A-1. Pascal Run-time Errors and Warnings

Message Number	Message Text and Description	User Response
<0	<p>Pass.ErrorExit Called at xxxx With AA-BB</p> <p>The Pascal error routine Pas.ErrorExit has been called. Calls to Pas.ErrorExit are automatically emitted at the error return points of system arithmetic intrinsics used by Pascal, and are emitted for user calls to externals declared with the ERROREXIT option. See the appropriate system documentation for a description of the error codes returned in the A and B registers.</p>	<p>Correct the argument to the affected arithmetic routine or user called external routine.</p>
001	<p>Undefined Case In Line xxxx</p> <p>The line number is the line of the beginning of the CASE statement. A case selector had a value which did not correspond with any case label, and no OTHERWISE clause was specified.</p>	<p>Add a case to handle the value which caused the error, or add an OTHERWISE clause to handle the value, or change the program logic so the value of the selector corresponds with one of the case labels.</p>
002	<p>Heap/Stack Collision In Line xxxx</p> <p>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.</p>	<p>Check for an infinite loop in the program. It may be that part of the loop is allocating memory by calling new, and it eventually runs out of memory. Check to make sure the program has been loaded with enough heap area available. If it hasn't, reload the program allocating more heap space.</p>
003	<p>Nil Pointer Dereferenced In Line xxxx</p> <p>An attempt was made to use a dynamic variable that did not exist (i.e. the pointer to it was nil).</p>	<p>Check to see that the pointer has been initialized properly; check the program logic to see if the pointer is being dereferenced before it has been initialized; check to see that the pointer has not been corrupted.</p>
004	<p>Value Out Of Range In Line xxxx</p> <p>A value to be assigned, used as an array subscript, used in a set denotation, or passed as a value parameter, is out of the range of valid values. The line number is the beginning of the assignment statement, beginning of the pointer or subscript expression, or the beginning of the routine to which the argument is being passed.</p>	<p>Correct the program logic error which has caused the invalid value to be used; check the value which has caused the error to see if it is a legitimate value. Also, check the type definition.</p>
005	<p>MOD By Invalid Value In Line xxxx</p> <p>The value on the right hand side of a MOD was not positive.</p>	<p>Correct the program logic error which has caused the invalid value to be used. Note that MOD is not the remainder operator.</p>

Table A-1. Pascal Run-time Errors and Warnings (Continued)

Message Number	Message Text and Description	User Response
006	<p>String Underflow In Line xxxx A string procedure or function attempted to create a string with negative length or to remove characters beyond the end of the string.</p>	Correct the argument to the string procedure or function or the program logic. Only the StrDelete routine's error on underflow can be disabled with Pas.StrEndCheck (see Appendix D).
007	<p>String Overflow In Line xxxx A string operation or procedure or function attempted to create a string with more characters than can be assigned, passed as a parameter, or compared.</p>	Correct the string operation, procedure or function call arguments, or the program logic. Errors on overflow can be disabled with Pas.StrEndCheck (see Appendix D) for concatenation, (+) Strappend, Strinsert, Strwrite and Stmore.
008	<p>String Bad Index In Line xxxx The string element index passed to a string procedure or function or used to index to a character in the string is outside of the currently valid string contents or is negative.</p>	Correct the argument or the program logic.
009	<p>Invalid String In Line xxxx A string was used that has not been initialized or has been corrupted. Not all uninitialized or corrupted strings can be detected.</p>	Correct the program logic to initialize the string before it is used. This error cannot be disabled.
091	<p>Dispose Called with a Nil Ptr Dispose was called with a pointer that did not point to any dynamic variable (i.e. the pointer was nil).</p>	Check to see that the pointer has been properly initialized and has not been corrupted before calling dispose.
092	<p>Disposed Of An 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.</p>	Correct the program logic error which has caused the size mismatch in the alternative form of dispose.
093	<p>Release Called With a Nil Ptr Release was called with a pointer that was nil.</p>	Check to see that the pointer was correctly set by a call to mark and has not been corrupted before calling release.
094	<p>Dispose Called With a Bad Ptr 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).</p>	Correct the program logic error which caused the uninitialized or corrupted pointer to be passed to dispose.

Error Messages and Warnings

Table A-1. Pascal Run-time Errors and Warnings (Continued)

Message Number	Message Text and Description	User Response
095	Release Called With A Bad Ptr 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).	Correct the program logic error which caused the uninitialized or corrupted pointer to be passed to release.
096	Overflow Of Two Word Integer The result of a numeric conversion function exceeds the representable range of a two word integer.	Correct the argument to the numeric conversion function to be a representable value.
097	Illegal Char For Base The parameter of a numeric conversion function contains a character which is not valid in the particular conversion base.	Correct the argument to the numeric conversion function to contain only valid characters in the particular base.
098	No Value To Convert The argument of a numeric conversion function contains no characters on which to perform the conversion (leading and trailing blanks are ignored).	Correct the argument to the numeric conversion function to contain the appropriate characters in the particular base.
099	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.	Check to see if the amount of IMAGE area can be reduced. If not, the program can be made to run in a larger partition with the SZ command.

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. I/O errors pertain to file reads/writes as well as string reads/writes.

The error message is of the form.

```
*** Pascal I/O Error On File xxxx
  ``Error Message''
or

*** Pascal String I/O Error In Line xx
  ``Error Message''
```

Where "xxxx" is the name of the Pascal/1000 logical file.

Message 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	Neg FLD/DEC 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 Name
19	File Is Not Open
20	Identifier Not In Enumerated Type
21	Value Not In Enumerated Type

The following pages describe the I/O errors and give corrective actions for each error.

Error Messages and Warnings

Table A-2. Pascal I/O Errors

Message Number	Message Text and Description	User Response
01	<p>Unexpected EOF The program attempted to read data or check the EOLN status of a file for which EOF would have returned TRUE prior to the attempt. The program attempted to write data to a record beyond the maximum accessible record (the value of MAXPOS) in a direct access file.</p>	<p>Correct the program logic to check EOF before reading file data or checking EOLN status. Correct the program logic to not write data to a direct access file at a record greater than that returned by MAXPOS.</p>
02	<p>File Must Be Text The program attempted to set carriage control status on a RESET, REWRITE, or APPEND call for a non-TEXT file.</p>	<p>Correct the program logic not to set carriage control status on a non-TEXT file.</p>
03	<p>File Must Be Direct SEEK, REaddir, WRITEDIR, or MAXPOS was used on a non-TEXT file that was opened for sequential access.</p>	<p>A non-TEXT file must be opened for direct access (with OPEN) to use SEEK, REaddir, WRITEDIR, POSITION, or MAXPOS.</p>
04	<p>Bad Record Length The file name specified on a file open call that actually creates the file (i.e. REWRITE, APPEND, or OPEN when the file does not exist) specifies a record length that is not compatible with the size of the file component, or an existing file being opened for direct access (with OPEN) has a record length that is not compatible with the size of the file component.</p>	<p>Correct (or remove) the record length specification from the file name, or make sure the correct file is being opened for direct access.</p>
05	<p>Must Reset Or Open File The program attempted to read data from a file that is not open or is open in a "write only" state, or the program used EOLN on such a file.</p>	<p>Correct the program logic to not read from the file or open the file in a way that permits reading (i.e. RESET or OPEN).</p>
06	<p>Must Rewrite Or Open File The program attempted to write data to a file that is not open or is open in a "read only" state, or attempted to open a new scratch file for "read only."</p>	<p>Correct the program logic to not write to the file or open the file in a way that permits writing (i.e. REWRITE, APPEND, or OPEN). Scratch files can only be created by opening them in a way that permits writing.</p>
07	<p>Direct Access Read Error The program read a record from a direct access file that had a size which is not compatible with the component type of the file.</p>	<p>Correct the program logic to use a file containing compatible data.</p>

Table A-2. Pascal I/O Errors (Continued)

Message Number	Message Text and Description	User Response
08	<p>Sequential Access Read Error The program read a record from a sequential access (non-TEXT) file that had a size which is not compatible with the component type of the file. This occurs most often when a variable size record file is accessed and all the records are not the same size or have a size different than that of the file component.</p>	Correct the program logic to use a file containing compatible data.
09	<p>Invalid Integer Read The program attempted to read an integer value from a TEXT file or string and no legal "integer" character sequence was detected after leading blanks and end-of-line characters were skipped.</p>	Correct the program logic to read the integer from the correct place in the file or string, or verify that the correct file or string is being accessed, or correct the corrupted file or string.
10	<p>Line Read Was Too Long A line in a physical file is longer than the line length limit of the logical TEXT file being used to access it.</p>	Make the lines in the physical file short enough to be within the line length of the TEXT file or increase the line length limit with the LINESIZE option.
11	<p>Invalid Real Number Read The program attempted to read a real/longreal value from a TEXT file or string and no legal "real" character sequence was detected after leading blanks and end-of-line characters were skipped.</p>	Correct the program logic to read the real/longreal from the correct place in the file or string or verify that the correct file or string is being accessed, or correct the corrupted file or string.
12	<p>File Is Not CCTL The routine OVERPRINT or PROMPT was used on a TEXT file which was opened with the NOCCTL option set. OVERPRINT and PROMPT can only be used on TEXT files which have carriage control on.</p>	Correct the program logic to not OVERPRINT or PROMPT to TEXT files which have carriage control off, or open the affected file with carriage control on.
13	<p>No Scratch File Available All scratch file names are in use.</p>	Remove unneeded scratch files or run the program in an environment in which all the scratch file names are not in use.
14	<p>Neg FLD/DEC Width Not Allowed The field width or decimal position specification in a write to a TEXT file or string is negative.</p>	Correct the program logic to not use negative values for the field width or decimal position.

Table A-2. Pascal I/O Errors (Continued)

Message Number	Message Text and Description	User Response
15	<p>File Cannot Be Type 1 Or 2 The routine APPEND was used on a physical file of type 1 or 2. Such files cannot be APPENDED to.</p>	Correct the program logic to specify a file which can be APPENDED to.
16	<p>File Must Be Type 1 Or 2 The routine OPEN was used on a physical file of other than type 1 or 2. Such files cannot be OPENed for direct access.</p>	Correct the program logic to specify a file which can be OPENed for direct access.
17	<p>Cannot Open LU 0 For Read Only The program attempted to open LU 0 with the routine RESET. LU 0 can only be opened for writing or reading/writing.</p>	Correct the program logic to open the correct file or open LU 0 in a way that permits writing (i.e. REWRITE APPEND, or OPEN).
18	<p>Missing File Name The file name passed to RESET, REWRITE, APPEND, or OPEN did not contain a valid file name, or RESET, REWRITE, APPEND, or OPEN was applied to a program parameter file and the corresponding argument was not supplied when the program was scheduled (note that setting the RUN-STRING option to zero will cause this error on any attempt to open a program parameter file.)</p>	Correct the program logic to provide a valid file name, or supply the appropriate argument when the program is scheduled, or set the RUN-STRING option to a value large enough to allow the command line arguments to be accessed by the I/O package.
19	<p>File Is Not Open The EOF function was applied to a file which is not open.</p>	Correct the program logic to not apply the EOF function to unopened files, or open the file.
20	<p>Identifier Not In Enumerated Type The program attempted to read an enumerated value from a TEXT file or string and did not find a sequence of characters which formed an identifier in the enumerated type after leading blanks and end-of-line characters were skipped. The program logic is in error or the contents of the file are corrupt.</p>	Correct the program logic to read the enumerated value from the correct place in the file or string, or verify that the correct file or string is being accessed, or correct the corrupted file or string.
21	<p>Value Not In Enumerated Type The program attempted to write an enumerated value and the internal representation of the value did not correspond with any identifier in the enumerated type. The expression being written involves uninitialized or corrupted data, or is an incorrect expression.</p>	Correct the program logic to yield a valid value for the enumeration expression.

The I/O warning message is of the form:

```
*** Pascal I/O Warning On File xxxxx
  ``Warning Message''
```

Where "xxxxx" is the name of the Pascal/1000 logical file.

Message Number	Warning Message
01	Output Line Moved To Next Line
02	Output Line Split

The following gives a description and corrective action for the warning messages.

Table A-3. Pascal I/O Warning Messages

Message Number	Message Text and Description	User Response
01	Output Line Moved To Next Line A field to be written to a TEXT file is longer than the space remaining on the current line. The field is moved to the next line to avoid it being split across lines.	If the warning occurred because a WRITELN was omitted, correct the program logic to do the WRITELN at the appropriate point. If the warning occurred because the file line length is not sufficient for the output to be generated, change the line length in the file with the LINESIZE option.
02	Output Line Split A field to be written to a TEXT file is longer than the space remaining on the current line and is also longer than the line length of the file. The field will be moved to a new line, split at the file line length, and continued on successive lines as necessary.	If the warning occurred because the field length was too long, correct the program logic to use the correct field length. If the warning occurred because the file line length is not sufficient for the output being generated, change the line length in the file with the LINESIZE option.

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 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 Pas.SegmentLoad. The error message has the form:

```
* * * Pascal segment xxxxx not found
```

Where "xxxxx" is the name of the segment passed to Pas.SegmentLoad.

Error Message Printers

The standard error message printer Pas.ErrorPrinter 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 runtime 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 runtime errors itself.

Each of the six types of runtime errors or warnings normally causes an error catching routine, Pas.ErrorCatcher (from the Pascal library), to be invoked. The version of Pas.ErrorCatcher in the library simply calls the error printer routine Pas.ErrorPrinter and then terminates the program. A program can catch and handle runtime errors by providing its own version of Pas.ErrorCatcher. An appropriate heading for Pas.ErrorCatcher 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 'Pas.ErrorCatcher'
  (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 Pas.ErrorCatcher 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: Zero for file I/O; line number of error for string I/O.
 err—file: The Pascal logical file name (undefined for string I/O error).
 err—flen: The length of the file name (undefined for string I/O error).

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 warning.
 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 `Pas.ErrorCatcher` can cause the default error printer to be invoked (which will write a message describing the error to LU 1) by calling the procedure `Pas.ErrorPrinter` with the same parameters (in the same order) that were passed to `Pas.ErrorCatcher`.

Note that `Pas.ErrorPrinter` does not terminate the program; this must be done by the user-supplied `Pas.ErrorCatcher` routine if that is the desired action. Every attempt is made to ensure that if the user `Pas.ErrorCatcher` exists, 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.

Any user-supplied `Pas.ErrorCatcher` must be compiled with `$RANGE OFF$` and any CASE statement in it must have an OTHERWISE associated with it. Further, an error catching routine must not do Pascal file or string I/O, pointer dereferencing, or procedure calls to new, dispose, mark or release. Attempts to perform these actions may cause the recursive invocation of error checking or error handling routines that were not designed to be so invoked.

A possible use of this mechanism would be to place Pas.ErrorPrinter in a segment so that it would not be present during normal execution. However, replacing Pas.ErrorCatcher with a version that does a segment load and then calls Pas.ErrorPrinter will not work, as the external reference to Pas.ErrorPrinter will cause it to be loaded from the library with Pas.ErrorCatcher (which is what you were trying to avoid). To make it work, replace Pas.ErrorCatcher with a version that does the segment load, and then calls a routine (of yours) in the segment, which in turn calls Pas.ErrorPrinter. Obviously the parameters must be passed to your routine so that it can pass them to Pas.ErrorPrinter.

NOTE

The user-supplied version of Pas.ErrorCatcher as well as any routines which are given access to any of the parameters of Pas.ErrorCatcher must be compiled with either the \$HEAP 1\$ or \$HEAPPARMS OFF\$ option.

The loading of an error segment should not be done if the error being caught is a segment loading error. The user-supplied Pas.ErrorCatcher should process such an error itself. For any other kind of error, this avoids an unpleasant situation that would occur if the process of loading the error segment caused an error. The error catcher would be reentered, would try to load the error segment again, would probably get an error again, and infinite looping would result.

Compile-Time Warnings

The following conditions and examples can cause compile-time warnings:

- An unrecognized compiler option, e.g. \$NOT_AN_OPTION\$.
- An unrecognized compiler option setting, e.g. \$STANDARD_LEVEL'hp16c'\$.
- A construct that is not allowed at the current standard level, e.g. \$STANDARD_LEVEL 'ansi'\$ CONST pi=3.0+0.14.
- use of a compiler option which is ignored or interpreted in other than the usual way because of the context in which it appears.

Compile-Time Errors

The following pages list the compile-time errors, their descriptions and a suggested user response.

Table A-4. Pascal Compile-Time Errors

Message Number	Message Text and Description	User Response
001	Error in simple type Array index types and set element types must be simple types.	Change the array index type or set element type to be scalar or subrange type.
002	Identifier expected Compiler looking for a predefined or user-defined identifier that is either missing or misplaced.	Check for missing identifier or incorrect syntax.
003	'PROGRAM' expected The word 'PROGRAM' was expected to begin the main program.	Insert 'PROGRAM' in program heading.
004	') expected The compiler has found a left '(' but no closing ')'.	Insert the ')' in the appropriate place.
005	:: expected The compiler expected a colon.	Check syntax. Insert colon where necessary.
006	Illegal symbol The compiler does not recognize the letter, digit or special symbol.	Check symbols validity in Chapter 2.
007	Error in parameter list The actual and formal parameter list must be compatible.	Check actual and formal parameter lists. Refer to Chapter 4.
008	'OF' expected A 'type declaration' or CASE statement is missing the word 'OF'.	Check syntax and insert 'of.'
009	(' expected The compiler expected a '(' to start a parameter list.	Insert '(' where necessary.
010	Error in type A type definition is expected and not seen by the compiler, or the word PACKED appears before an inappropriate type definition.	Make sure type definition is a legal one.
011	[' expected The compiler expected a '[' in an array definition, a structured constant definition, or a set constructor.	Insert '[' where necessary.
012	'] expected The compiler found a '[' but did not find a ']' to close the construct.	Insert ']' where necessary.

Table A-4. Pascal Compile-Time Errors (Continued)

Message Number	Message Text and Description	User Response
013	<p>'end' expected The word 'end' was expected to complete a block, CASE statement or RECORD definition. The compiler looks for an 'end' to correspond with each 'begin'.</p>	Check the block for the correct number of 'begin' and 'end'.
014	<p>";" expected The compiler expected a semicolon.</p>	Check syntax and insert semicolon if necessary.
015	<p>Integer expected The compiler expected an integer, for example in a LABEL declaration or definition.</p>	Check syntax and insert integer where necessary.
016	<p>'=' expected Compiler expected an equal sign.</p>	Check syntax for missing equal sign.
017	<p>'BEGIN' expected The word 'BEGIN' was expected to start the block. Pascal requires a 'begin' to correspond with each 'end'.</p>	Insert 'begin' where necessary. Check nesting.
018	<p>Error in declaration part The compiler has reached what it considers the end of the declarations but did not find the beginning of a block (PROCEDURE, FUNCTION), body (BEGIN) or the end of a \$SUBPROGRAM\$ unit (.)</p>	Check syntax of declarations and/or PROCEDURE or FUNCTION heading.
019	<p>Error in field list The compiler has detected an error in the field list of your record declaration.</p>	Check syntax of record declaration.
020	<p><b',' b="" expected<=""> The compiler expected a comma.</b','></p>	Check syntax and insert comma where necessary.
021	<p>'. expected The compiler expected a period (i.e. at the end of the compilation unit).</p>	Insert period where needed. Also check block nesting.
023	<p>String literal expected The argument to a compiler option that takes a string literal was incorrectly specified or a string structured constant contains something other than string or character literals.</p>	Check the option syntax. Correct the option argument. Check string structured constant syntax. Correct the string structured constant.
024	<p>'..' expected The double periods (..) were not found by the compiler in the subrange declaration.</p>	Check subrange syntax and insert '..' where necessary.

Table A-4. Pascal Compile-Time Errors (Continued)

Message Number	Message Text and Description	User Response
025	<p>Illegal character in this context A non-printing character was found in the source code or a printing character that is only legal within a string literal was found outside of a string literal.</p>	Check character validity in context.
026	<p>', or ';' expected A comma or semicolon is expected between multiple compiler options in an option list.</p>	Check syntax of compiler options.
027	<p>'EXPORT' expected The word 'EXPORT' was expected in a module declaration but it was omitted or incorrectly specified.</p>	Insert missing word, check syntax.
028	<p>'IMPLEMENT' expected The word 'IMPLEMENT' was expected in a module declaration but it was omitted or incorrectly specified.</p>	Insert missing word, check syntax.
029	<p>'IMPORT' may not appear here An 'IMPORT' declaration appears somewhere other than in a module or in the program global declarations.</p>	Remove/correct the import declaration, check syntax.
030	<p>'MODULE' expected In a module list compilation unit, a semicolon followed a module declaration, but the reserved word 'MODULE' was not found following the semicolon. In a module-list compilation unit, semicolons are used to separate module declarations. The last module declaration in a module-list unit should be followed by a period (not a semicolon).</p>	Remove the semicolon from the declaration if it is the last one in the module-list unit, or correct the source so that a valid module declaration follows the semicolon.
047	<p>Expression must have integer value The expression preceding 'OF' in a structured constant declaration must be an integer type constant expression.</p>	Correct the constant expression that precedes 'OF'.
048	<p>Expression must be non-negative The expression preceding 'OF' in a structured constant declaration must have a non-negative value (zero is permitted).</p>	Correct the constant expression that precedes 'OF'.

Table A-4. Pascal Compile-Time Errors (Continued)

Message Number	Message Text and Description	User Response
049	<p>Expression must be a constant A variable was found where a constant is required; an expression with variables where a constant expression is required; an operator, standard procedure or function is in the expression which is not legal in a constant expression; the expression contains constant operands which are not legal, e.g. real, set or boolean values.</p>	Check the constant expression for a variable, or illegal type of operand.
050	<p>Error in constant An invalid expression occurs where a constant expression is required.</p>	
051	<p>' := ' expected The compiler expected a ' := ' but it was omitted or incorrectly specified.</p>	Insert missing symbol/word; check syntax.
052	<p>'THEN' expected The compiler expected the word 'THEN' but it was omitted or incorrectly specified.</p>	Insert missing symbol/word; check syntax.
053	<p>'UNTIL' expected The compiler expected 'UNTIL' but it was omitted or incorrectly specified.</p>	Insert missing symbol/word; check syntax.
054	<p>'DO' expected The compiler expected 'DO' but it was omitted or incorrectly specified.</p>	Insert missing symbol/word; check syntax.
055	<p>'TO' or 'DOWNTO' expected The compiler expected 'TO' or 'DOWNTO' but they were either omitted or incorrectly specified.</p>	Insert missing symbol/word; check syntax.
060	<p>Error in expression A type identifier (other than a set type) or an invalid symbol appears in an expression.</p>	Check syntax of expression.
061	<p>Modules may not be in a subprogram or segment A module may not be declared in a program unit compiled with either the subprogram or segment compiler option (although an import declaration can appear).</p>	Remove the module declaration or make the compilation unit a module-list unit.
062	<p>Module has internal error A module being imported from a separately and previously compiled source contains internal errors. The relocatable file has either been corrupted or the compiler generated invalid information.</p>	Recompile the source containing the affected module then recompile the importing source. If the error is not eliminated report as a bug.

Table A-4. Pascal Compile-Time Errors (Continued)

Message Number	Message Text and Description	User Response
063	<p>Modules must be visible A module may not be declared when the visible compiler option is off.</p>	Set the visible option on around the module declaration.
064	<p>Modules must be declared at outermost level Modules may only be declared in the program global declarations or in a module unit. They may not be declared within procedures, functions, or other modules.</p>	Move the module declaration to the program global level or into a module-list unit.
065	<p>Modules must have non-empty export section The export section of a module must contain the definition or declaration of at least one "object" to be exported.</p>	Declare or define at least one 'object' in the export section.
066	<p>Modules may not declare labels A label declaration may not appear in a module declaration (except within a procedure or function declared in the module.)</p>	Remove the label declaration.
067	<p>Module version inconsistency During importation the compiler detected that a module being imported or one of its ancestors (modules imported by it, and so on recursively) has the same name but a different compilation time as a module or one of its ancestors that was previously encountered during importation.</p>	Recompile all the compilation units which import the affected module and any modules that import those modules etc. as necessary.
068	<p>Module unit may only contain modules A module unit (a compilation unit consisting of one or more module declarations) can only contain module declarations.</p>	Remove the non-module declarations or put the modules and the declarations in the program globals.
069	<p>Module not found An import declaration specified a module that cannot be found in the current compilation unit, nor in a compilation unit specified by the search compiler option (if any), nor in the default module library.</p>	Correct the search option, or compile the module being imported and add it to the search option, or declare the module being imported earlier in the compilation unit.
070	<p>External routine must be declared at outermost level A routine declared inside another routine may not be declared EXTERNAL.</p>	Move the EXTERNAL routine declaration so it is not nested within any other routine.

Table A-4. Pascal Compile-Time Errors (Continued)

Message Number	Message Text and Description	User Response
071	<p>Aliased routine must be declared at outermost level</p> <p>A routine declared inside another routine may not have the ALIAS option specified.</p>	Remove the ALIAS or move the routine so that it is not nested within any other routine.
072	<p>Recursive routine may not be direct</p> <p>A procedure or function declaration has the direct compiler option specified while the recursive compiler option is on.</p>	Remove the direct option or set the recursive option to off.
073	<p>Errorexit/noabort may not be used here</p> <p>An actual procedure or function was declared with the errorexit or noabort compiler option specified, or an external function was declared with the noabort option specified. These options may not be specified for actual procedures or functions, nor for external functions (noabort only, errorexit is permitted).</p>	Remove the errorexit or noabort option from an actual declaration. If the external function can be declared and used as a procedure instead, then noabort can be used.
074	<p>Library routines must be visible</p> <p>If the library compiler option is on, the visible compiler option must also be on. Note that specifying CDS ON (or using a compiler which defaults to CDS ON) will set LIBRARY ON.</p>	Specify visible on or library off. See library, visible, and CDS option rules (Appendix C).
075	<p>Forward/exported routine may not be external</p> <p>A procedure or function previously declared forward or exported may not be declared external.</p>	Declare the actual routine completely, it cannot be declared as external.
077	<p>Search file open error</p> <p>A file specified in the search compiler option could not be opened in any of the expected places (see SEARCH option) or was opened but was not a type 5 file.</p>	Correct the search option or make sure that the file exists, is type 5 and is where it can be found.
078	<p>Search file read error</p> <p>A file system error occurred trying to read a file specified in the search compiler option.</p>	Correct the search option or file system problem. The file may be corrupt, restore or recompile as necessary.
079	<p>Search file record length error</p> <p>The internal record length field in a relocatable record is not consistent with the record length returned by the file system.</p>	Correct the search option or file system problem. The file may not be a relocatable file or may be corrupt.

Table A-4. Pascal Compile-Time Errors (Continued)

Message Number	Message Text and Description	User Response
080	<p>Negative field width not allowed A field width or decimal width specification in a call to write, writeln, prompt, or overprint is negative.</p>	Make the field width positive, or 0.
081	<p>Type too big An array contains more than MAXINT elements.</p>	Decrease the number of elements in the array type.
082	<p>Expression must be in the range 1..32767 The expression which specifies the maximum length of a string type must be in the range 1..32767.</p>	Correct the string maximum length specification to be in the permitted range.
083	<p>Type not imported An object was imported whose type was not imported, or a pointer type was imported but the type it pointed to was not.</p>	Either import the module which defines the type(s) in question or give the imported type a new name which is exported along with objects typed with the new name; define the pointed-to type in the export section of the module which defines pointer type (if it is acceptable to make the details of the internal representation known to the importer) or remove the pointer de-reference operation (see Chapter 4, Modules).
096	<p>Overflow of two word integer The compile-time evaluation of one of the numeric conversion functions (hex, octal, or binary) yielded a result that cannot be represented in a two-word integer.</p>	Correct the argument to the numeric conversion function to be in the representable range.
097	<p>Illegal character for given base The compile-time evaluation of one of the numeric conversion functions (hex, octal, or binary) found a character which is not legal in the appropriate base.</p>	Correct the argument to the numeric conversion function.
098	<p>No value to convert The compile-time evaluation of one of the numeric conversion functions (hex, octal, or binary) did not find any characters (other than blanks) on which to perform the conversion.</p>	Correct the argument to the numeric conversion function.
100	<p>Duplicate or invalid external name The external name duplicates a previously encountered user defined external name or a compiler generated external name.</p>	Check names; make external name unique. Don't use names that begin with 'Pas.' or end with '.dddd' (where dddd are any number of digits).

Table A-4. Pascal Compile-Time Errors (Continued)

Message Number	Message Text and Description	User Response
101	Identifier redeclared The identifier already exists in the current scope.	Delete duplicate declaration.
102	Low bound exceeds high bound The lower bound is larger than the high bound; (i.e. TYPE int = 32768..-32767)	Increase the upper bound, and/or decrease lower bound.
103	Identifier is not of appropriate class The identifier is not of a class appropriate in the current context. For example, a variable name appears before a '(' or a expression.	Change either the declaration or the usage of the identifier to make sure they are consistent.
104	Identifier not declared The identifier is an undeclared variable constant, procedure or function; or the type identifier is undeclared.	Add identifier to declaration section.
105	Sign not allowed A sign (+ or -) appears where one is not permitted, for example in a LABEL declaration.	Remove the sign.
106	Identifier redefined after use in this scope An identifier associated with a type has been redefined after being used in this scope.	Correct redefinition or initial use, check scope of identifier.
107	Incompatible subrange types The upper and lower bounds of a subrange declaration must be of compatible type.	Check type compatibility.
108	File not allowed here A file type may not appear as the component type of a file or in the variant part of the record.	Check file usage in the source program.
109	Tag type must not be real The tag type cannot be REAL.	Change tag to scalar or subrange type.
110	Tag type must be scalar or subrange. The tag is not valid for the variant part.	Change tag to scalar or subrange type.
111	Incompatible with tagfield type A label in the variant CASE declaration is of a type not compatible with that of the tag type.	Change either the tagfield type or the variant specifier so the two are consistent.

Table A-4. Pascal Compile-Time Errors (Continued)

Message Number	Message Text and Description	User Response
112	Index type must not be real The compiler will not accept a real as an index type.	Change index to scalar or subrange.
113	Index type must be scalar or subrange The index type is not valid.	Change index to scalar or subrange.
114	Base type must not be real or longreal The base type of a set may not be REAL.	Change base type to scalar or subrange.
115	Base type must be scalar or subrange The base type of a set must be scalar or subrange.	Change base type to scalar or subrange.
116	Error in type of standard procedure parameter The parameter that is passed to the standard procedures is of an incorrect type.	Check type declarations for consistency.
117	Unsatisfied forward reference A pointer type was declared but the type being pointed to was not declared.	Declare the type being pointed to or remove the pointer declaration.
118	Undeclared forward/exported procedure or function A routine was declared FORWARD but the actual routine was never declared. A routine heading appeared in the EXPORT part but the routine was not declared in the implement part of a module.	Remove the FORWARD or EXPORT declaration or insert the routine declaration.
119	Forward/exported; repeated parameter list mismatch. The parameter list of a FORWARD or exported routine was incorrectly specified at the actual declaration of the routine.	Correct or delete parameter list in actual procedure/function heading (it must match exactly or be completely omitted.)
120	Function may not return this type A function may not return a file or a type containing a file.	Check the function result type.
121	File value parameter not allowed A file must be passed as a VAR VAR parameter.	Make sure that the file parameter is a VAR parameter.
122	Forward/exported; repeated result type mismatch. The result type of a FORWARD or exported function was incorrectly specified at the actual declaration of the function.	Correct or delete the result type in actual function heading (it must match exactly or be completely omitted.)

Table A-4. Pascal Compile-Time Errors (Continued)

Message Number	Message Text and Description	User Response
123	<p>Missing result type in function declaration The function header is missing the result type declaration.</p>	Insert the result type declaration.
124	<p>Decimal position for real only The decimal position specifier in a call to write, writeln, prompt, or overprint can only be used with real/longreal parameters.</p>	Check type of parameter or remove decimal position specifier.
125	<p>Error in type of standard function parameter The parameter in the standard function is not correct type.</p>	Check type of parameter. Refer to Chapter 7.
126	<p>Number of parameters does not agree with declaration The number of parameters declared does not match the number of parameters being passed to the routine.</p>	Check consistency between the procedure call and procedure declaration.
127	<p>Missing parameter to standard routine The standard routine required a certain parameter that has not been passed.</p>	Check procedure syntax in Chapter 7.
128	<p>Result type of parameter function conflicts with declaration A function being passed as a parameter does not have the same function result type as the function parameter was declared to have.</p>	Check the result type of the function being passed and the type of the function parameter for consistency.
129	<p>Type conflict of operands An expression cannot contain conflicting operator types (i.e., boolean + integer)</p>	Check operands to ensure compatible types.
130	<p>Expression is not of set type A set operator was used and the compiler expected the expression to be of set type.</p>	Change expression to set type.
131	<p>Only tests of equality are allowed Pointers may only be compared for equality or inequality.</p>	Remove pointer comparison or change test.
132	<p>Strict inclusion not allowed Sets can be tested for quality, inequality, sets and subsets.</p>	Change relational operators. Refer to Chapter 5.
133	<p>File comparison not allowed Files cannot be compared to one another.</p>	Remove file comparison.

Table A-4. Pascal Compile-Time Errors (Continued)

Message Number	Message Text and Description	User Response
134	Illegal type of operands(s) An expression cannot contain conflicting operand types.	Check types of operands for incompatibility.
135	Type of operand must be Boolean The operand must be boolean when the operator is boolean (i.e. OR, AND, NOT)	Change the operand to a boolean type.
136	Set element type must be scalar or subrange The type of elements in a set must be a scalar or subrange type.	Change set element type to scalar or subrange.
137	Set element types not compatible The types of all operands in a set denotation must be compatible.	Change types to be compatible. See Chapter 4.
138	Subscript may not be used here A subscript was used on something other than on array expression.	Correct the array expression or remove the subscript.
139	Index type is not compatible with declaration The type of an indexing expression must be compatible with the declared type of the index.	Check declaration and index type for compatibility.
140	Type of variable is not record The variable has not been declared as a record.	Declare variable as a record.
141	Type of variable must be file or pointer. The up-arrow must be used with a file or pointer expression.	Remove up-arrow or change preceding expression to be of type pointer or file.
142	Illegal parameter substitution Parameter declaration inconsistent with parameter passing.	Check the declaration against what is being passed.
143	Illegal type of loop control variable A loop control variable must be scalar or subrange and not real.	Change the type of the loop control variable.
144	Illegal type of expression The type of a case selector and case labels must be scalar or subrange and not real.	Check the type of the case selector expression and/or the case label.

Table A-4. Pascal Compile-Time Errors (Continued)

Message Number	Message Text and Description	User Response
145	<p>Loop bounds type conflict The type of a FOR loop initial or final value expression is not compatible with the loop control variable.</p>	Change the types of the initial and final value expressions or of the loop control variable as appropriate.
146	<p>Assignment of files not allowed Files cannot be assigned to one another.</p>	Remove the file assignment.
147	<p>Label type incompatible with selecting expression. The type of a case label is not compatible with the type of the case selector expression.</p>	Change the label or selecting expression as appropriate.
148	<p>Subrange bounds must be scalar The type of subrange bounds must be scalar and not real.</p>	Change subrange bounds to a scalar type.
149	<p>Not assignment compatible Reals are not assignable to integers. The expression on the right hand side of an assignment statement or being passed to a value parameter is not assignable to the left hand side or parameter type.</p>	Check the assignment compatibility rules.
150	<p>Assignment to standard function is not allowed A standard predefined function cannot be assigned a value.</p>	Remove the assignment to the predefined function identifier.
151	<p>Assignment to formal function is not allowed Functions passed as arguments cannot be assigned to.</p>	Remove the assignment to the formal function.
152	<p>No such field in this record The field specified after a field selection operator is not a field of the record being referenced.</p>	Check the field name and the record type definition.
153	<p>Type error in read A parameter to read, readln, readdir, is of a type which is incompatible with the file being read from.</p>	Check the parameter and file types for compatibility.
154	<p>Actual parameter must be a variable The actual parameter being passed where a VAR parameter is expected must be a variable.</p>	Check the parameter, it must be a variable and not an expression or constant.

Table A-4. Pascal Compile-Time Errors (Continued)

Message Number	Message Text and Description	User Response
155	<p>Loop control variable must be simple/local variable A FOR loop control variable must be declared in the routine containing the FOR loop and it must not be a component of an array or record.</p>	Use a local simple variable for the loop control variable.
156	<p>Multidefined case label The case label has been defined more than once.</p>	Remove or change the duplicate or original label as appropriate.
157	<p>Loop control variable may not be assigned to Assigning to a loop variable is not allowed.</p>	Remove assignment to loop control variable.
158	<p>Missing corresponding variant declaration The tag specified in the alternate form of new or dispose does not correspond to any variant in the record.</p>	Check the record type definition for the correct variant or add the correct variant to definition.
159	<p>Real or string tagfields not allowed The tag specified in the alternate form of new or dispose cannot be a string or real.</p>	Use valid tags in the new or dispose call.
160	<p>Previous declaration was not forward The previous declaration of the procedure or function was not a FORWARD declaration.</p>	If the routine is being duplicated, remove the duplicate, otherwise give it a name which does remove it.
161	<p>Again forward declared Routine can be declared forward only once.</p>	Remove all but one forward declaration per routine.
162	<p>Type error in write A parameter to write, writeln, or writedir, is of a type which is incompatible with the file being written to.</p>	Check the parameter and file types for compatibility.
163	<p>Missing variant in declaration Something other than a tag value was specified in the alternate form of new or dispose.</p>	Correct or remove the non-tag value.
164	<p>Substitution of standard procedure or function not allowed A standard routine may not be passed as a parameter.</p>	If the standard routine must be passed, declare your own routine which calls the standard routine and pass your routine. Otherwise remove/correct the parameter.

Table A-4. Pascal Compile-Time Errors (Continued)

Message Number	Message Text and Description	User Response
165	Multidefined label The label has been used more than once.	Remove/correct the duplicate definition.
166	Multideclared label The label has been declared more than once.	Delete the duplicate label declaration.
167	Undeclared label A label is used that has not been declared.	Declare label.
168	Undefined label A label has not been defined.	Check label definitions.
169	Error in base set A set contains more than 32,767 elements.	Define/declare set to have 32,767 or less elements.
170	Program parameters must be files The identifiers which appear as the program parameters must be declared as files within the program global declarations.	Correct the actual declaration to be a file declaration or remove the identifier from the program statement.
171	Actual parameter cannot be component of packed type or string A component of a packed type or string cannot be a VAR parameter.	Unpack the type in which the component appears; do not pass string components by VAR.
172	Substitution of routine with non-standard attributes not allowed. A routine passed as a parameter may not have been declared with any of the direct, errorexit, or noabort compiler options in effect.	Compile the routine being passed without any of the non-standard options or pass a different routine as appropriate.
173	Too many enumerated values More than 32,768 values were defined in an enumerated type.	Define/declare the enumerated type with 32,768 or less values.
174	File cannot be text file Certain files cannot be textfiles; i.e., those used with open, seek, readdir, etc.	Remove/change the file being referenced or the routine being used.
175	Default 'input' file is not accessible here The program heading is missing the 'input' file declaration or the identifier 'input' has been redefined.	Add 'input' to the program heading or remove the redefinition of 'input'.
176	Default 'output' file is not accessible here The program heading is missing the 'output' file declaration or the identifier 'output' has been redefined.	Add 'output' to the program heading or remove the redefinition of 'output'.

Table A-4. Pascal Compile-Time Errors (Continued)

Message Number	Message Text and Description	User Response
177	<p>Only variables may be assigned to Something other than a variable appears on the left side of an assignment statement.</p>	Correct the statement.
178	<p>Duplicate tag A tag value is specified more than once in a record variant definition.</p>	Remove/change the duplicate tag.
179	<p>Function Identifier not assignable here Cannot assign to a function identifier outside the function.</p>	Remove/change incorrect assignment.
180	<p>Type of expression must be Boolean The type of expression used must be boolean; i.e., IF b THEN... (b must be a boolean or boolean expression.)</p>	Correct the expression.
181	<p>No function result defined in the body of the function No value is assigned to the function.</p>	Add the assignment statement.
182	<p>Noabort procedure may not be invoked here A procedure declared with the noabort compiler option in effect may not be invoked where the "next" statement involves a flow of control change or is otherwise ambiguous.</p>	Move the procedure invocation to an appropriate place or add a clearly defined "next" statement to execute on the error return.
183	<p>Tag field cannot be passed by VAR The tag field of a record cannot be passed as a VAR parameter.</p>	Assign the tag to a local pass the local, then assign the local back to the tag, or change the parameter to not be the tag field of a record, or change the record definition so that the field is not a tag field or pass the entire record as a VAR parameter.
184	<p>Division by zero or MOD by non-positive value Division by constant zero or MOD by non-positive was detected in an expression.</p>	Correct the expression.
185	<p>Undeclared external file A file specified in the program heading (other than INPUT, OUTPUT) was not declared in the global area. INPUT and OUTPUT are special cases and must not be declared.</p>	Add the file declaration.

Table A-4. Pascal Compile-Time Errors (Continued)

Message Number	Message Text and Description	User Response
186	<p>File must be a text file Readln, writeln, eoln, page, prompt and overprint can only operate on textfiles.</p>	Remove/change the file being referenced or the routine being used.
187	<p>Option conflict Compiler option conflicts with the setting of a previous option.</p>	Check compiler option rules (Appendix C).
188	<p>Option cannot be specified here Option cannot be specified here. This option can only occur in certain specified places; for example, in the program heading or within a procedure or function heading.</p>	Check compiler option rules (Appendix C).
189	<p>Heap option must be set to use this routine The standard procedures new, dispose, mark, and release may only be used when the heap compiler option is set to a value other than zero.</p>	Set the heap option to the appropriate non-zero value before the program heading.
190	<p>Recursive option must be set to do recursion A recursive procedure or function call was detected in a routine declared with the recursive compiler option off. Only direct recursion causes this error; indirect recursion (A calls B then B calls A and A is not recursive) is not detected.</p>	Set the recursive option on for the routine.
191	<p>Option cannot be respecified This option cannot be specified more than once.</p>	Check options. Delete duplications.
193	<p>Include level too deep There are too many levels of include.</p>	See explanation for INCLUDE in Chapter 3, and Appendix C.
194	<p>Include file cannot be read The include file can't be read (does not exist or does not have read access.)</p>	Check file validity.
195	<p>Option has invalid parameter A compiler option was specified with a parameter of an unexpected type. Options take one of: no parameter, 'ON' or 'OFF', a numeric constant, or a string literal, as specified in Appendix C.</p>	Correct the option argument; check compiler option syntax (Appendix C).

Table A-4. Pascal Compile-Time Errors (Continued)

Message Number	Message Text and Description	User Response
196	<p>ON or OFF expected A compiler option which expects an 'ON' or 'OFF' (or omitted for 'ON') parameter, was specified with some other type of parameter.</p>	Correct the option argument, check compiler option syntax (Appendix C).
197	<p>Only options may appear in the option file Anything not between the \$ signs causes this error.</p>	Check option list.
198	<p>Heap variable may not be passed by reference in this context A variable in the heap in a heap 2 program may only be passed to a VAR parameter declared with the heapparms compiler option off if only one such parameter substitution is present in the procedure or function invocation.</p>	Remove the heapparms off specification or pass a variable which is not in the heap.
200	<p>Numeric constant too long A numeric constant is more than 150 characters long.</p>	Shorten the constant.
205	<p>Real constant exceeds range A real constant exceeds the range representable in this implementation.</p>	Check the permitted range of real: longreal value.
206	<p>Missing fractional part of real A real constant which contains a decimal point must have at least one digit of fractional part.</p>	Correct the constant to specify a fractional part.
207	<p>Missing scale factor of real or longreal An 'E' pr 'L' exponent specification appears in a real or longreal constant but no exponent appears.</p>	Correct the constant to specify an exponent.
209	<p>Overflow or underflow A constant or constant expression is outside the range representable in this implementation.</p>	Correct the constant or constant expression.
210	<p>Integer constant exceeds range An integer constant exceeds the range representable in this implementation.</p>	Check the permitted range of integer values.
215	<p>String constant too long Too many characters were declared in the string constant.</p>	Check string, max allowed is 150 characters.

Table A-4. Pascal Compile-Time Errors (Continued)

Message Number	Message Text and Description	User Response
216	<p>String constant exceeds source line A string constant must fit on a line.</p>	Check string constant for missing closing quote or make constant shorter.
218	<p>Non-printing character invalid in string A string cannot contain a non-printing character.</p>	Check string constant formation rules (Chapter 2).
219	<p>Invalid non printing character The non-printing character is invalid.</p>	Check non-printing character formation rules (Chapter 2).
220	<p>Character constant exceeds range A character specified in the # form range exceeds the range of characters supported by this implementation.</p>	Check the permitted range of character values.
225	<p>Label exceeds range A statement label must be in the range 0 to 9999.</p>	Check the label to ensure it is within the permitted range.
230	<p>Structured type identifier expected Structured constant requires that a type name precede the specification of structured components.</p>	Check the syntax of structured constants.
231	<p>Too few constants Structured constant declared with too few constants.</p>	Add constants or change declaration.
232	<p>Too many constants Structured constants declared with too many constants.</p>	Delete constants or change declarations.
233	<p>Field(s) not specified Structured constant records must specify all fields.</p>	Supply values for all fields.
234	<p>Field respecified Structured constant records can specify each field only once.</p>	Remove/change duplicate specification.
235	<p>Tag not set or set to another variant Structured constant record contains a specification for a field that is in a variant other than that specified, or the variant was never specified.</p>	Make sure the variant is specified before the fields of the variant are specified, or that the field is in the specified variant.
236	<p>Set type id expected Something other than a set type identifier appeared before a specification in an expression.</p>	Check for a missing or incorrect set type identifier.

Table A-4. Pascal Compile-Time Errors (Continued)

Message Number	Message Text and Description	User Response
237	Constant of wrong type Constant used in structured constant did not have the type expected in the constant being specified.	Check structured constant syntax.
240	Anonymous STRING parameter must be VAR A procedure or function parameter of the anonymous STRING type must be passed by VAR.	Make the declaration a VAR parameter or make the type a specific string type.
250	Too many nested scopes of identifiers Procedures, functions, and/or with statements are nested too deeply.	Reduce nesting level of routines plus records to less than 20.
251	Too many nested blocks of code Procedures and/or functions are nested too deeply.	Reduce nesting level of routines to less than 10.
253	Unexpected end of source file End of the source file was encountered before a valid program was processed.	Check for missing END, semicolon, period or incomplete statement.
254	Source line too long The source line was longer than 150 characters.	Shorten source file lines.
255	Too many errors on this source line More than 10 errors occurred. Further errors on this line will be counted but not displayed.	Correct earlier errors so later ones can be displayed (during next compilation).
256	Source appears after the end of the program unit Only comments or compiler options may appear after the end of the program unit.	Remove the extra source or correct the errors which cause the compiler to believe it was at the end of the source when it was not.
257	Too much local variable space The amount of space required by local variables in a procedure or function exceeds the limits of the machine. This is a total space overflow condition which is detectable at compile time. Stack overflow at execution time cannot be detected by the compiler.	Declare fewer or smaller local variables or put some of the variables in the heap.
260	Compiler label overflow: break into separate compilations The compiler has run out of internal labels.	Break compilation unit into separate units replace with or break up level-1 routines into multiple level-1 routines.

Table A-4. Pascal Compile-Time Errors (Continued)

Message Number	Message Text and Description	User Response
262	<p>Small temps exhausted: use more SMALL—TEMPS or simplify expression</p> <p>The compiler has run out of space for small temporaries.</p>	<p>Use the SMALL—TEMPS option to provide more small temporary space or declare fewer or smaller local variables, or break large expressions into a series of smaller expressions keeping intermediate results in your own temporaries, or break up the routine into smaller routines.</p>
302	<p>Index expression out of bounds</p> <p>An array index was detected to be out of bounds at compile time.</p>	<p>Check the index expression and array type definition.</p>
303	<p>Value to be assigned is out of bounds</p> <p>A value to be assigned was detected to be out of bounds at compile time.</p>	<p>Check the type of the destination of the assignment and the value of the expression.</p>
304	<p>Element expression out of range</p> <p>The expression is using an element not in the set range.</p>	<p>Check the set type and expression value.</p>
305	<p>Actual parameter out of bounds</p> <p>A value parameter was detected to be out of bounds at compile time.</p>	<p>Check the type of the parameter and the value of the expression.</p>
306	<p>String length out of bounds</p> <p>The result of a string expression was detected to be too long (or otherwise invalid) at compile time.</p>	<p>Correct the string expression to yield a result that is correct and of an acceptable size.</p>
307	<p>Expression out of bounds</p> <p>An expression was determined to be out of bounds at compile time.</p>	<p>Correct the expression to be in bounds appropriate for its context.</p>
308	<p>Case label out of bounds</p> <p>A case label has a value that the case selection expression cannot have.</p>	<p>Check the possible values of the case selection expression and the values of the case labels.</p>
398	<p>Implementation restriction</p> <p>This construct cannot be handled in this implementation.</p>	<p>Check manual for restrictions on the indicated construct.</p>
400 or greater	<p>Compiler error</p>	<p>Please report as a bug.</p>

NOTE

Some syntax errors are not detected/reported if code generation has been suppressed. Errors numbered 300 through 350 will not be reported, other errors may not be detected as well. Code generation can be suppressed by specifying 0 (zero) as the relocatable file name or by turning the CODE option OFF. Code generation is automatically suppressed following a procedure or function which contains any syntax errors.

Appendix B

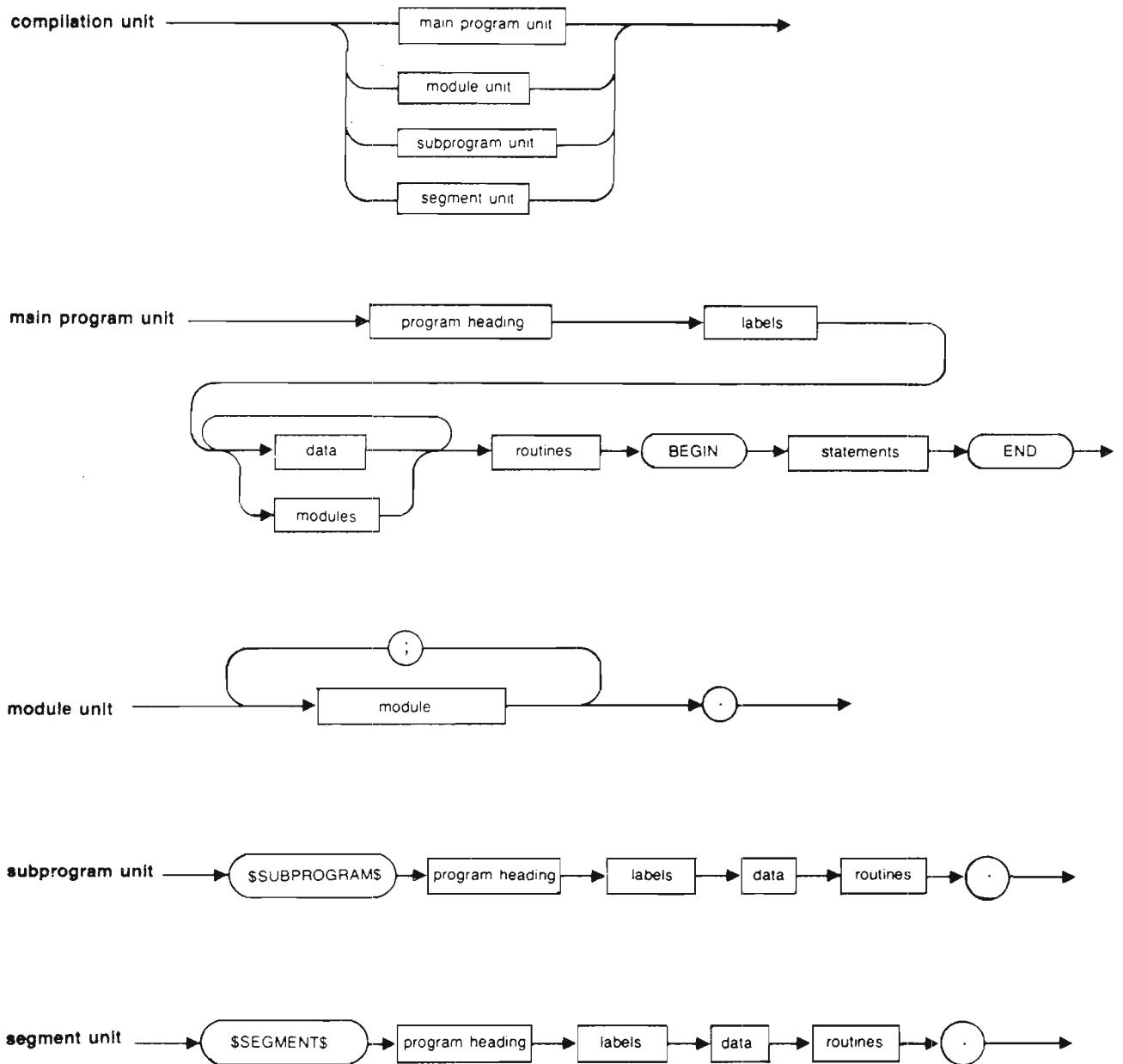
Syntax Diagrams

Table B-1. Alphabetized List and Location of Constructs

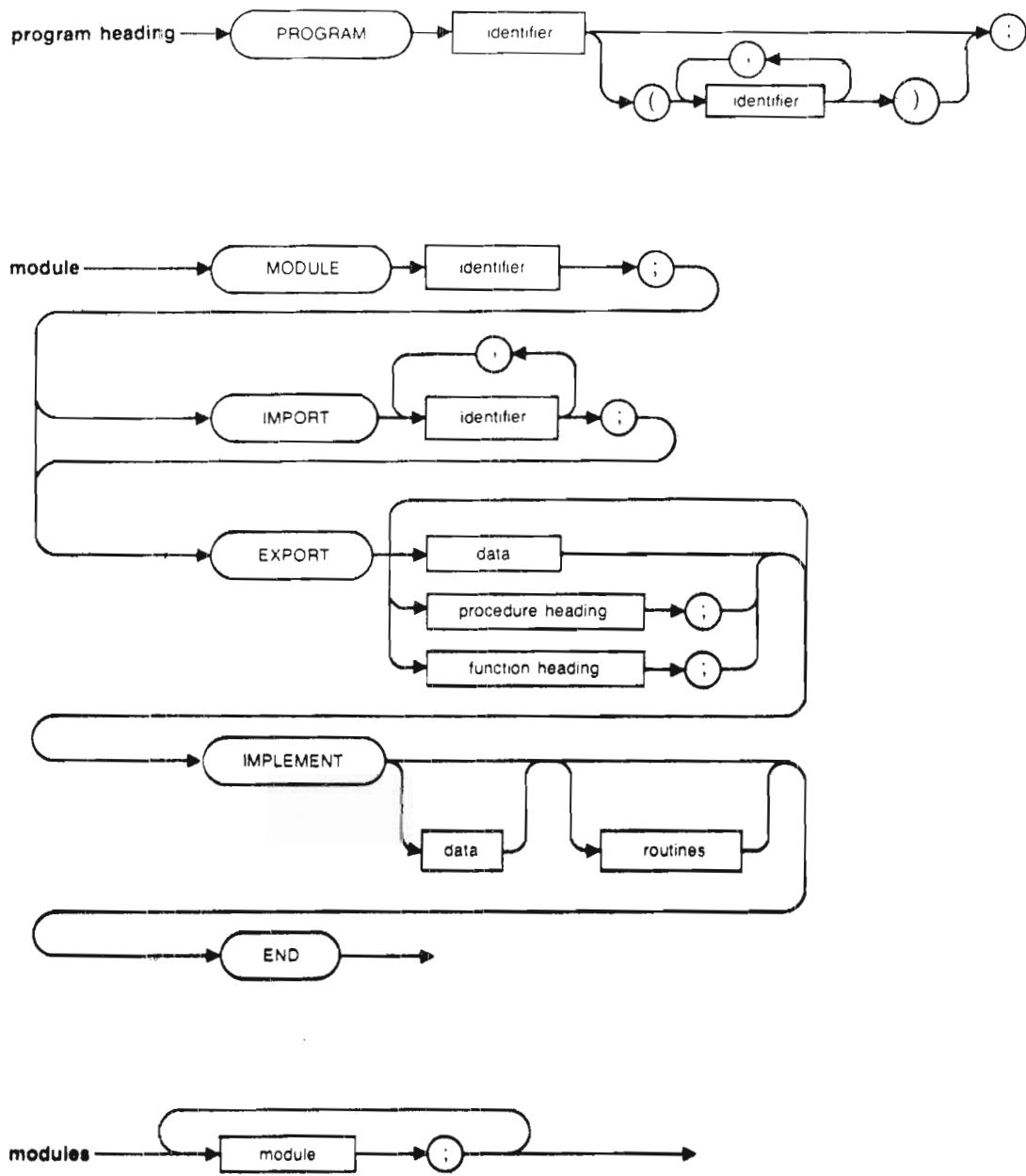
CONSTRUCT	DEFINED ON PAGE B-	USED ON PAGES B-
actual parameters	9	8,9
block	5	5
comment	11	
compilation unit	3	
compiler option	11	
constant	6	6,7
constant identifier	10	6,8
data	6	3,4,5
expression	9	6,8,9
field list	7	7
formal parameters	5	5
function heading	5	4,5
function identifier	10	8,9
identifier	10	4,5,7,10,11
integer	10	6,8,9,10,11
labels	6	3,5
main program unit	3	3
module	4	3,4
module unit	3	3
modules	4	3
procedure heading	5	4,5
procedure identifier	10	5,8,9
program heading	4	3
ranges	7	7,8
real	10	9
routines	5	3,4,5
segment unit	3	3
selector	9	8,9
statement	8	8
statements	8	3,5,8
string literal	8	9,11
structured constant	6	6
subprogram unit	3	3
type	7	6,7
type identifier	10	5,6,7,9
variable identifier	10	6,8,9

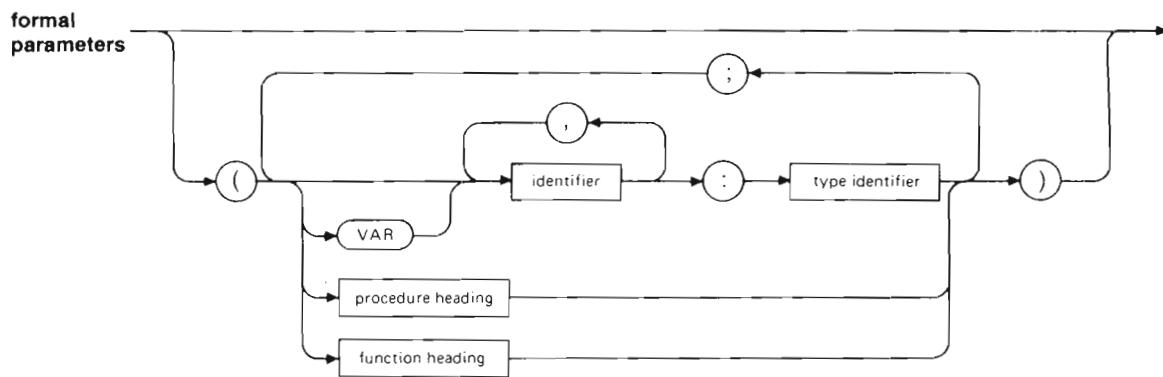
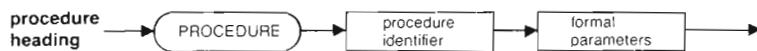
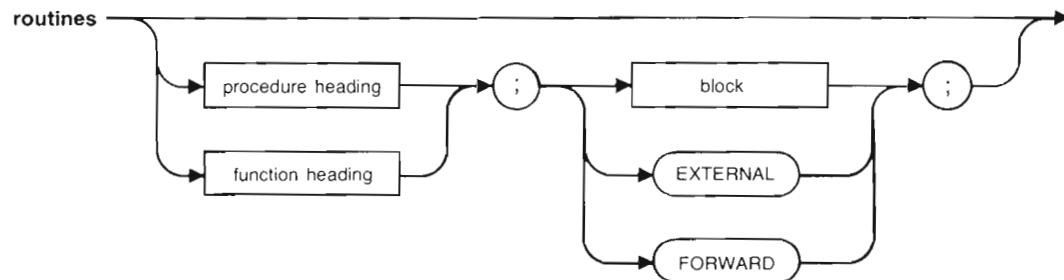
Table B-2. Syntax Diagram Organization

Page B-3 compilation unit main program unit module unit subprogram unit segment unit	Page B-4 program heading module modules	Page B-5 routines procedure heading function heading formal parameters block
Page B-6 labels data constant structured constant	Page B-7 type field list ranges	
Page B-8 statement statements	Page B-9 expression selector actual parameters	
Page B-10 integer real string literal identifier constant identifier type identifier variable identifier field identifier procedure identifier function identifier	Page B-11 comment compiler option	

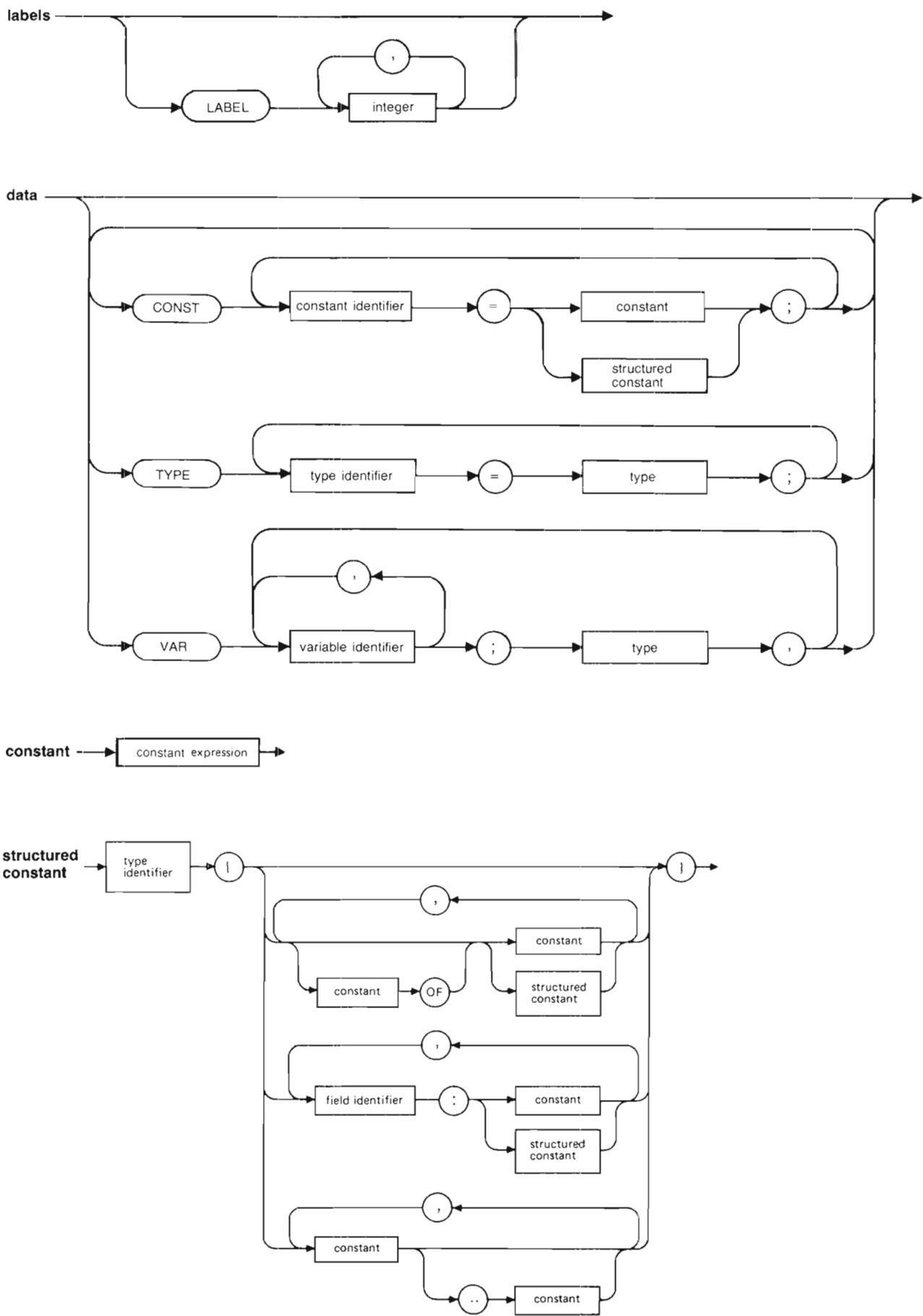


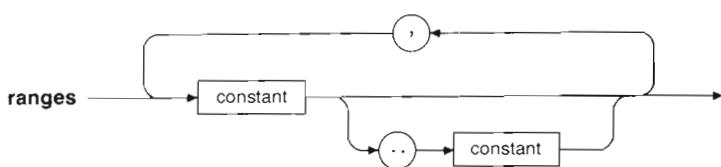
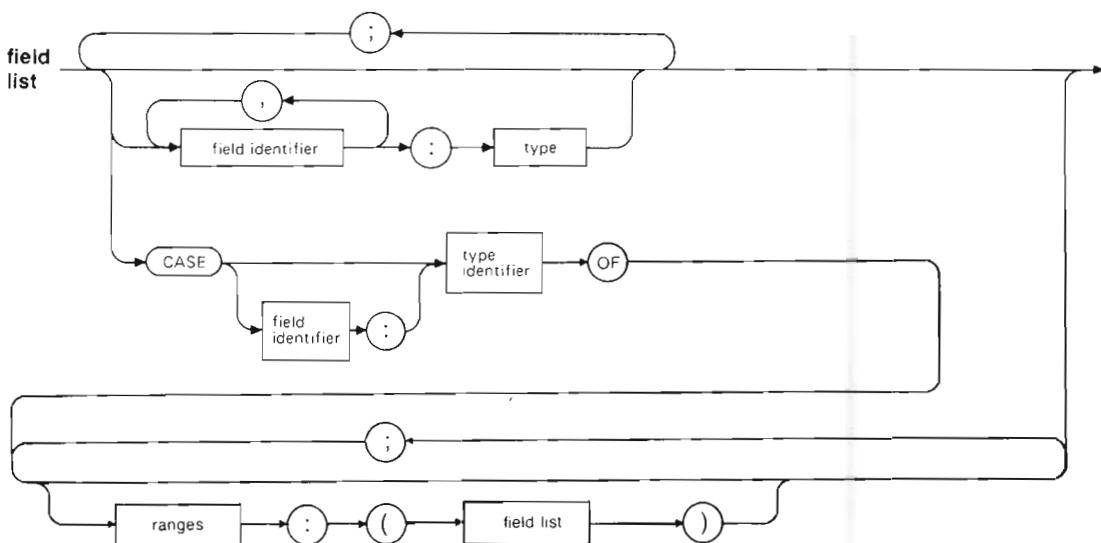
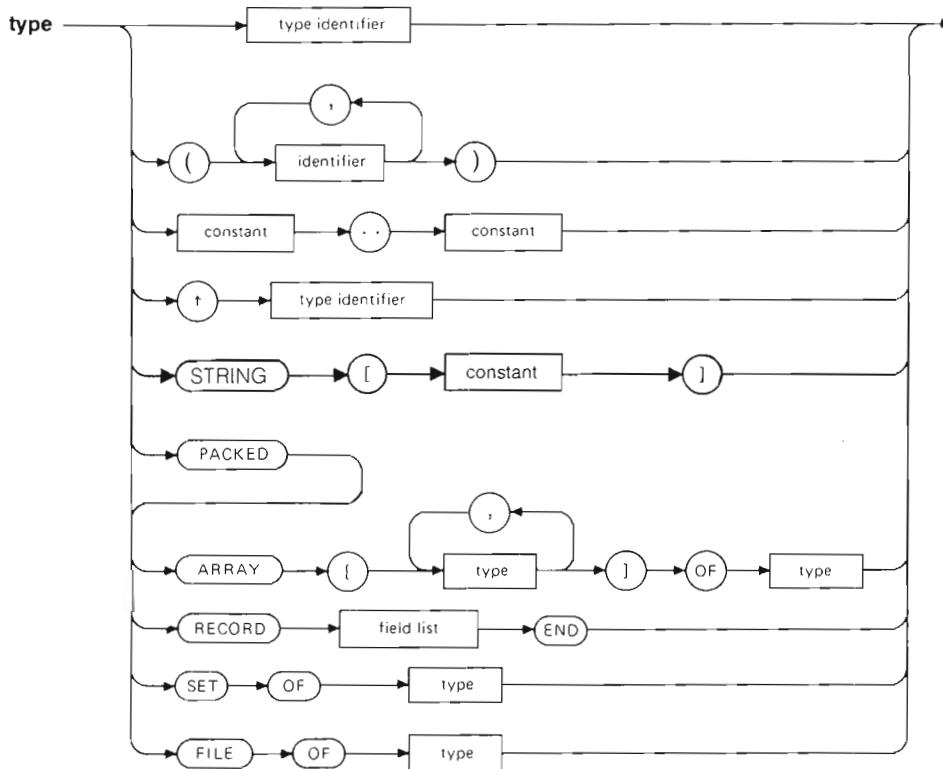
Syntax Diagrams



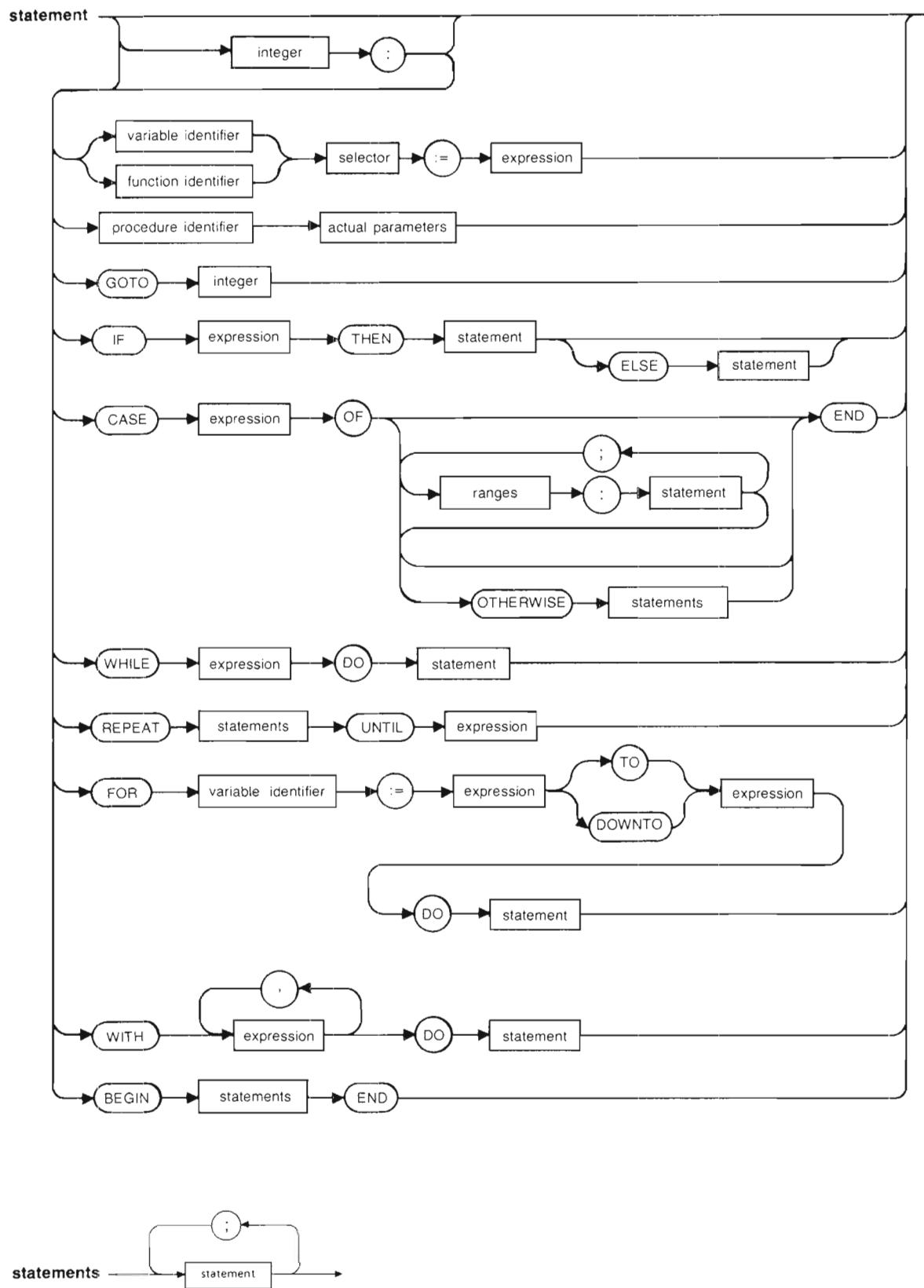


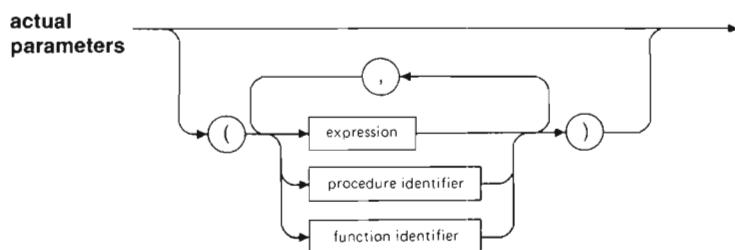
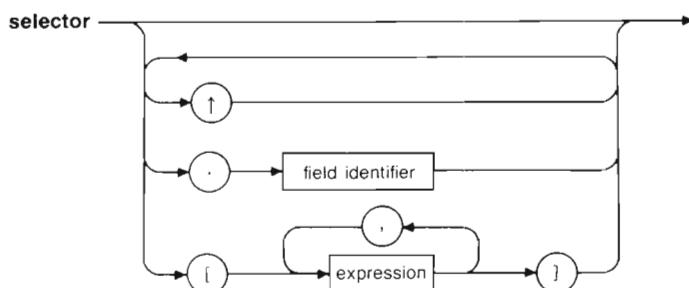
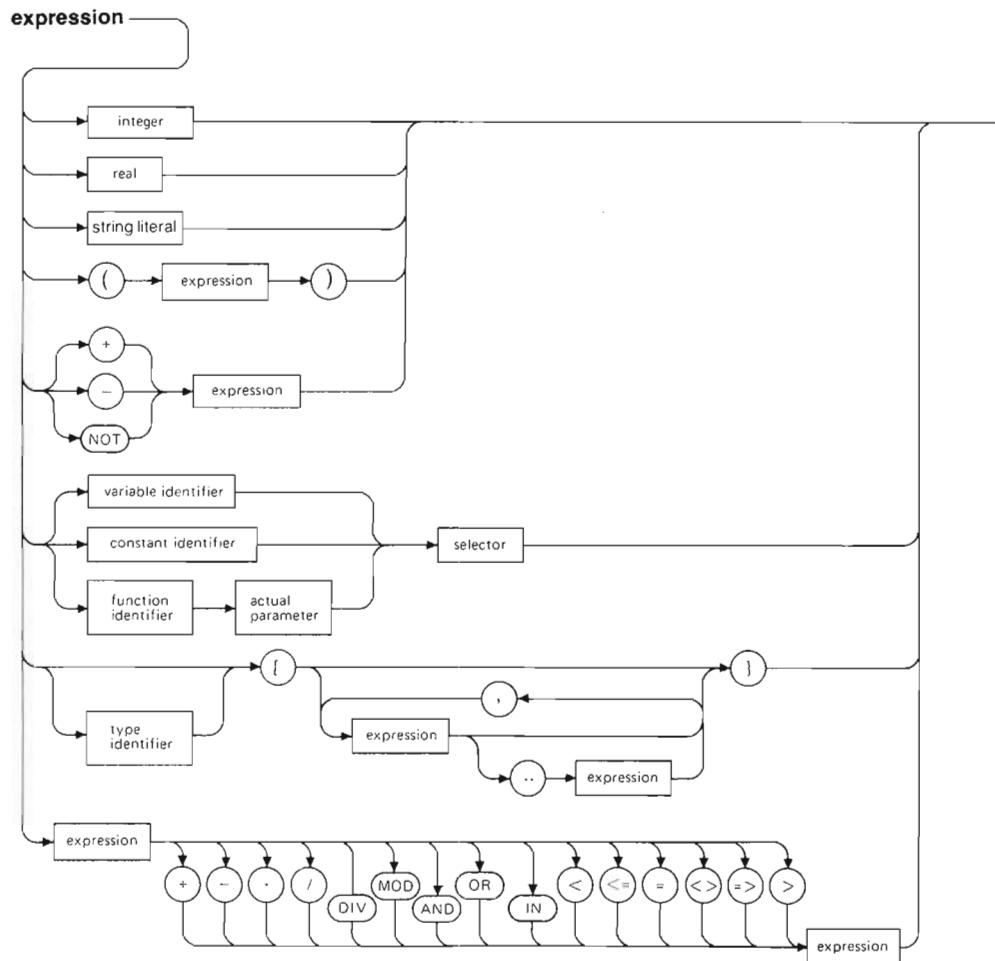
Syntax Diagrams



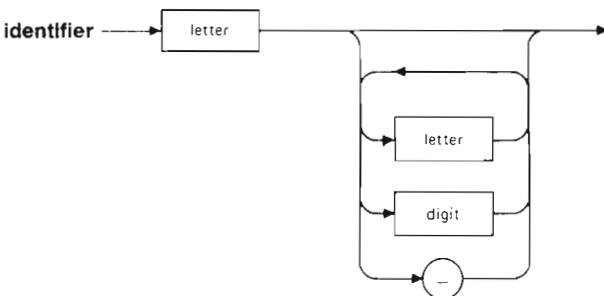
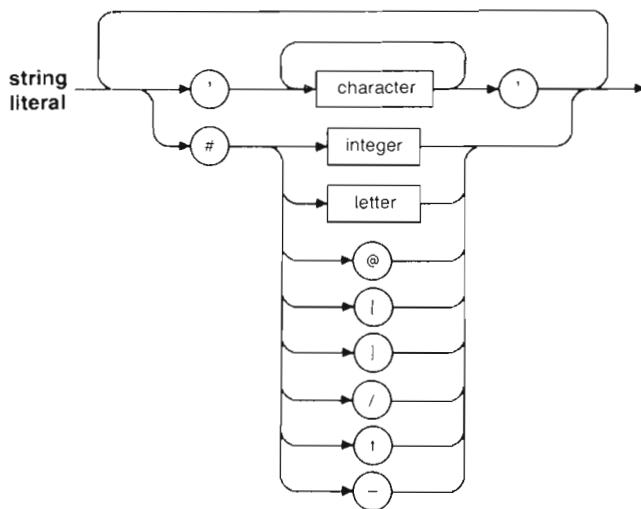
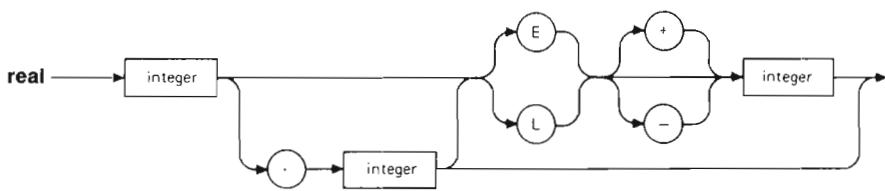


Syntax Diagrams





Syntax Diagrams



constant identifier → identifier

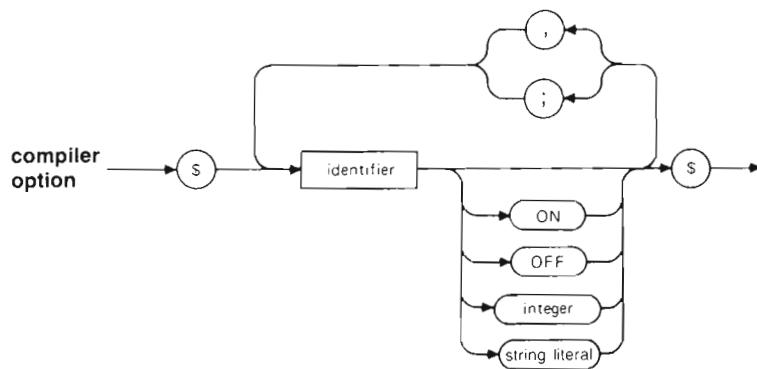
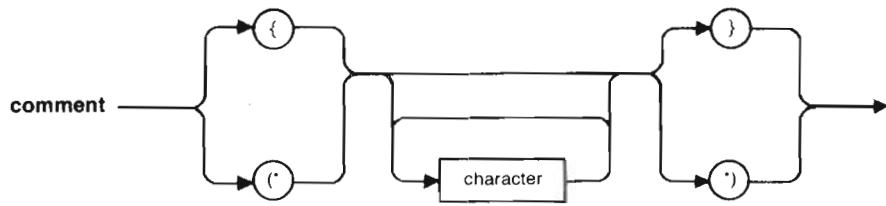
type identifier → identifier

variable identifier → identifier

field identifier → identifier

procedure identifier → identifier

function identifier → identifier

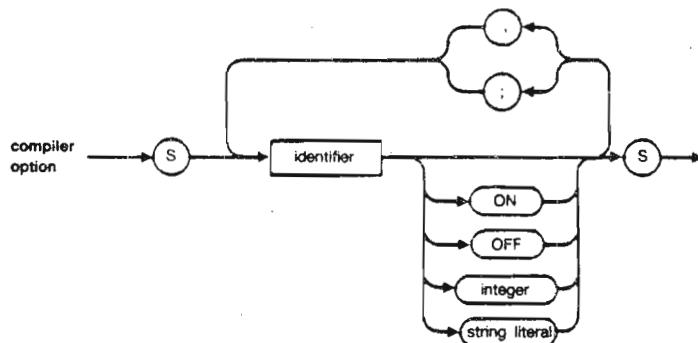


Appendix C

Compiler Options

Compiler options direct the compiler in processing the source program.

Syntax



If no option argument is specified, options which expect ON or OFF will assume ON. Options which require an integer or string argument will use their default values if the argument is omitted. The trailing \$ may be omitted, in which case the compiler option is assumed to stop at the end of the line. Compiler options can appear in one or all of the following places:

- The run string to the compiler (see Chapter 9)
- The option file (see Chapter 9)
- Anywhere in the source program that a comment is allowed

Certain options are restricted to only some of these places. Such restrictions are noted with the affected option.

Some options which apply to routines, apply to only one routine at a time, and must be respecified for each routine to which they are to apply. Some options which apply to routines, apply to all routines until the option value is changed. In the former case, the option must be specified with the appropriate value by a certain point in the source, in order to apply to a given routine. In the latter case, the value of the option is sensed at the same point at which one of the former kinds of options must have been set appropriately.

To take effect for a given procedure or function, such options must appear between the reserved word PROCEDURE or FUNCTION and the block or directive, if the routine is not being declared in the export section of a module. If the routine is being declared in the export section of a module, the option must appear between the reserved word PROCEDURE or FUNCTION and the next reserved word PROCEDURE, FUNCTION, or IMPLEMENT.

If a routine is declared FORWARD or is exported from a module, the option must appear at the FORWARD declaration or export point, not at the actual declaration of the routine.

Options which follow these rules will reference this description.

The options CODE, LIST_CODE, MIX, PARTIAL_EVAL, RANGE and STANDARD_LEVEL affect entire statements. The effect depends on the state of the option at the beginning of the statement. In particular, the effect of these options on the conditional part of a REPEAT/UNTIL loop depends on their state at the reserved word REPEAT, not at the UNTIL or conditioned part.

ALIAS <string>

The ALIAS option specifies an alternate name for a program, module, level-1, procedure, or level-1 function. This alternate name will be used, instead of the Pascal identifier, as the externally visible name of the program, module, level-1 procedure, or level-1 function.

The option applies only to the program, module, level-1 procedure, or level-1 function with which it is associated according to the following rules. Successive program objects will not have an ALIAS unless the option is respecified in the appropriate context.

To take effect for a module, the option must appear between the reserved word MODULE and the reserved word IMPORT if an IMPORT section is present or the reserved word EXPORT if no IMPORT section is present.

For procedures and functions, ALIAS follows the routine option setting/sensing rules noted above.

Blanks are removed from the specified string and alphabetic characters are shifted to upper case.

In general, the first sixteen non-blank characters of the specified string are used, exactly as specified, as the externally visible name of the entity which is being aliased. The one exception occurs when ALIAS is applied to a level-1 procedure or level-1 function which is declared within a MODULE. If the ALIAS string does not begin with the character +, the first sixteen characters are used as usual. If the ALIAS string begins with +, the compiler will generate an externally visible name for the routine which is comprised of part of the MODULE name (or MODULE ALIAS) and part of the user specified ALIAS for the procedure or function. If an ALIAS is required within a MODULE to produce a unique externally visible name, where the normal module name construction rules would generate a duplicate name, the + form is recommended to increase the chances that the resulting name is unique across the entire program. Default: No ALIAS

ANSI <ON or OFF>

ON causes a compile time warning for the use of any feature which is not part of the ANSI standard for Pascal:

`$ANSI ON$` (or `$ANSI$`) is equivalent to `$STANDARD_LEVEL 'ANSI'$`.

`$ANSI OFF$` is equivalent to `$STANDARD_LEVEL 'HP'$`

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

BASIC_STRING <ON or OFF>

This option specifies that a STRING parameter of a procedure or function which is declared EXTERNAL should be handled specially. It indicates that the parameter should be converted from the Pascal string format to the BASIC/1000C string format before being passed. In addition, if the parameter is being passed by VAR, it indicates that the Pascal string should be updated appropriately on return. If the BASIC routine modifies a string passed by value, the actual characters of the string will be modified, but the length will not be updated on return.

The option remains in effect until changed and should be set before the identifier of the string type parameter it is to effect. This option may not be set ON while the FIXED_STRING option is in effect.

The BASIC_STRING option follows the routine option sensing rules noted above. 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 or pointers to files (it 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

CDS <ON or OFF>

CDS ON specifies that code should be generated for a code and data separation environment. OFF specifies that code should be generated for a non-separated environment. Default: determined at compiler installation time.

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

NOTE

Some syntax errors are not detected/reported if code generation has been suppressed. Errors numbered 300 through 350 will not be reported, other errors may not be detected as well. Code generation can be suppressed by specifying 0 (zero) as the relocatable file name or by turning the CODE option OFF. Code generation is automatically suppressed following a procedure or function which contains any syntax errors.

CODE_CONSTANTS <ON or OFF>

ON specifies that string literals and structured constants in a CDS program, declared within a procedure or function, are to be stored in the code segment and moved to the data segment for access when the routine is entered. OFF indicates that string literals and structured constants should be stored in the data segment. Since data segment space is usually the limiting resource, CODE_CONSTANTS ON ensures that it is only used to hold constants as needed, at the expense of run-time overhead to move the constants when the routine is entered. CODE_CONSTANTS OFF eliminates the run-time overhead at the expense of having the constants reside in the data segment for the entire execution of the program. The CODE_CONSTANTS option follows the routine option sensing rules noted above. Default: ON

NOTE

Error 399 will be reported if more than 512 words of structured constants are declared in any routine compiled with both the CDS and CODE_CONSTANTS options ON. This is a temporary implementation restriction. If this occurs, either declare fewer/smaller constants, turn CODE_CONSTANTS OFF, or move some of the constants to an enclosing routine.

CODE_INFO <ON or OFF>

ON specifies that information about code, variable, and temporary usage should be displayed in the listing file at the end of each procedure or function (if LIST is ON). OFF specifies that such information should not be displayed. The value of the option can be changed at any time and is checked at the end of each routine to determine the appropriate action. The option can be used in conjunction with the SMALL_TEMPS option to optimize temporary use in CDS programs. Default: OFF

CODE_OFFSETS <ON or OFF>

ON specifies that a table of line numbers and their associated code offsets should be displayed at the end of each procedure or function or main program body (if LIST is ON at that time). OFF indicates that the table should not be displayed. The offsets are in octal, relative to the beginning of the procedure or function or main program body. The option setting is checked at the end of each procedure or function or main program body and the appropriate action taken. Default: OFF

DEBUG <ON or OFF>

Indicates that information required by Symbolic DEBUG should be put into the relocatable file. This option must appear before the program heading. Default: OFF

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 to only a single routine, and it follows the routine option setting rules noted above.

A DIRECT routine cannot be RECURSIVE, nor can it be passed as an actual parameter (an EXTERNAL routine declared with RECURSIVE ON can be declared DIRECT as RECURSIVE does not apply to EXTERNAL routines).

If the source is being compiled with the CDS option ON, the following considerations apply. The DIRECT option is ignored (and generates a warning) if it appears on an actual routine declaration (CDS code routines cannot be called using the DIRECT sequence). The DIRECT option is noted, however, on EXTERNAL declarations, and information is placed in the relocatable to indicate that the EXTERNAL routine may be in non-CDS code with a DIRECT calling sequence. No harm will be done if the EXTERNAL is in fact CDS code, all calls will be made correctly. If an EXTERNAL is in fact non-CDS code with a DIRECT calling sequence, the DIRECT option must be specified on the EXTERNAL declaration to ensure that the routine is called correctly. Default: routine is not DIRECT

EMA <string>

The <string> specifies parameters which are provided to the loader to describe EMA and MSEG allocation. The string is of the form 'ema_size,mseg_size'; for example, '65,2' or '+20,3' where ema_size is either an unsigned integer or an integer preceded by a plus, with a minimum value of 2, and mseg_size is an unsigned integer in the range 2 to 32. The ema_size is passed to the loader as an EM command embedded in the relocatable (of the form EM,ema_size), the mseg_size is passed to the loader in a MSEG relocatable record.

An unsigned ema_size specifies the total amount of EMA to be allocated. Any EMA_VARS will be allocated within this area and only the unused amount will be available for the heap/stack (heap only in a CDS program). If this form is used and the total size specified is less than that required for any EMA_VARS, the required size will be used instead (resulting in a small or non-existent heap/stack area).

A signed ema_size specifies an additional amount of EMA to be allocated above and beyond that allocated for any EMA_VARS. The entire amount specified will be available for the heap/stack (heap only in CDS programs). The total amount of EMA is the sum of any EMA_VAR areas and the amount of additional EMA specified.

For VMA programs, neither the working set (WS) size nor the virtual size (VS) are directly effected by this option. Attempting to set the virtual size to a value less than that required for the total amount of EMA (as defined above) will result in the required size being used instead. The working set size and virtual size must, in general, be set or defaulted by the user when loading the program.

Note that the loader uses the command which results in the largest total EMA size when determining actual allocation. If an EM command is provided to the loader which results in a larger total size, that specification will be used instead of the one embedded in the relocatable. The EM command which results in the largest total EMA size will determine whether a total amount or additional amount allocation is performed.

For compatibility with earlier versions of the compiler, the specification '0,0' is permitted and is treated as '+32,2'. No other error checking is performed and errors may be reported by MACRO or the loader if invalid values are provided.

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.

If this option is not used, the compiler does not emit the EM record and it emits an MSEG record with a value of 2. The loader will default the EMA or VMA size as appropriate. Default: 'default,2'

EMA_VAR <ON or OFF>

ON specifies that subsequent globally declared variables or variables declared within the EXPORT or IMPLEMENT part of a module, should be placed in the extended memory area (EMA or VMA). This saves space in the programs partition (data partition in CDS programs) at the expense of slower access to variables. OFF specifies that such variables should be placed in the programs partition (data partition in CDS programs) as usual.

The option is checked when each variable name is encountered in a VAR declaration and the variable is assigned to the appropriate area.

If this option is used on globally declared variables, it must be used identically in each SUBPROGRAM or SEGMENT which has access to the global variables.

The standard files INPUT and OUTPUT are not effected by the EMA_VAR option (they are allocated as if EMA_VAR was OFF).

Variables placed in the extended memory area have the same implementation restrictions as dynamic variables in the heap (in \$HEAP 2\$) with respect to assignment, comparison, and passing as parameters.

This option functions at all settings of the HEAP option (0,1,2). Default: OFF

ERROREXIT

Specifies that an external routine's calling sequence includes an error return (a call to Pas.ErrorExit 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. This option cannot be specified if the NOABORT option is in effect. Default: routine does not have ERROREXIT

FAST_REAL_OUT <ON or OFF>

ON indicates that a faster and less precise routine should be used when writing REAL or LONGREAL expressions to TEXT files or strings (write or strwrite). OFF indicates that the slower and more precise routine should be used. The option is noted at the end of each REAL or LONGREAL expression being written (before the field width specification, if any) and can be changed at any time. Default: OFF

FIXED_STRING <ON or OFF>

This option indicates that a STRING parameter of a procedure or function which is declared EXTERNAL should be handled specially. It indicates that the parameter should be converted from the Pascal variable string format to a fixed string format (also known as the FTN7X string format, which is compatible with the file system access routines) before being passed. The current length of the Pascal string is used as the length of the fixed string.

VAR parameters declared FIXED_STRING ON in HEAP 2 programs will probably have to be declared HEAPPARMS OFF as well (unless the receiving routine expects EMA/VMA addresses). If the actual string passed to such a HEAPPARMS OFF parameter is actually in the heap and has a current length greater than 2044, the results are unpredictable.

The option remains in effect until changed and should be set before the identifier of the string type parameter it is to effect. This option may not be set ON while the BASIC-STRING option is in effect.

The FIXED_STRING option follows the routine option sensing rules noted above. Default: OFF

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 VMA/EMA. Pointers are two-word offsets from the start of VMA/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 VMA/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). See the Section on Pas.InitMemInfo1 and Pas.InitMemInfo2 in Appendix D for information regarding initialization of the appropriate heap/stack area or areas.

Programs compiled using the HEAP 2 option can only be run on an RTE-6/VM or RTE-A system.

With HEAP 2, objects larger than 1024 words in the heap may not be assigned or passed as value parameters. Default: 1

HEAP_DISPOSE <ON or OFF>

HEAP_DISPOSE ON indicates that a call to dispose will create free space in the heap that can be reused by a subsequent call to new. HEAP_DISPOSE OFF indicates that disposed storage will not be reused (although the disposed pointer will be set to nil), and that mark/release will provide the only heap management.

HEAP_DISPOSE OFF is equivalent to loading a Pascal program, in the past, with the Short Heap Stack Library (\$SHSLB). If this option is specified, it must appear before the program heading. Default: ON

HEAPPARMS <ON or OFF>

The HEAPPARMS option controls the parameter passing mechanism for VAR parameters. It is only effective in compilation units which have the \$HEAP 2\$ option in effect, it is ignored in other compilation units.

HEAPPARMS ON specifies that a two-word address for the actual parameter should be passed to the routine. This is the default (normal) case for VAR parameters in \$HEAP 2\$ programs. Any variable of the appropriate type (local, global, EMA_VAR, or dynamic) can be passed to such a VAR parameter.

HEAPPARMS OFF specifies that a one-word address for the actual parameter should be passed to the routine. If the called routine was compiled either \$HEAP 0\$ or \$HEAP 1\$, or is in another language which does not expect a two-word address, or is a file system or operating system routine which does not expect a two-word address (most expect one-word addresses), HEAPPARMS OFF must be used to ensure that one-word addresses are passed.

HEAPPARMS OFF can also be used for efficiency reasons, if it is known that the actual parameter will never be an EMA_VAR or a dynamic variable, as one-word address parameter accessing is faster than two-word address parameter accessing.

A two-word-addressable variable is one of the following: (everything else is a one-word-addressable variable)

- A dynamic variable in a Heap 2 program (resides in EMA).
- An EMA_VAR, (variable declared while \$EMA_VAR ON\$, resides in EMA).
- A formal VAR parameter declared while \$HEAPPARMS ON\$



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.

The HEAPPARMS option follows the routine option sensing rules noted above. Default: ON

Compiler Options

Consider the program:

```
$HEAP 2$  
PROGRAM show_addresses;  
VAR  
  ptr1: ^INTEGER;  
  ptr2: ^INTEGER;  
  arr: ARRAY [1..10] OF INTEGER;  
  i,  
  j: INTEGER;  
  
PROCEDURE proc  
  $HEAPPARMS OFF$  
  (VAR p1: INTEGER;  
   VAR p2: INTEGER);  
  $HEAPPARMS ON$  
  
BEGIN  
  ...  
END;  
  
PROCEDURE show_it  
  (VAR var_parm: INTEGER);  
  
BEGIN  
  new (ptr1);  
  new (ptr2);  
  
  ptr1^ := 5;  
  ptr2^ := 6;  
  a [ptr1^] := ptr1^;  
  a [ptr2^] := ptr2^;  
  
  proc (i,j);  
  proc (ptr1^, j);  
  proc (ptr1^, ptr2^);  
  proc (a [ptr1^], ptr2^);  
  proc (var_parm, ptr2^);  
END;  
  
BEGIN  
  show_it (ptr1^);  
END.
```

Mapping the EMA/VMA area on the HP 1000 consists, very briefly, of the following characteristics:

- Under most circumstances, only two contiguous 1024KW pages may be *mapped in* at any one time.
- Mapping in a data item consists of mapping in the page containing the first word of that item, plus the page following it.
- Once a data item is mapped in, it must be manipulated using its mapped-in, one-word address. Once the map changes to map in another item in EMA/VMA, the original item can not be manipulated until it is mapped in again.

In Pascal, a two-word-addressable variable is manipulated by mapping the two-word address into a one-word address, and then accessing the variable.

A two-word addressable variable can be passed to a HEAPPARMS OFF VAR parameter (the compiler maps it in, and passes its one-word address), BUT there are some restrictions:

- a. If the actual variable is in fact in EMA/VMA, only the first 1024 words of the variable (if it is larger than 1024 words) are guaranteed to be accessible within the called routine.
- b. At most one two-word-addressable variable can be passed to a HEAPPARMS OFF VAR parameter in any given routine call. This is because HEAPPARMS OFF VAR parameters must be mapped into one-word addresses prior to the call. Mapping a second HEAPPARMS OFF VAR parameter would invalidate the map set up for the first such parameter. The compiler generates a syntax error when this condition is detected.
- c. Within the called routine, the actual parameter cannot be accessed safely if, prior to its access, the routine causes the maps to be altered by any means:
 - pointer dereferencing
 - accessing an EMAVAR variable
 - accessing a HEAPPARMS ON VAR parameter whose actual parameter is in EMA/VMA,
 - invoking a heap management routine (e.g. new, release, etc)
 - doing recursion in a non-CDS environment
 - calling other routines which alter the maps

IF THE MAPS ARE ALTERED PRIOR TO ACCESSING THE PARAMETER, THE PROGRAM WILL NOT WORK CORRECTLY. THIS IS AN ERROR THAT CANNOT BE DETECTED EITHER AT COMPILE TIME NOR RUN TIME.

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>

All blanks are removed from the <string> before the file specification is used.

If the file specification does not specify a global directory or FMGR cartridge, it will be searched for in the following places:

1. In the directory of the file currently being processed (which is the source program if this is a first level INCLUDE, or the current INCLUDE file if it is a nested INCLUDE)
2. In the current working directory (if any)
3. In the global directory /INCLUDES/ (if present)
4. On FMGR cartridges (if any)

INCLUDE-DEPTH <integer>

This option specifies the maximum depth of include/import file nesting that the compiler should permit (range 0 to 100). Large values may limit the size of the program that can be compiled or may adversely impact compiler performance. Values below the default will be enforced by the compiler but will not affect compiler performance. This option must be set before the program heading (the default is in effect until the program heading). Default: 10

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. If the source is a file system file, the assembly file name is the source file name with .MAC replacing .PAS. If the source is a FMGR file, the assembly file name is the source file name with a (^) replacing the initial (&). If the source does not start with (&), the assembly file name is the source file name with a (^) prefixed (possibly dropping the last character). Default: OFF

LINES <integer>

The <integer> (range 5 to 32767) specifies the number of source lines to print per page of source listing. The default is 56 (which includes three lines of heading). One line is always left blank at the top of the page (before the program heading) and with the default setting, three lines are left blank at the end of a 60 line page. Default: 56

LINESIZE <integer>

The <integer> (range 1 to 32000) 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. The LINESIZE option does not effect procedure or function parameters of type TEXT or of types which contain TEXT files. 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)

NOABORT

This option indicates that an external procedure has a noabort error return. It does not set the noabort parameter, that is the program's responsibility.

If the program sets the noabort bit on the call (the *normal* case when this option is in effect), then on an error return the statement following the call (which can be any Pascal statement or compound statement) will be executed, and on a normal return the statement following the call will be skipped.

If the program does not set the noabort bit on the call, then there is no error return (the program aborts on an error) and the statement following the call will be executed on a normal return (not skipped).

A procedure declared with the noabort option in effect, cannot be called if the following statement involves a flow of control change or is otherwise ambiguous (the call is at the end of a THEN part, ELSE part, or WHILE loop, for example).

This option should always be used to recover control after a noabort call. The results of using either a GOTO or a parameterless direct procedure call to recover control, which were suggested for earlier versions of Pascal/1000, are unpredictable. In addition, the compiler does not guarantee to preserve the registers when calling a Pascal procedure, so the error return statement can't call a Pascal procedure which in turn calls ABREG. To recover the registers on an error return, the following is suggested:

```

TYPE
  INT = -32768..32767;
  MES = PACKED ARRAY [1..6] OF CHAR;

VAR
  areg, breg: INT;

PROCEDURE sys_write
  $ALIAS 'EXEC', NOABORT'$
  (icode: INT; icnwd: INT; ibfr: MES; ilen: INT); EXTERNAL;

PROCEDURE abreg
  (VAR a: INT; VAR b: INT); EXTERNAL

BEGIN
  {Set noabort and Write to LU 1.}
  sys_write (-32768 + 2, 1, 'Hello!', -6);

  BEGIN {Error return comes to here.}
    abreg (areg, breg);
    {Call your routine here, it can get the register}
    {contents from the global variables areg & breg.}
  END;

  {Normal return comes to here.}

```

This option cannot be specified if the ERROREXIT option is in effect. Default: routine does not have a noabort error return.

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>

The optional <string> is used to place information in the NAM record of a program unit. If the first character of the string is a comma then the string specifies the contents of the NAM record 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 30 characters of the comment part of the string will be used in either case.

The compiler will use the current value at the time it emits any NAM record. NAM records are generally emitted when the first declaration in a block is encountered (LABEL, CONST, TYPE, VAR, PROCEDURE, or FUNCTION). If no declarations are present, the NAM is emitted when the BEGIN is encountered.

In addition, a NAM is emitted when the first level-1 routine is encountered in a main program and when the BEGIN of the main program block is encountered. The former contains DEBUG and MODULE information, if necessary, but is always present even if empty. The latter contains the main program code, data, and associated information.

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.

Additional care must be taken when the first form is used in a SEGMENT unit. The Pascal option which is used on the NAM for the overlay header is the one in effect just before the first global declaration (LABEL, CONST, TYPE, VAR, PROCEDURE or FUNCTION). If the Pascal option in effect at that time specifies a program type, it must specify type-5 (an overlay). If such a Pascal option is used, the option must be changed before the declaration section or block of the first level-1 routine to avoid making any level-1 routine type-5. It can be changed to a default value by specifying no argument, or to a comment only value, or to a value which specifies a type which is appropriate for a level-1 routine in an overlay (usually type-7).

PRIVATE_TYPES

This option specifies that all types defined from this point or in the compilation (whether explicitly with a TYPE definition, or implicitly in a VAR declaration) are to be considered by Symbolic DEBUG to be local to this compilation unit. This option takes no parameters.

In a multiple-source program, Symbolic DEBUG uses the global type information from the main program only (to save space). However, if any global types appear in a SUBPROGRAM or SEGMENT, Symbolic DEBUG will check them for consistency with the main program global type information (i.e., that the globals match).

PRIVATE_TYPES is normally used when compiling a library with DEBUG that will be used by more than one program. It is placed at the beginning of such a unit and makes all the *global* types *local* to the library. This keeps Symbolic DEBUG from complaining about inconsistencies in global type information.

PRIVATE_TYPES can also be used to add additional types to a compilation unit that is part of a multiple-source program. It is placed after the shared globals (usually at the end of the include file which contains the shared globals) and permits additional types to be declared in that unit without causing inconsistency errors in Symbolic DEBUG.

Due to debug requirements, the PRIVATE_TYPES option will be implicitly set at the first MODULE declaration within the global declarations. This can cause type inconsistency errors if the program contains SUBPROGRAM and SEGMENT units (because such units cannot contain MODULE declarations in their global declarations). This difficulty can be avoided if modules are imported into the global declarations from a separate compilation unit instead of being declared in the global declarations, or if all module declarations come at the end of the global declarations. Default: end of global declarations or first module declaration within the global declarations.

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 that range-checking is always performed for set expressions upon assignment and for most string operations 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. The RECURSIVE option follows the routine option sensing rules noted above.

The RECURSIVE option has no effect on routines which are declared EXTERNAL and is ignored in sources which are compiled with the CDS option ON. Default: ON

RESULTS <string>

The <string> names a file to which the results of the compilation are to be appended. At the start of compilation a line of the form:

```
Pascal: Compiling SOURCE, Date/Time
```

is displayed, where SOURCE is the source file name, and Date/Time is the current date and time. Any lines which contain syntax errors will be listed. At the end of the compilation, a decription of each of the errors encountered (like that normally provided at the end of the listing file) will be produced, followed by a line of the form:

```
Pascal: X errors[, Y warnings], SOURCE, Date/Time
```

The *warnings* count will not be displayed if there were no warnings. The Date/Time will be the date and time at which the compilation finished.

The RESULTS option can only be specified in the option file or on the run string. When the listing is going to a file, the RESULTS option with a <string> of '1' can be used to watch for syntax errors during the compilation.

If more than one compiler is posting RESULTS to the same file, the final contents of the RESULTS file is unpredictable. Default: No RESULTS file

RUN_STRING <integer>

The integer (range 0 to 32767) specifies the maximum number of characters of run string information that will be saved by the Pascal startup routine Pas.Initialize. This saved information is used by the Pascal I/O package to retrieve the file names for files declared in the program heading, and is available to the user program via the routines Pas.Parameters and Pas.Sparameters (see Appendix D).

If the integer value is zero, the Pascal routines Pas.Initialize and Pas.GetNewParms will not do an EXEC 14, and the program will have to do its own EXEC 14 to recover run string information. The Pascal I/O system will not be able to find names for any files which are specified in the program heading and are opened without an explicit name, nor will the routines Pas.Parameters and Pas.Sparameters be able to return any information.

The maximum number of characters is limited by the amount of system available memory (SAM) available for the passing of run-string buffers. FMGR passes only 80 characters to the programs it schedules, while other programs may pass more. Insufficient SAM will cause the scheduling program to be suspended (if SAM could ever become available) or aborted (if SAM will never be available because the required amount does not exist). Default: 80

SEARCH <string>

The string literal specifies a comma separated list of relocatable files or libraries to be searched to resolve IMPORT specifications that cannot be resolved within the current compilation unit. The list may end with a comma and blanks within the <string> are ignored when the file specifications are used.

If the first non-blank in the <string> is not a plus '+' character, the specified SEARCH list replaces any previous (possibly catenated) search lists.

If the first non-blank character is a plus '+', the specified SEARCH list is logically catenated on the end of any existing search lists. The plus can be followed by either a comma or the first file specification. There is no arbitrary limit on the number of search strings which can be catenated.

Search files which do not reference a global directory or FMGR cartridge will be searched for in the same places as INCLUDE files (see INCLUDE), with the exception that the third place searched will be the global directory /LIBRARIES/ (instead of /INCLUDES/).

Example

```
SEARCH 'XMYMOD:::-10,/LIBRARIES/GRAPH.MOD,XSYMOD::QQ'
```

The SEARCH option must be specified at the global or module level (outside of any procedure or function declaration). Default: No SEARCH list

SEGMENT

Specifies that the current compilation unit is a segment overlay, rather than a subprogram or main program. Note that this option is used only for programs which use single-level segmentation. 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. If this option is specified, it must appear before the program heading.

The SEGMENT option will elicit a warning at STANDARD_LEVEL 'ANSI', ANSI ON, ANSI OFF, or STANDARD_LEVEL 'HP', as it is a Pascal-1000 extension.

CAUTION

A type-5 relocatable is generated for the segment entry point, and type-7 relocatables for any/all level-1 routines in the segment. LOADR, upon encountering the first segment (type-5) relocatable, will search after it for any routines that it can load with the main, before it relocates the segment. Since the usual reason for having a segment is to not have routines loaded with the main, this can cause difficulty. Two simple solutions are, to use LINK if it is available and appropriate, or to compile and relocate an empty segment as the first segment. This empty segment will be immediately followed by the first real segment and LOADR will not find any type-7 relocatables between them that it can load into the main.

Default: Compilation unit is a main program or module list

SEGMENTED <ON or OFF>

ON specifies that a main program unit is a non-CDS program which uses the SEGMENT/overlay mechanism. It indicates that the initialization code that is generated for modules which contain files is such that modules relocated in the main program are initialized at program start and those relocated into overlays are initialized each time the overlay is loaded.

OFF indicates that all modules known to the main program are relocated in the main program and that all modules known to an overlay are relocated in that overlay. If this is not true, and the main knows about modules in overlays, or an overlay knows about modules in the main, the results will be unpredictable.

The option is ignored in SUBPROGRAM and SEGMENT units and when CDS is ON. Default: OFF

SKIP_TEXT <ON or OFF>

ON specifies that source code is to be ignored until the next SKIP_TEXT OFF is encountered. OFF specifies that source code should be processed. Default: OFF

SMALL_TEMPS <integer>

The <integer> (0 to 700) specifies the number of words to reserve in local data space for *small* temporaries in CDS programs. The number of small temporaries used by the compiler in any given routine is a function of the complexity of expressions in the routine. This value is a compiler internal parameter that normally does not need to be modified. The compiler uses a default value which is adequate for most routines. The option is provided so that routines which use fewer than the default number of small temporaries can minimize their use of local data space and routines which use more than the default number can be compiled without eliciting syntax error 262.

The CODE_INFO option can be used to determine the amount of small temporary space actually used by a given routine.

The SMALL_TEMPS option follows the routine option sensing rules noted above. Default: 25

STANDARD_LEVEL <'ANSI', 'HP', or 'HP1000'>

Specifies the level of syntax that the compiler will process normally. If a feature is encountered which is not acceptable at the current STANDARD_LEVEL, a warning is produced and the feature is compiled normally.

The current setting of STANDARD_LEVEL applies when a module is being imported from a separately compiled source and warnings may be issued if things being imported (either explicitly or implicitly) would elicit such warnings if they were actually declared in the source which is doing the importing.

Note that the 'HP1000' level permits comments with alternate delimiters to nest. Comments do not nest at any other level.

The string can be specified in either upper or lower case.

Note that ANSI or ANSI ON is equivalent to STANDARD_LEVEL 'ANSI' and that ANSI OFF is equivalent to STANDARD_LEVEL 'HP'. Default: 'HP'

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

300 Pages of VMA,
50 Pages of workspace requested.
12 Pages of workspace used.
```

```
Compilation started: Tue Dec 20, 1983 9:49 pm
Compilation completed: Thr Dec 20, 1983 9:50 pm
```

COMPILER OPTIONS:

ANSI	Off
Asmb	R
Autopage	Off
BASIC_String	Off
Buffers	1
CDS	Off
Code	On
Code_Constants	On
Code_Info	Off
Code_Offsets	Off
Debug	Off
EMA	0,0
EMA_Var	Off
Fast_Real_Out	Off
Fixed_String	Off
Heap	1
Heap_Dispose	On
HeapParms	On
IdSize	150
Image	0
Include_Depth	10
KeepAsmb	Off
Line_Info	On
Lines	56
LineSize	128
List	On
List_Code	Off

Mix	Off
Partial_Eval	On
Pascal	
Private_Types	Off
Range	On
Recursive	On
Results	
Run_String	80
Search	
Segmented	Off
Skip_Text	Off
Standard_Level	HP
Stats	On
Subtitle	
Tables	Off
Title	Sample Program
Trace	0
Trace_Back	On
Unit	Subprogram Library
Warn	On
Width	80
Work	50
XREF	Off

The *Dynamic Memory Allocation* information tells the user that the compiler had 300 pages of VMA available to it, and at least 50 pages were available for the compiler's workspace. These parameters can be adjusted by the VS command and the WORK option, respectively (see the Pascal Configuration Guide for further information). The workspace requested value is always that specified by the WORK option (or defaulted). When the compiler itself is a CDS program, the WORK option has no effect, but its value will still be displayed by the STATS option as the amount of workspace requested. Default: No STATS display

When using RTE-XL, the VMA information is not displayed.

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 will be produced for each level-1 routine in the subprogram and the result will be a searchable library of routines. If this option is specified, it must appear before the program heading.

The SUBPROGRAM option will elicit a warning at STANDARD_LEVEL 'ANSI', ANSI ON, ANSI OFF, or STANDARD_LEVEL 'HP', as it is a Pascal/1000 extension. Default: Compilation unit is a main program or module list

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 0 : PROGRAM example {input};
2 0 :
3 0 : program to demonstrate the different kinds of table entries
4 0 :
5 0 :
6 0 : CONST
7 0 :     exclamation = '!';
8 0 :     perfect      = 8128;
9 0 :     pi           = 3.1415926;
10 0 :    big          = 123456789L10;
11 0 :    pac          = 'A PAC indeed';
12 0 :
13 0 : TYPE
14 0 :     POSINT      = 0..32767;
15 0 :     CHARSET     = SET OF CHAR;
16 0 :     ANARRAY     = PACKED ARRAY [1..100] OF BOOLEAN;
17 0 :     ARECORD     = RECORD
18 1 :         one: INTEGER;
19 1 :         CASE two: POSINT OF
20 1 :             10: (ten: POSINT);
21 1 :             20: (twenty: REAL);
22 1 :             30: (thirty: LONGREAL);
23 0 :         END;
24 0 :     ASTRING      = STRING [128];
25 0 :
26 0 : CONST
27 0 :     frano = ANARRAY [50 of false, 50 of true];
28 0 :
29 0 : VAR
30 0 :     once:  CHARSET;
31 0 :     upon:  TEXT;
32 0 :     atime: ARECORD;
33 0 :     year:   1900..2000;
34 0 :     era:    (past, present, future);
35 0 :
36 1 : PROCEDURE donotdo;
37 1 1 BEGIN {donotdo}
38 1 1 END; {donotdo}
39 0 :
40 0 : $TABLES ON$ {only want main program tables}
41 0 1 BEGIN {example}
42 0 1 END. {example}

```

Global Identifiers

ANARRAY	<19> Type <19> 6/ 4 Packed Array <20> of Boolean
	<20> 0/ 7 Subrange 1..100 of <1>
	<1> 1/ 0 Scalar Standard
	<9> 0/ 1 Boolean
ARECORD	<21> Type <21> 7/ 0 Record
	ONE <2> Field 0/2 <2> 2/ 0 Integer
TEN	<17> Field 3/1 <17> 0/15 Subrange 0..32767 of <1>
THIRTY	<3> Field 3/4 <3> 4/ 0 Longreal
TWENTY	<4> Field 3/2 <4> 2/ 0 Real
TWO	<17> Field 2/1 <22> 3/ 0 Tag <17>
	<23> 4/ 0 Variant 10 <24> 5/ 0 Variant 20
	<25> 7/ 0 Variant 30
ASTRING	<26> Type <26> 66/ 0 String [128]
ATIME	<21> Actual Variable Pas.1+240
BIG	<3> Constant 1.2345678900000000L+18
CHARSET	<18> Type <18> 17/ 0 Set of Char
	<5> 0/ 8 Char DONOTDO Declared Procedure Actual Recursive
ERA	<29> Actual Variable Pas.1+248
	<29> 0/ 2 Scalar Declared
EXCLAMATION	<5> Constant '!' <19> Constant Array Pas.3+0
FRANO	<29> Constant 2
FUTURE	<13> Actual Variable Pas.4+0
INPUT	<13> 223/ 0 Text
	<18> Actual Variable Pas.1+0
ONCE	PAC <15> Constant String Pas.2+3
	<15> 6/ 0 Packed Array <16> of Char
	<16> 1/ 0 Subrange 1..12 of <1>
PAST	<29> Constant 0
PERFECT	<1> Constant 8128
PI	<4> Constant 3.141593L+00
POSINT	<17> Type
PRESENT	<29> Constant 1
UPON	<27> Actual Variable Pas.1+17
	223/ 0 Text
YEAR	<28> Actual Variable Pas.1+247
	<28> 0/11 Subrange 1900..2000 of <1>

A TABLES listing contains two types of entries: identifier entries and type entries. The first line of the example is an identifier entry:

ANARRAY <19> TYPE

An identifier entry has three fields. The first is the identifier itself. If an identifier 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 <19>, and it is used to locate the type entry for the identifier. The third field is the class of the identifier, 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:

<19> 6/ 4 PACKED ARRAY <20> OF <9>

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 <20> and an element type of <9>.

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 three kinds of labels. If a structured constant or variable is global it is located at an offset from a label of the form Pas.DDDD. If it is not global and therefore cannot be accessed from other compilation units, it is located at an offset from a label of the form .DDDD in a non-CDS program and from the label Q in a CDS program since non-globals are allocated relative to the current stack frame, which is designated Q. The offset is a decimal value.

Structured constants declared in a procedure or function in a source compiled with the CDS option ON and CODE_CONSTANTS ON, will have the location of the data space copy of the constant displayed. Default: OFF

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..255;
  LEN     = 1..64;
  STR64  = PACKED ARRAY [1..64] OF CHAR;

PROCEDURE trace_init   {Call emitted at start of program      }
  $ALIAS 'Pas.TraceInit'$$
  (trace_lu: LU);      {LU of trace output                  }
  EXTERNAL;

PROCEDURE trace_close  {Call emitted at end of program      }
  $ALIAS 'Pas.TraceClose'$$
  EXTERNAL;

PROCEDURE trace_begin   {Call emitted at start of each routine}
  $ALIAS 'Pas.TraceBegin'$$
  (name_len: LEN;        {Length of routine name            }
  name    : STR64);     {Name of routine                   }
  EXTERNAL;

PROCEDURE trace_end     {Call emitted at end of each routine  }
  $ALIAS 'Pas.TraceEnd'$$
  (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 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 Pas.TraceInit and Pas.TraceClose 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

TRACE_BACK <ON or OFF>

TRACE_BACK only has an effect if the CDS option is ON. This option must be set before the program heading.

TRACE_BACK ON specifies that code should be emitted in every procedure and function to enable the run-time error processor to display the names of the routines in the current routine call chain at the time of a run-time error. OFF specifies that such code should not be emitted.

A trace back through a routine compiled with TRACE_BACK OFF will only be able to display the location of the routine in the code segment. Default: ON

WARN <ON or OFF>

ON specifies that warning messages should be sent to the listing. OFF specifies that warning messages should not be sent to the listing. Warnings are always counted and summarized at the end of compilation. Default: ON

WIDTH <integer>

The <integer> specifies the number of significant characters in a source line (range is 20 to 150). Additional characters on the line are ignored. This option takes effect starting with the line it appears on. Default: 80

WORK <integer>

The <integer> (range 1 to the working set size of the compiler minus one) specifies the number of pages of workspace guaranteed to the memory-resident that the compiler should use. If not specified, a default (50) sufficient for most small programs is used.

The work option is used only to increase performance of the compiler for compilation of large programs, since increasing the work size improves compilation speed, up to a point. After this point, speed decreases, since code segments will be swapped in favor of keeping data in memory. This point depends on the working set size of the compiler (see Pascal Configuration Guide for further information).

This option has no effect on the performance of the version of the compiler which itself runs in CDS mode. Default: 50

Appendix D

User-Callable Pascal/1000

Library Routines

Library Routines

The Pascal library contains several routines that are directly callable from user programs. The external names of these routines are given below may be aliased to user-chosen valid Pascal procedure or function names.

Pas.Parameters Function

The Pas.Parameters (run string parameter) 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. The value -1 is returned if the selected parameter (any succeeding parameters) does not exist. Zero will be returned for any empty parameter, if there are any parameters after the empty parameter. If either 0 or -1 is returned, the user supplied parameter buffer will not be modified.

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, if individual parameters are to be selected, the function expects the buffer to include the entire run string as would be typed at the terminal, including the initial *RU,<program name>*. If the entire run string is retrieved, the contents of the optional buffer can be in any format, including binary data.

- **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.
- **Parameter.** A VAR parameter, typically of type PACKED ARRAY OF CHAR for use with a standard run string, but can be any other type compatible with the contents of the optional buffer set up when the program is scheduled programmatically. The parameter contains upon return either the entire run string or a selected parameter, depending on the value of item 1), Position, above.

If the entire run string is returned, it is returned unmodified; that is, blanks and nulls are not removed around commas nor at the end of the run string.

If a selected parameter is returned, leading and trailing blanks and nulls are stripped. If the selected parameter does not exist, the return value is unchanged. A parameter may not exist because it was not provided in the run string or optional buffer, because the program was scheduled with run-string passage inhibited, no optional buffer was passed, or the RUN_STRING option was set with the value zero.

- **Length.** A value parameter of type one-word integer that passes the maximum number of characters to be returned in Parameter. (The actual number of characters returned is the function value.) If fewer characters than specified are returned, Parameter is blank-filled.

If a program terminates saving resources and wishes to recover any or all parameters from the run string provided when the program is rescheduled, the procedure Pas.GetNewParms must be called immediately following the EXEC call that suspended execution. If this is not done, Pas.Parameters will continue to return parameters from the run string supplied at the initial scheduling, or from that supplied at the last reschedule that did call Pas.GetNewParms.

```
FUNCTION Pas_Parameters
$ALIAS 'Pas.Parameters'$
( position : INT;
  VAR parameter : PAC80;
    length : INT)
: INT;
EXTERNAL;

where: TYPE INT = -32768..32767;
      TYPE PAC80 = 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.

Pas.Sparameters Function

The routine Pas.Sparameters is provided to permit access to the run string parameters using strings.

The position parameter and return value are interpreted as they are for Pas.Parameters. The second parameter is the string into which the argument will be placed. If the function is declared as indicated below, with the second parameter of the anonymous STRING type, it is not necessary to initialize the actual string before passing it to the function. If the second parameter is declared as a specific string type, then the actual string must be initialized before passing it to the function. If the argument is empty or non-existent, the string will not be modified; if it was not initialized before being passed to the function it will still not be initialized.

If a program terminates saving resources and wishes to recover any or all parameters from the run string provided when the program is rescheduled, the procedure Pas.GetNewParms must be called immediately following the EXEC call that suspended execution. If this is not done, Pas.Sparameters will continue to return parameters from the run string supplied at the initial scheduling, or from that supplied at the last reschedule that did call Pas.GetNewParms.

```
FUNCTION Pas_Sparameters
  $ALIAS 'Pas.Sparameter'$*
  (  position: INT;
    VAR parameter: STRING)
  : INT; EXTERNAL;
```

NOTE

The \$HEAPPARMS OFF\$ compiler option must be in effect for the VAR parameter if the \$HEAP 2\$ compiler option is in effect.

Pas.NumericParms Procedure

The Pas.NumericParms routine returns RMPAR parameters. This routine takes one VAR parameter, a five-element array of one-word integers, and returns the values normally obtainable by using the system routine RMPAR at the beginning of the program. Pas.NumericParms is supplied because run-time startup code executed by all Pascal programs makes the use of RMPAR unreliable.

This routine can be used only to pick up the initial parameters. A program that terminates saving resources should not use this routine upon being rescheduled; it should use RMPAR.

```
TYPE
  INT  = -32768..32767;
  PARMs = ARRAY [1..5] OF INT;

PROCEDURE Pas_NumericParms
  $ALIAS 'Pas.NumericParms$'
  (VAR p: PARMs);
EXTERNAL
```

NOTE

The \$HEAPPARMS OFF\$ compiler option must be in effect if Pas.NumericParms is to be called from a HEAP 2 program.

Pas.GetNewParms Procedure

The Pas.GetNewParms routine is called immediately after a terminate saving resources call to acquire the RMPAR and run string parameters that were provided when the program was rescheduled. It saves the parameters in the places that Pas.Parameters, Pas.Sparameters, and Pas.NumericParms expect to find them. It replaces those saved when the program was initially scheduled or which were saved by a previous call to Pas.GetNewParms.

If the RUN_STRING option is set to zero, Pas.GetNewParms will not get the new run string. See RUN_STRING in Appendix C for more information.

Pas.GetNewParms should be called only once immediately after the terminate saving resources call. The effect of a call at any other time is undefined and may destroy the existing saved parameters.

NOTE

The association of names specified on the run string with files specified in the program heading will be done with the new run string after a call to Pas.GetNewParms.

```

TYPE
  INIT = -32768..32767;

PROCEDURE stopsave
  $ALIAS 'EXEC'$;
  (icode: INT;
  iprog: INT;
  itype: INT); EXTERNAL;

PROCEDURE Pas_GetNewParms;
  $ALIAS 'Pas.GetNewParms'$;
  EXTERNAL;

.
.
.

stopsave (6, 0, 1); {Terminates self saving resources}
Pas_GetNewParms; {Get the new RMPAR/run string parameters}

```

Pas.DcbAddress1 Procedure

The Pas.DcbAddress1 (get address of DCB) procedure, to be called from \$HEAP 1\$ programs only, returns a pointer to the DCB located in a Pascal file. The returned pointer is the two-word address of the file DCB. It can be dereferenced and passed to FMP routines that expect a DCB parameter by VAR if the \$HEAPPARMS OFF\$ option is in effect for the DCB parameter to the FMP routine.

```

TYPE
  INT      = -32768..32767;
  DCB      = ARRAY [1..144] OF INT;
  DCBPTR   = ^DCB;
  FILENAME = PACKED ARRAY [1..6] OF CHAR;
  FILETYPE = FILE OF INT;

PROCEDURE real_dcb_address
  $ALIAS 'Pas.Dcbaddress1'
  (VAR dcbp: DCBPTR;
   VAR file: FILETYPE);
EXTERNAL;

```

To use Pas.Dcbaddress1 with different types of files, multiple EXTERNAL declarations with different Pascal names (but the same ALIAS) can be used.

To pass the 'DCB' to the FMP routine, the pointer returned by Pas.Dcbaddress1 must be dereferenced. In the following example:

```

VAR
  ints: FILETYPE;
  dcgp: DCBPTR;
  errn: INT;

PROCEDURE fmp_close
  $ALIAS 'CLOSE'
  (VAR idcb: DCB;
   VAR ierr: INT);
EXTERNAL;
:
real_dcb_address (dcgp, ints);
fmp_close (dcgp^,errn);

```

the file will be 'closed' as far as FMP is concerned, but not Pascal. Chapter 8, FMP vs. Pascal/1000 I/O, should be consulted for things that must be taken into consideration when using FMP calls to access Pascal files.

Pas.DcbAddress2 Procedure

The Pas.DcbAddress2 (get address of DCB) procedure is similiar to Pas.Dcbaddress1, except that it is to be called from \$HEAP 2\$ programs only. It returns a pointer to the DCB located in a Pascal file. The returned pointer is the two-word address of the file DCB, it can be dereferenced and passed to FMP routines which expect a DCB parameter by VAR, if the \$HEAPPARMS OFF\$ option is in effect for the DCB parameter to the FMP routine.

```

TYPE
  INT      = -32768..32767;
  DCB      = ARRAY [1..144] OF INT;
  DCBPTR   = ^DCB;
  FILENAME = PACKED ARRAY [1..6] OF CHAR;
  FILETYPE = FILE OF INT;

PROCEDURE real_dcb_address
  $ALIAS 'Pas.DcbAddress2';
  (VAR dcbp: DCBPTR;
  VAR fyle: FILETYPE);
  EXTERNAL;

```

NOTE

The VAR parameters to Pas.DCBAddress2 must be declared with \$HEAPPARMS ON\$.

Pas.FileNamr Function

The Pas.FileNamr function returns the external name of a currently open Pascal file.

```
TYPE
  FILENAME = PACKED ARRAY [1..64] OF CHAR;

  FUNCTION getfilename
  $ALIAS 'Pas.FileNamr'
  (VAR x: FILETYPE)
  : FILENAME; EXTERNAL;
```

If the program is loaded with the FMGR access only library, only the first 18 characters of the returned result will be valid, and the remaining 46 characters will be unchanged and they will not be blank padded. The result will be in the form:

FILENAME:::-LUN:FILETYPE

If the program is loaded with the file system access library, all 64 characters will be returned, blank padded as necessary. The result will be in the form returned by the routine FMPFileName.

To use Pas.FileNamr with different types of files, multiple EXTERNAL declarations with different Pascal names, but the same ALIAS, can be used.

NOTE

The \$HEAPPARMS OFF\$ compiler option must be in effect for the VAR parameter if the \$HEAP 2\$ option is in effect.

Pas.InitMemInfo1 Procedure

The Pas.InitMemInfo1 procedure initializes the \$HEAP 1\$ heap and stack area. It effectively makes the heap and stack empty. It can be called by a \$HEAP 0\$ main program to enable proper function of routines which were compiled using \$HEAP 1\$ such as modules, SUBPROGRAMs, and SEGMENTs.

```
PROCEDURE InitMem1;
  $ALIAS,'Pas.InitMemInfo1';
  EXTERNAL;
```

The following conditions should apply when the procedure is called:

- a. The call should be from the main program. Clearing the stack from a procedure or function has unpredictable results.
- b. The call should be before any call to a \$HEAP 1\$ routine.
- c. The call should be before any objects are allocated in the heap.

In general, this call should be the first action taken by the main program. See also the Pas.InitMemInfo2 procedure discussed later in this Appendix.

Pas.GetMemInfo1 Procedure

The Pas.GetMemInfo1 (get heap 1 and stack information) 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 'Pas.GetMemInfo1'
  (VAR heap_info1 : INFO_REC1);
EXTERNAL;

```

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.

The above description applies to programs running in a non-CDS environment. For programs running in a CDS environment some of the fields of the returned record have different meanings.

tos — Q register at the time of this call to Pas.GetMemInfo1
toh — Z register at the time of this call + 264
init_tos — Q register at the time the program started
init_toh — Z register at the time the program started + 264
high_tos — Highest value of Q at a call to Pas.GetMemInfo1
high_toh — Lowest value of Z + 264 so far during execution

Pas.SetMemInfo1 Procedure

The Pas.SetMemInfo1 (set heap 1 and stack information) 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
  $ALIAS 'Pas.SetMemInfo1'$
  (heap_info1 : INFO_REC1);
EXTERNAL;
```

For programs running in a CDS environment, the Q register will not be changed by this call. The Z register will be set 264 words below the toh value.

Pas.InitialHeap1 Procedure

The Pas.InitialHeap1 (initialize heap 1 and stack information) 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 'Pas.InitialHeap1';
  EXTERNAL;
```

Pas.InitMemInfo2 Procedure

The Pas.InitMemInfo2 procedure initializes the \$HEAP 2\$ heap and stack area. It effectively makes the heap and stack empty. It can be called by a \$HEAP 0\$ or \$HEAP 1\$ main program to enable proper function of routines which were compiled using \$HEAP 2\$ such as modules, SUBPROGRAMs, and SEGMENTs. This procedure also calls Pas.InitMemInfo1. Therefore, both routines need not be called.

```
PROCEDURE InitMem2;
  $ALIAS,'Pas.InitMemInfo2';
  EXTERNAL;
```

The following conditions should apply when the procedure is called:

- a. The call should be from the main program. Clearing the stack from a procedure or function has unpredictable results.
- b. The call should be before any call to a \$HEAP 2\$ routine.
- c. The call should be before any objects are allocated in the heap.

In general, this call should be the first action taken by the main program. See the Pas.InitMemInfo1 procedure discussed earlier in this appendix.

Pas.GetMemInfo2 Procedure

The Pas.GetMemInfo2 (get heap 2 and stack information) procedure is similar to Pas.GetMemInfo1, 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,                      {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 : ADDR2;       {current mark list block}
  END;

PROCEDURE get_heap_2_stack_info
  $ALIAS 'Pas.GetMemInfo2'
  (VAR heap_info2 : INFO_REC2);
EXTERNAL;

```

The above description applies to programs running in a non-CDS environment, for programs running in a CDS environment some of the fields of the returned record have different meanings.

tos — amount of relocated EMA area if the program was loaded by LINK, zero otherwise
 init_tos — same as tos
 high_tos — same as tos

Q and Z register information are not available with this routine.

NOTE

The VAR parameter to Pas.GetMemInfo2 must be declared with \$HEAPPARMS ON\$.

Pas.SetMemInfo2 Procedure

The Pas.SetMemInfo2 (set heap 2 and stack information) procedure is similar to Pas.SetMemInfo1, 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 in the Pas.GetMemInfo2 Procedure.

```
PROCEDURE set_heap_2_stack_info
  $ALIAS 'Pas.SetMemInfo2'
  (heap_info2 : INFO_REC2);
  EXTERNAL;
```

For programs loaded with LINK, setting tos may effect protection of relocated EMA areas in either the CDS or non-CDS environment.

For programs running in a CDS environment, the Q register and Z register will not be changed by this call.

Pas.InitialHeap2 Procedure

The Pas.InitialHeap2 (initialize heap 2 and stack information) procedure is similar to Pas.InitialHeap1 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_info;
  $ALIAS 'Pas.InitialHeap2';
  EXTERNAL;
```

Pas.Coalescel and Pas.Coalesce2 Procedures

Dynamic variables that are disposed are maintained in a free space list by the heap management routines. Normally these free spaces are not coalesced into large spaces where possible. Should *new* fail to find a large enough free space to satisfy a request, and should there be insufficient space at the top of the heap, the appropriate free space coalescing routine will be called. Should it recover sufficient space, the new will succeed. Should it fail, a run time error will occur.

Pas.Coalescel
Pas.Coalesce2

Free List Coalescers

As the coalescing process takes time proportional to the square of the number of objects in the free list, this automatic process may be detrimental to program performance. The coalescing routines Pas.Coalescel and Pas.Coalesce2 (for HEAP 1 and HEAP 2, respectively) are user callable to provide preventive free space coalescing at a time of the program's choosing.

```
PROCEDURE PasCoalescel;  
  $ALIAS 'Pas.Coalescel'$  
  EXTERNAL;  
  
PROCEDURE PasCoalesce2;  
  $ALIAS 'Pas.Coalesce2'$  
  EXTERNAL;
```

Pas.TimeString Procedure

The Pas.TimeString procedure returns a PACKED ARRAY [1..26] OF CHAR containing an ASCII representation of the current date and time. A sample time string:

Sun Jan 6, 1980 3:01 pm

```
TYPE  
  PAC26 = PACKED ARRAY [1..26] OF CHAR;  
  
PROCEDURE get_time  
  $ALIAS 'PasTimeString'$  
  (VAR time_string : PAC26);  
  EXTERNAL;
```

NOTE

The \$HEAPPARMS OFF\$ compiler option must be in effect if PasTimeString is to be called from a HEAP 2 program.

Pas.SegmentLoad Procedure

The Pas.SegmentLoad (segment load) procedure loads a Pascal compilation unit that was compiled with the \$SEGMENT\$ option. It requires a parameter of type PACKED ARRAY [1..5] OF CHAR in which the program name of the compilation unit is passed.

In the CDS version of the library, this routine does nothing. It is present to allow programs which used it in non-CDS mode to be compiled and loaded in CDS mode without having to remove Pas.SegmentLoad calls.

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

PROCEDURE seg_load
  $ALIAS 'Pas.SegmentLoad$'
  (name: SEG_NAME);
EXTERNAL;
```

Pas.TraceBack Procedure

The routine Pas.TraceBack is provided to permit display of the current procedure/function call chain in CDS programs.

```
PROCEDURE Pas_TraceBack
  $ALIAS 'Pas.TraceBack'$
  (lu: INT); EXTERNAL;
```

The current procedure/function call chain is displayed on the lu (range 0 to 255) passed as the parameter. The declaration level number, the name, and the octal offset from the beginning of the routine to the call to the next routine, are displayed for routines compiled with the TRACE_BACK option in effect. Routines which do not contain trace back information will have the segment number and octal offset of the call to the next routine displayed.

FTN7X routines containing trace back information will have the name and octal offset of the call to the next routine displayed. A level number may or may not be displayed, if it is, it is a meaningless value.

Pas.TraceBack can be called from non-CDS programs, for source compatibility, but no trace back will be produced.

Pas.StringData1 and Pas.StringData2 Functions

The functions Pas.StringData1 and Pas.StringData2 are provided to give the user access to the part of a string which contains the actual characters. The functions return a pointer which effectively points to a PACKED ARRAY [1..strmax(s)] OF CHAR. Pas.StringData1 is used with HEAP 1 programs and Pas.StringData2 is used with HEAP 2 programs. The parameter must be passed by VAR and must be the anonymous string type STRING.

Example

```

PROGRAM example;

CONST
  string_size = 80;

TYPE
  INT      = -32768..32767;
  STR80   = STRING [string_size];
  PAC80   = PACKED ARRAY [1..string_size] OF CHAR;
  PACPTR  = ^PAC80;

VAR
  s: STR80;

FUNCTION Pas_StringData1
$ALIAS 'Pas.StringData1'$*
(VAR s: STRING)
: PACPTR; EXTERNAL;

PROCEDURE syswrite
$ALIAS 'EXEC'$*
(icode: INT;
  icnwd: INT;
  ibfr: PAC80;
  ilen: INT); EXTERNAL;

BEGIN
  s := 'A message for the user...';
  syswrite(2, 1, Pas_StringData1(s)^, -strlen(s));
END.

```



NOTE

The VAR parameter to Pas.StringData2 must be declared with \$HEAPPARMS ON\$.

Pas.StrEndCheck

The routine Pas.StrEndCheck is provided to enable programs to disable certain string under/overflow checks at runtime.

```
PROCEDURE Pas_StrEndCheck
  $ALIAS 'Pas.StrEndCheck';
  (check: BOOLEAN); EXTERNAL;
```

When a program starts there is effectively a call to this routine with *check* set to true. This corresponds to the HP Standard for Pascal which requires that all string under/overflow errors be fatal runtime errors. Setting *check* to false will effectively cause truncation of characters pushed off the end in the case of overflow and completion of the operation to the extent possible on underflow. This is a Pascal/1000 extension, it is not part of HP Standard Pascal. Programs utilizing this feature are not portable among HP systems.

These general cases are disabled:

<i>:=</i>	— overflow:	result will be truncated
value parameter	— overflow:	result will be truncated
<i>+</i>	— overflow:	result will be truncated
Strappend	— overflow:	result will be truncated
Strinsert	— overflow:	result will be truncated
Strmove	— overflow:	result will be truncated
Strwrite	— overflow:	result will be truncated
Strdelete	— underflow:	deletion will occur to end of string

These specific cases are NOT disabled:

Detectable at compile time:		Error
:=	— overflow	149
value parameter	— overflow	149
Strdelete	— start_pos <= 0 OR > strmax (s) num_to_delete < 0 underflow	302 306 306
Strinsert	— start_pos <= 0 OR > strmax (s) overflow	302 306
Strmove	— source_pos <= 0 OR > strmax (source) dest_pos <= 0 OR > strmax (dest) overflow	302 302 306
Strwrite	— start_pos <= 0 OR > strmax (s)	302

Detectable at run-time:		
Strdelete	— start_pos <= 0 OR > strlen (s) num_to_delete < 0	8 6
Strinsert	— start_pos <= 0 OR > strlen (s2) + 1	8
Strmove	— source_pos <= 0 OR > strlen (source) dest_pos <= 0 OR > strlen (dest) + 1	8 8
Strwrite	— start_pos <= 0 OR > strlen (s) + 1	8

All other string procedures, functions, and operations cannot be disabled and will always cause runtime errors.

WARNING

Expressions involving string expressions of unknown maximum length have assumed maximum lengths of 256 characters and truncation to that length will occur if 'check' is false.

System Common Access Routines

The following functions provide access to realtime and background system common under RTE-6/VM and to blank and labeled system common under RTE-XL and RTE-A:

Pas.RealTimeSize	RTE-6/VM
Pas.RealTimeCom1	RTE-6/VM
Pas.RealTimeCom2	RTE-6/VM
Pas.BackGrndSize	RTE-6/VM
Pas.BackGrndCom1	RTE-6/VM
Pas.BackGrndCom2	RTE-6/VM
Pas.BlankSize	RTE-XL and RTE-A.1
Pas.BlankCom1	RTE-XL and RTE-A.1
Pas.LabelSize	RTE-XL and RTE-A.1
Pas.LabelCom1	RTE-XL and RTE-A.1

Pascal cannot check that the RECORD, ARRAY, or other object being used as the template for system common layout does in fact fit into the common area. This responsibility is left to the application program. If the size function for a particular type of common returns zero, the pointer functions for that common area will return nil (and the common area does not exist).

Programs using these routines must be loaded with OP,SC under RTE-6/VM and with OP,SC or OP,LC (as appropriate) under RTE-A. No loader error will occur if the system common loader directive is omitted, but the program will not be accessing system common.

```

TYPE
  INT = -32768..32767;
  COM = RECORD
    {Record which describes common layout}
    field1: INT;
    field2: CHAR;
  END;
  COMPTR = ^COM;

```

{These functions return the size of the respective common area}

```
FUNCTION Pas_RealTimeSize      {requires OP,SC}
  $ALIAS 'Pas.RealTimeSize'$ 
  : INT; EXTERNAL;

FUNCTION Pas_BackGrndSize     {requires OP,SC}
  $ALIAS 'Pas.BackGrndSize'$ 
  : INT; EXTERNAL;

FUNCTION Pas_BlkSize          {requires OP,SC}
  $ALIAS 'Pas.BlkSize'$ 
  : INT; EXTERNAL;

FUNCTION Pas_LabelSize        {requires OP,LC}
  $ALIAS 'Pas.LabelSize'$ 
  : INT; EXTERNAL;
```

{These functions return a pointer to the system common area}
{Functions with the suffix 2 are used in HEAP 2 programs }

```
FUNCTION Pas_RealTimeCom1    {requires OP,SC}
  $ALIAS 'Pas.RealTimeCom1'$ 
  : COMPTR; EXTERNAL;

FUNCTION Pas_BackGrndCom1    {requires OP,SC}
  $ALIAS 'Pas.BackGrndCom1'$ 
  : COMPTR; EXTERNAL;

FUNCTION Pas_BlkCom1         {requires OP,SC}
  $ALIAS 'Pas.BlkCom1'$ 
  : COMPTR; EXTERNAL;

FUNCTION Pas_LabelCom1       {requires OP,LC}
  $ALIAS 'Pas.LabelCom1'$ 
  : COMPTR; EXTERNAL;
```

Shareable EMA Access

An example application which shows the use of both system common and shareable EMA follows the information on the shareable EMA access routines.

For RTE-6/VM: Pas.SharedSize
 Pas.SetShared

For RTE-A: Pas.A1SharedSize
 Pas.A1SetShared

These functions provide access to shareable EMA under RTE-6/VM and RTE-A. Programs using shareable EMA must be loaded with the SH option.

Sharing of items in the heaps is totally under program control. To share a data item, a pointer to it is passed from one program to another that has access to the same shareable EMA area.

The overlapping of stack areas is not recommended, because stack growth is not directly under user control (unless a program does no recursion). Exact overlapping of heaps would in fact work, one program specifying (0,10000) and another specifying (5000,5000), but if one program deallocated variables without the other's knowledge, the results, would be unpredictable. Additionally the first program could grow the heap into the second programs stack without the second program being aware that it has happened.

For RTE-6/VM: FUNCTION Pas_SharedSize
 \$ALIAS 'Pas.SharedSize'\$
 : INTEGER; EXTERNAL;

For RTE-A: FUNCTION Pas_SharedSize
 \$ALIAS 'Pas.A1SharedSize'\$
 : INTEGER; EXTERNAL;

Pas.SharedSize and Pas.A1SharedSize return the size of the shareable EMA partition the program has access to (as a 32-bit integer). A zero return indicates that the program does not have access to shareable EMA.

For RTE-6/VM: FUNCTION Pas_SetShared
 \$ALIAS 'Pas.SetShared'\$
 (start_address: INTEGER;
 heap_stack_size: INTEGER)
 : BOOLEAN; EXTERNAL;

For RTE-A: FUNCTION Pas_SetShared
 \$ALIAS 'Pas.A1SetShared'\$
 (start_address: INTEGER;
 heap_stack_size: INTEGER)
 : BOOLEAN; EXTERNAL;

Pas.SetShared sets the programs heap/stack area to start at the specified address and to extend for the specified number of words. A true return indicates that Pas.TopOfStack2/Pas.TopOfHeap2 have been set to the start and end of the specified area. A false return indicates that the program does not have access to shared EMA or the specified area will not fit within the shareable EMA area (and Pas.TopOfStack2/Pas.TopOfHeap2 were not modified).

To ensure that the heap/stack do not contain information that might be destroyed by moving the heap/stack, the appropriate SetShared routine should be called before any call to new, dispose, mark, or release (CDS or non-CDS environment) and before invocation of any procedure or function declared with the RECURSIVE option ON (non-CDS environment).

For programs run in the CDS environment, the stack is not kept in EMA, and so is not a concern in the partitioning of EMA if all the sharing programs are running in the CDS environment.

Programs which relocate data into EMA have additional constraints. If the program is loaded with LINK, Pas.GetMemInfo2 should be called to determine a safe value to be passed to the appropriate SetShared routine as the start_address. The value passed as the start_address should be the value of init_tos (or a greater value) in order to maintain the protection of relocated EMA areas. If the program is loaded with something other than LINK, it is the program's responsibility to ensure that the start_address value is above any relocated EMA area (as Pas.GetMemInfo2 will return zero for init_tos, whether or not there is relocated EMA area, if the program was not loaded with LINK).

An example application that shows the use of both system common and shareable EMA follows. The example contains complete listings of two Pascal programs, SENDR and RECVR. They implement a TEXT file copy utility that can be used across sessions.

SENDER copies its input into a shareable EMA partition, and RECVR takes the data from the EMA partition and writes it to its output. A record in system common is used to synchronize the process.

Obviously both must be loaded with access to the same shared EMA partition (if not, the results are unpredictable).

To ensure that system common is properly initialized, SENDER must be run before RECVR.

Both programs check that sufficient system common and shareable EMA exists before committing themselves to action.

In order to run the program under RTE-A, the following changes must be made:

1. Change Pas.BackGrndSize to Pas.BlankSize
2. Change Pas.BackGrndCom2 to Pas.BlankCom2
3. Change Pas.SharedSize to Pas.A1SharedSize
4. Change Pas.SetShared to Pas.A1SetShared

User-Callable Pascal/1000 Library Routines

Pascal/1000
Ver. 1/2135

Thu Aug 6, 1981 7:32 pm
&SENDER::JS Page 1

```
1 00000 $PASCAL ',4,90 Send', HEAP 2, RECURSIVE OFF$  
2 00000  
3 00000 PROGRAM sendr (input);  
4 00337  
5 00337 $INCLUDE '[SHTYP'$  
1 00337  
1 00337 TYPE  
3 00337     INT = -32768..32767;  
4 00337  
5 00337 {-----}  
6 00337  
7 00337 CONST  
8 00337     buffer_size = 10000;  
9 00337  
10 00337 TYPE  
11 00337     BUFFER_INDEX = 0..buffer_size - 1;  
12 00337  
13 00337     DATA_STRUC = RECORD  
14 00337         buffer: PACKED ARRAY [BUFFER_INDEX] OF CHAR;  
15 00337         END;  
16 00337  
17 00337     DATA_PTR = ^DATA_STRUC;  
18 00337  
19 00337 CONST  
20 00337     data_size = 2 + (buffer_size DIV 2);  
21 00337  
22 00337 VAR  
23 00337     data_area: DATA_PTR;  
24 00341     data_avail: INTEGER;  
25 00343  
26 00343 {-----}  
27 00343  
28 00343 TYPE  
29 00343     COM_STRUC = RECORD  
30 00343         sender: BOOLEAN;  
31 00343         receiver: BOOLEAN;  
32 00343         reader: BUFFER_INDEX;  
33 00343         writer: BUFFER_INDEX;  
34 00343         data: DATA_PTR;  
35 00343         END;  
36 00343  
37 00343     COMM_PTR = ^COMM_STRUC;  
38 00343  
39 00343 CONST  
40 00343     comm_size = 1 + 1 + 1 + 1 + 2;  
41 00343  
42 00343 VAR  
43 00343     comm_area: COMM_PTR;
```

Pascal/1000
Ver. 1/2135Thu Aug 6, 1981 7:32 pm
&SENR::JS Page 2

```
44 00345     comm_avail: INT;
45 00346
46 00346 {-----}
47 00346
48 00346 $PAGE$
49 00346 CONST
50 00346     heap_stack = 1000;
51 00346
52 00346     send_start = 0;
53 00346     send_size = data_size + heap_stack;
54 00346     receive_start = send_size;
55 00346     receive_size = heap_stack;
56 00346
57 00346     eoln_char = chr (255);
58 00346
59 00346 VAR
60 00346     x: CHAR;
61 00347     outf: TEXT;
6 00706
7 00706 $INCLUDE '[SHEXT$'
1 00706
2 00706 FUNCTION back_ground_size
3 00000     $ALIAS 'Pas.BackGrndSize'$
4 00000     : INT; EXTERNAL;
5 00000
6 00000
7 00000
8 00000 FUNCTION back_ground_common
9 00000     $ALIAS 'Pas.BackGrndCom2'$
10 00000     : COMM_PTR; EXTERNAL;
11 00000
12 00000
13 00000
14 00000 FUNCTION shared_size
15 00000     $ALIAS 'Pas.SharedSize'$
16 00000     : INTEGER; EXTERNAL;
17 00000
18 00000
19 00000
20 00000 FUNCTION set_shared
21 00000     $ALIAS 'Pas.SetShared'$
22 00000     (f: INTEGER;
23 00000     n: INTEGER)
24 00000     : BOOLEAN; EXTERNAL;
25 00000
26 00000 $PAGE$
```

User-Callable Pascal/1000 Library Routines

Pascal/1000
Ver. 1/2135

Thu Aug 6, 1981 7:32 pm
&SENR::JS Page 3

```
27 00000 PROCEDURE sys_wait
28 00000     $ALIAS 'EXEC'$
29 00000     (icode: INT;
30 00000         iname: INT;
31 00000         iresl: INT;
32 00000         imult: INT;
33 00000         iofst: INT);
34 00000     EXTERNAL;
35 00000
36 00000
37 00000
38 00000 PROCEDURE wait
39 00000     (seconds: INT);
40 00000
41 00000 CONST
42 00001     schedule      = 12;
43 00001     self          = 0;
44 00001     units_r_seconds = 2;
45 00001     run_once      = 0;
46 00001
47 00001 BEGIN
48 00005     sys_wait (schedule, self, units_r_seconds, run_once, -seconds);
49 00017 END;
50 00034
51 00034
52 00034
53 00034 FUNCTION space_available
54 00000     : BOOLEAN;
55 00000
56 00000 BEGIN
57 00034     rewrite (outf, '1');
58 00047     comm_avail    := back_ground_size;
59 00052     comm_arem    := back_ground_common;
60 00056     data_avail    := shared_size;
61 00062     space_available := true;
62 00064
63 00064     IF comm_avail < comm_size THEN BEGIN
64 00074         writeln (outf, 'Insufficient system common: ',
65 00074             comm_size:0, ' required, ',
66 00074             comm_avail:0, ' available.');
67 00135         space_available := false;
68 00137     END;
69 00137
70 00137     IF data_avail < send_size + receive_size THEN BEGIN
71 00146         writeln (outf, 'Insufficient shared EMA: ',
72 00146             send_size + receive_size:0, ' required, ',
73 00146             data_avail:0, ' available.');
74 00207         space_available := false;
75 00211     END;
76 00211 END;
77 00323 $PAGE$
```

Pascal/1000
Ver. 1/2135Thu Aug 6, 1981 7:32 pm
&SENDER::JS Page 4

```
8 00323 FUNCTION next
9 00000      $DIRECT$
10 00000      : BUFFER_INDEX;
11 00000
12 00000 BEGIN
13 00323      next := (comm_area^.writer + 1) MOD buffer_size;
14 00346 END;
15 00361
16 00361
17 00361
18 00361 FUNCTION busy
19 00000      $DIRECT$
20 00000      : BOOLEAN;
21 00000
22 00000 BEGIN
23 00361      busy := (comm_area^.reader = comm_area^.writer);
24 00407 END;
25 00425
26 00425
27 00425
28 00425 PROCEDURE put
29 00000      (c: CHAR);
30 00000
31 00000 VAR
32 00001      temp: BUFFER_INDEX;
33 00002
34 00002 BEGIN
35 00425      WHILE busy DO BEGIN
36 00430          wait (1);
37 00433      END;
38 00434
39 00434      WITH data_area^, comm_area^ DO BEGIN
40 00450          buffer [writer] := c;
41 00475          writer := next;
42 00507      END;
43 00507 END;
44 00542
45 00542 $PAGE$
46 00542 BEGIN {sendr}
47 00634      IF space_available THEN BEGIN
48 00640          {initialize the communication area. so we must}
        {get to here before the receiver starts up.    }
```

Pascal/1000
Ver. 1/2135Thu Aug 6, 1981 7:32 pm
&SENR::JS Page 5

```

50 00640      WITH comm_area^ DO BEGIN
51 00640          sender := false;
52 00646          receiver := false;
53 00652          reader := buffer_size - 1;
54 00657          writer := 0;
55 00664      END;
56 00671
57 00671      {set up our heap stack and create the data buffer}
58 00671
59 00671      IF set_shared (send_start, send_size) THEN BEGIN
60 00671
61 00677          new (data_area);           {put a pointer to the data }
62 00677          WITH comm_area^ DO BEGIN {buffer in the communication}
63 00704              data := data_area; {area and then tell the }
64 00712              sender := true;    {receiver that it is there. }
65 00722          END;
66 00726
67 00726      {copy the file to the data buffer (and ultimately the}
68 00726      {receiver). we may finish before the receiver even }
69 00726      {starts if the buffer is bigger than the file.       }
70 00726
71 00726
72 00726      WHILE NOT eof (input) DO BEGIN
73 00732          WHILE NOT eoln (input) DO BEGIN
74 00736              read (x);
75 00745              put (x);
76 00750          END;
77 00751          readln;
78 00753              put (eoln_char);
79 00756          END;
80 00757
81 00757      WITH comm_area^ DO BEGIN
82 00765          {wait for receiver to exist (as we could have}
83 00765          {dumped the whole file into the buffer before}
84 00765          {it started up.                                }
85 00765
86 00765      WHILE NOT receiver DO BEGIN
87 00773          wait (1);
88 00776          END;
89 00777
90 00777      {now say that we are done, then wait for   }
91 00777      {the receiver to finish so as not to     }
92 00777      {confuse the users as to what is going on.}
93 00777
94 00777          sender := false;
95 01003          WHILE receiver DO BEGIN
96 01011              wait (1);
97 01014          END;
98 01015      END

```

User-Callable Pascal/1000 Library Routines

Pascal/1000
Ver. 1/2135

Thu Aug 6, 1981 7:32 pm
&SENDR::JS Page 6

```
99 01015      END
100 01015     ELSE BEGIN
101 01016       writeln (outf, 'Heap/Stack setup failure.');
102 01027     END;
103 01027   END;
104 01027 END.
```

```
0 Errors detected.
242 Source lines read.
1048 Words of program generated.
```

User-Callable Pascal/1000 Library Routines

Pascal1/1000
Ver. 1/2145

Fri Jul 24, 1981 11:37 am
&RECVR::JS Page 1

```
1 00000 $PASCAL ',4,90 Receive', HEAP 2, RECURSIVE OFF$  
2 00000  
3 00000 PROGRAM recvr (output);  
4 00337  
5 00337 $LIST OFF, INCLUDE '[SHTYP', LIST$ {see SENDR for listing}  
66 00706  
67 00706 $LIST OFF, INCLUDE '[SHEXT', LIST, PAGE$ {see SENDR for listing}
```

Pascal/1000
Ver. 1/2145Fri Jul 24, 1981 11:37 am
&RECVR::JS Page 2

```
145 00323 FUNCTION next
146 00000      $DIRECT$
147 00000      : BUFFER_INDEX;
148 00000
149 00000 BEGIN
150 00323      next := (comm_area^.reader + 1) MOD buffer_size;
151 00346 END;
152 00361
153 00361
154 00361
155 00361 FUNCTION busy
156 00000      $DIRECT$
157 00000      : BOOLEAN;
158 00000
159 00000 BEGIN
160 00361      busy := (next = comm_area^.writer);
161 00401 END;
162 00415
163 00415
164 00415
165 00415 FUNCTION get
166 00000      (VAR c: CHAR)
167 00000      : CHAR;
168 00000
169 00000 VAR
170 00001      temp: BUFFER_INDEX;
171 00002
172 00002 BEGIN
173 00415      WHILE busy DO BEGIN
174 00420          wait (1);
175 00423      END;
176 00424
177 00424      WITH data_area^, comm_area^ DO BEGIN
178 00440          temp := next;
179 00445
180 00445          c := buffer [temp];
181 00470
182 00470          get := c;
183 00477
184 00477          reader := temp;
185 00511      END;
186 00511 END;
187 00553
188 00553 $PAGE$
```

User-Callable Pascal/1000 Library Routines

Pascal/1000
Ver. 1/2145

Fri Jul 24, 1981 11:37 am
&RECVR::JS
Page 3

```
189 00553 BEGIN {recvr}
190 00645     IF space_available THEN BEGIN
191 00651         WITH comm_area^ DO BEGIN
192 00657             {wait for the sender to exist    }
193 00657             {then pick up the buffer pointer}
194 00657
195 00657             WHILE NOT sender DO BEGIN
196 00664                 wait (1);
197 00667             END;
198 00670
199 00670             data_area := data;
200 00677
201 00677             {set up our stack area, which we actually never use}
202 00677
203 00677             IF set_shared (receive_start, receive_size) THEN BEGIN
204 00705                 {admit our existence, then copy data}
205 00705                 {until it and the sender are gone,   }
206 00705                 {then admit we are gone and quit   }
207 00705
208 00705             receiver := true;
209 00712
210 00712             REPEAT
211 00712                 WHILE get (x) <> eoln_char DO BEGIN
212 00717                     write (x);
213 00724                     END;
214 00725                     writeln;
215 00727                     UNTIL busy AND NOT sender;
216 00737
217 00737             receiver := false;
218 00744         END
219 00744         ELSE BEGIN
220 00745             writeln (outf, 'Heap/Stack setup failure.');
221 00756         END;
222 00756     END;
223 00756     END;
224 00756 END.
```

0 Errors detected.
224 Source lines read.
999 Words of program generated.

Index

A

abs function, 7-20
access rights
 dangling pointers, 4-52
 explicit import, 4-49
 implicit import, 4-50
actual parameters, 4-28
addition operator, 5-35
ALIAS compiler option, 3-12, C-3
allocation
 elements of packed structures, 8-4
 scalar variables, 8-1
 storage, 8-1
 structured variables, 8-1
AND operator, 5-36
ANSI compiler option, C-4
append procedure, 6-15
arctan function, 7-20
arithmetic functions
 abs, 7-20
 arctan, 7-20
 cos, 7-20
 exp, 7-21
 ln, 7-21
 odd, 7-21
 predefined, 7-19
 round, 7-22
 sin, 7-22
 sqr, 7-22
 sqrt, 7-23
 trunc, 7-23
arithmetic operators, 5-34
array
 constant, 4-6
 multiply-dimensioned, 4-18
 packed, 4-18
 subscripts, 5-30
ASMB compiler option, C-4
assignment compatibility, 5-45
assignment statement, 4-30, 5-5
associating logical and physical files, 6-8
AUTOPAGE compiler option, C-4

B

base page, 8-13, 8-19
base type, 4-16, 4-22
basic symbols, 2-2

BASIC_STRING compiler option, C-5
binary function, 7-25
block
 main program, 3-5
 module, 3-6
Boolean, 4-11
Boolean operators, 4-11, 5-36
buffer variable, 5-31
BUFFERS compiler option, C-5

C

calling FMP routines from Pascal, 9-31
calling Non-Pascal routines from Pascal, 9-29
calling Pascal routines from Non-Pascal routines, 9-30
CASE statement, 5-12, 8-73
catching errors, A-10
CDS compiler option, C-6
CDS heap management, 8-42
CDS memory configuration, 8-17
CDS stack management, 8-24
CHAR, 4-11
chr function, 7-26
close procedure, 8-57
closing files, 8-57
CODE compiler option, C-6
code partition, 8-19
CODE_CONSTANTS compiler option, C-6
CODE_INFO compiler option, C-7
CODE_OFFSETS compiler option, C-7
comments, 2-8
common subexpressions, 8-70
compatibility, type, 5-44
compilation units, 3-1
compile-time warnings, A-12
compile-time errors, A-12
compiler options
 ALIAS, 3-12, C-3
 ANSI, C-4
 ASMB, C-4
 AUTOPAGE, C-4
 BASIC_STRING, C-5
 BUFFERS, C-5
 CDS, C-6
 CODE, C-6
 CODE_OFFSETS, C-7
 CODE_CONSTANTS, C-6
 CODE_INFO, C-7
 DEBUG, C-7
 DIRECT, C-8

EMA, C-9
EMA_VAR, C-10
ERROREXIT, C-10
FAST_REAL_OUT, C-11
FIXED_STRING, C-11
HEAP, C-12
HEAP n, 8-15, 8-21
HEAPPARMS, C-13
HEAP_DISPOSE, C-12
IDSIZE, 2-4, C-16
IMAGE, C-16
INCLUDE, 3-8, C-16
INCLUDE_DEPTH, C-17
KEEPASMB, C-17
LINES, C-17
LINESIZE, C-18
LIST, C-18
LIST_CODE, C-18
MIX, C-18
NOABORT, C-19
PAGE, C-20
PARTIAL_EVAL, C-20
PASCAL, C-21
PRIVATE_TYPES, C-22
RANGE, 8-71, C-22
RECURSIVE, 4-34, C-23
RESULTS, C-23
RUN_STRING, C-24
SEARCH, C-24
SEGMENT, 3-10, C-25
SEGMENTED, C-26
SKIP_TEXT, C-26
SMALL_TEMPS, C-27
STANDARD_LEVEL, C-27
STATS, C-27
SUBPROGRAM, C-30
SUBTITLE, C-30
TABLES, C-30
TITLE, C-34
TRACE, C-35
TRACE_BACK, C-36
WARN, C-36
WIDTH, C-36
WORK, C-37
compiler workspace, 9-14
compiling a program, 9-3
compound statements, 5-9
constants
 array, 4-6
 definitions, 4-4
 expressions, 5-42
 record, 4-7
 set, 4-8
 simple, 4-4
string, 4-9
structured, 4-5
symbolic, 5-27
cos function, 7-20

D

dangling pointers, 4-52
data
 access, 8-59
 allocation, 8-1
 dynamic, 8-22
 global, 8-22
 local, 8-22, 8-23, 8-24
 management, 8-22
 non-local, 8-22, 8-24
 packed and unpacked data access, 8-64
DEBUG compiler option, C-7
DEBUG/1000, 9-24
debugging a Pascal program, 9-23
debugging tools, 9-23
declaration part, 4-2
 routine, 4-31
declaration, module, 4-35
declarations
 array, 4-6
 constant, 4-4
 file, 4-23
 label, 4-3
 module, 4-35
 pointer, 4-16
 program, 4-1
 record, 4-20
 routine, 4-27
 set, 4-22
 string, 4-24
 type, 4-2
 variable, 4-26
default file names, use of, 9-10
dereferencing, pointer, 5-31
difference operator, 5-37
direct calling sequence, 8-77
DIRECT compiler option, C-8
direct-access files, 8-53
directives, 2-7, 4-32
dispose function, 7-3, 8-33
DIV operator, 5-35
dynamic allocation and procedures
 dispose, 7-3
 mark, 7-4
 new, 7-2
 release, 7-4
dynamic data, 8-22

E

efficiency considerations, 8-59
element type, 4-17
EMA addressing routines, 8-68
EMA compiler option, C-9
EMA errors, A-10
EMA heap management, 8-41
EMA_VAR compiler option, C-10
empty statement, 5-23
enumeration type, 4-14
eof function, 6-18
eoln function, 6-19
error message printers, A-10
error messages
 compile-time, A-12
 EMA, A-10
 FMP, A-9
 I/O, A-5
 load-time, 9-21
 run-time, 9-22
 segment, A-10
 warnings, A-12
ERROREXIT compiler option, C-10
EXEC calls, 9-37
executable parts, 5-1
exp function, 7-21
explicit import, 4-48
EXPORT section, 4-41
expressions,, 5-24, 8-70
extensions, Pascal language, 1-1
external routine declarations, 4-33

physical, 6-1, 6-6
position, 6-2, 6-30, 8-53
read-only, 6-3, 6-4, 6-7
program heading, 8-51
relationship between logical and FMP files, 8-54
reset, 6-43
restrictions, 8-58
rewrite, 6-45
scratch, 6-9
sequential, 6-1, 6-4
size, 6-2, 6-4, 6-6
source, 9-4
standard, 8-52
state, 6-2, 6-3, 6-4, 6-7
text, 6-2, 6-27
type, 6-1
write-only, 6-3, 6-4, 6-7
FIXED_STRING compiler option, C-11
FMP errors, A-9
FMP files, relationship between logical files, 8-54
FMP vs. Pascal I/O, 8-78
FOR statement, 5-17, 8-72
formal parameter list, 4-28
FORWARD directive, 2-7, 4-32
forward routine declarations, 4-32
function
 declaration, 4-27
 parameters, 4-28
 predefined, 2-6
 references, 5-41
 results, 4-30
functions and procedures, 8-74
 predefined, 2-6

F

FAST_REAL_OUT compiler option, C-11
field, 4-20
field selection, 5-20, 5-31
file buffer selection, 5-31
files
 associating logical and physical, 6-8
 automatic association, 8-52
 closing, 8-57
 current position pointer, 6-6
 declaration, 6-2
 direct-access, 8-53, 8-54
 in the heap, 8-53
 input, 8-52
 interactive I/O, 8-56
 number of components, 6-4
 opening, close options, 6-2, 6-4, 6-6
 output, 8-52
 parameters, 4-28

G

get procedure, 6-20
global data, 3-5, 4-55, 8-22, 8-19
globals, putting in heap, 8-80
GOTO statement, 5-22

H

halt procedure, 7-33
heading, 4-1
 module, 4-38
Heap Area, 8-20
HEAP compiler option, 4-28, 8-68, 8-69, C-12
heap management
 initialization, 8-29
 EMA, 8-41
 overview, 8-25

LIST_CODE compiler option, C-18

literals

 integer, 5-24

 real, 5-25

 string, 5-26

ln function, 7-21

load-time errors, 9-21

loaded program, reducing the size of, 8-79

loading a Pascal program, 9-17

loading a simple Pascal program, 9-20

loading segment overlays at run time, 3-12

local data, 8-22, 8-23, 8-24

local objects, 4-55

logical AND operator, 5-36

logical NOT operator, 5-36

logical OR operator, 5-36

LONGREAL, 4-12

M

main area, 8-13

main program block, 3-5

main program unit, 3-1, 3-3, 8-13

mark function, 7-4, 8-37

maxint, 4-11

maxpos function, 8-53

minint, 4-11

MIX compiler option, C-18

mixed heap programs, 8-45, 8-47

mixed listing, 9-28

MOD operator, 5-35

modifier, packed type, 4-25

module

 access rights, 4-48

 block, 3-6

 declaration, 4-35

 heading, 4-39

 interfaces, 4-45

 mixed heap, 8-47

 overlay considerations, 8-46, 8-48

 overview, 4-36

 relocating and searching, 8-46

 search pattern, 4-46

 structure, 4-38

 unique external names, 8-49

 unit, 3-6

 using files within, 8-48

multiplication operator, 5-35

multiply-dimensioned arrays, 4-18

N

short heap managers, 8-43

heap/stack, 8-15, 8-21

 area, 8-12, 8-18

HEAPPARMS compiler option, C-13

HEAP_DISPOSE compiler option, C-12

hex function, 7-25

I

I/O, 8-51

I/O errors, A-5

I/O procedures and functions, 6-14

I/O warnings, A-9

identical types, 5-44

identifiers

 legal, 2-5

 predefined, 2-6

IDSIZE compiler option, 2-4, C-16

IF statement, 5-10

IMAGE area, 8-15, 8-20

IMAGE compiler option, C-16

IMPLEMENT section, 4-42

implicit import, 4-50

IMPORT section, 4-40

IN operator, 5-40

INCLUDE compiler option, 3-8, C-16

INCLUDE_DEPTH compiler option, C-17

input, 8-52

integer, 4-11

integer division operator, 5-35

integer literals, 5-24

integer modulus operator, 5-35

interactive file I/O, 8-56

interfacing Pascal routines to other routines, 9-29

intermediate objects, 4-55

intersection operator, 5-37

K

KEEPASMB compiler option, C-17

L

LABEL declaration, 4-2, 4-3

language constructs, 1-8

lastpos function, 8-53

level-1 routines, 4-31

LINES compiler option, C-17

LINESIZE compiler option, C-18

LIST compiler option, C-18

listing, compiler, 9-14

naming restrictions, 8-58
new function, 7-2, 8-30
nil, 4-16
no-abort bit and error return, 9-40
NOABORT compiler option, C-19
non-cds memory configuration, 8-12
non-cds routine area, 8-19
non-cds stack management, 8-23
non-local data, 8-22, 8-24
NOT operator, 5-36
numbers, 2-8
numeric conversion functions
 binary, 7-25
 hex, 7-25
 octal, 7-25
numeric data types, 8-70

O

objects
 global, 4-55
 intermediate, 4-55
 local, 4-55
octal function, 7-25
odd function, 7-21
open procedure, 6-11
operands, 5-24
operator, string, 5-41
operators
 addition, 5-35
 arithmetic, 5-34
 Boolean, 4-11, 5-36
 concatenation, 5-41
 difference, 5-37
 IN, 5-40
 integer division, 5-35
 integer modulus, 5-35
 intersection, 5-37
 list of, 5-33
 logical AND, 5-36
 logical NOT, 5-36
 logical OR, 5-36
 multiplication, 5-35
 precedences, 5-32
 real division, 5-35
 relational operators, 5-39
 set, 5-37
 subtraction, 5-35
 unary +, 5-34
 unary -, 5-34
 union, 5-37
option file, 9-8
OR operator, 5-36

ord function, 7-27
ordinal functions
 chr, 7-26
 ord, 7-27
 pred, 7-28
 succ, 7-29
ordinal relational, 5-39
ordinal type identifier, 4-17
output, 8-52
overprint procedure, 6-27

P

PAC, 4-18
pack procedure, 7-30
packed array, 4-18
packed structures
 allocations for elements, 8-4
 array, 4-18
 examples of, 8-6
 record, 4-20
 vs. unpacked, 8-64
PACKED type modifier, 4-25
packing and unpacking procedures, 7-30, 7-32
PAGE compiler option, C-20
page procedure, 6-29
parameter list compatibility, 4-30
parameter list, formal, 4-28
parameter
 accessing, 8-59
 actual, 4-28
 call-by-reference, 4-29
 EMA, 4-28
 formal, 4-28
 function, 4-28
 HEAP 2, 4-28
 list, 4-28
 packed type, 4-28
 passing, 8-61
 procedure, 4-28
 string, 4-29
 value, 4-28
 variable, 4-28
partial evaluation, 5-36
PARTIAL_EVAL compiler option, C-20
Pas. Coalesce1 and Pas. Coalesce2 Procedures, D-19
Pas. DcbAddress1 procedure, D-7
Pas. DcbAddress2 procedure, D-8
Pas. ErrorPrinter procedure, A-10
Pas. FileNamr function, D-9
Pas. GetMemInfo1 procedure, 8-27, D-11
Pas. GetMemInfo2 procedure, 8-27, D-16
Pas. GetNewParms procedure, D-6

Pas.InitialHeap1 procedure, 8-27, D-14
 Pas.InitialHeap2 procedure, 8-27, D-18
 Pas.InitMemInfo1 procedure, D-10
 Pas.InitMemInfo2 procedure, D-15
 Pas.NumericParms procedure, D-5
 Pas.Parameters function, D-2
 Pas.SegmentLoad procedure, 3-12, D-21
 Pas.SetMemInfo1 procedure, 8-27, D-13
 Pas.SetMemInfo2 procedure, 8-27, D-17
 Pas.Sparameters function, D-4
 Pas.StrEndCheck routine, D-24
 Pas.StringData1 function, D-23
 Pas.StringData2 function, D-23
 Pas.TimeString procedure, D-20
 Pas.TraceBack procedure, D-22
 Pascal and FORTRAN, 9-32
 Pascal and IMAGE, 9-35
 PASCAL compiler option, C-21
 Pascal Extensions and Standards, 1-1
 Pascal program vocabulary, 1-7
 Pascal vs. FMP I/O, 8-78
 Pascal/Debug limitations, 9-25
PASCAL_CERR.REL, 8-79
PASCAL_ERR.REL, 8-79
 passing parameters, 8-61
 passing strings to other languages, 8-50
 physical files, 6-8
 pointers
 dereferencing, 5-31
 relational, 5-40
 dangling, 4-52
 position, file, 6-2, 6-30, 8-53
 position function, 8-53
 pred function, 7-28
 predeclared variables, 2-6
 predefined identifiers, 2-6
 predefined procedures and functions, 2-6
 predefined symbolic constants, 2-6
 predefined types, 2-6, 4-11
PRIVATE_TYPES compiler option, C-22
 procedure parameters, 4-29
 procedure statement, 4-27, 5-7
 procedure traceback, 9-27
 procedure tracing, 9-26
 procedures and functions, 8-74
 predefined, 2-6
 program control procedures, 7-33
 program heading, 4-1
 program heading files, 8-51
 program
 body, 3-5
 compiling, 9-3
 errors, A-1
 heading, 3-4
 identifier, 3-4
 main, 8-14
 parameter, 3-4
 programs, 8-45
 programs, mixed heap, 8-45
 prompt procedure, 6-31
 put procedure, 6-33

R

range checking, 8-71
RANGE compiler option, 8-71, C-22
 read procedure, 6-35
 read with text files, 6-35
 readdir procedure, 6-39
 readln procedure, 6-41
REAL, 4-12
 real division operator, 5-35
 real literals, 5-25
 record, 4-20
 recursion, 8-74
RECURSIVE compiler option, 4-34, C-23
 relational operators, 5-39
 relationship between logical files and FMP files, 8-54
 release function, 7-4, 8-39
 relocating modules, 8-46
REPEAT statement, 5-16
 reserved words, 2-3
 restrictions, naming, 8-58
RESULTS compiler option, C-23
 round function, 7-22
 routine
 body, 4-31
 declaration, 4-27
 declaration part, 4-31
 heading, 4-27
 names, 4-31
 recursive, 4-34
 run-time errors, A-2
 running a Pascal program, 9-22
RUN_STRING compiler option, C-24

S

scalar variables, allocation of, 8-1
 scope, 4-55
 scratch files, 6-9
SEARCH compiler option, C-24
 searching modules, 8-46
 section
 export, 4-41
 implement, 4-42
 import, 4-40

seek procedure, 6-47
SEGMENT compiler option, 3-10, C-25
 segment errors, A-10
 segment overlays, 3-12, 8-15
 segment unit, 3-1, 3-10, 3-11
SEGMENTED compiler option, C-26
 selectors, 5-29
 separators, 2-9
 sequential files, 8-54
 sets
 constant, 4-8
 constructor, 5-38
 operators, 5-37
 relational, 5-40
 set type, 4-22
 setstrlen procedure, 7-5
 shareable EMA access, 8-68, 8-69, D-28
 simple constants, 4-4
 sin function, 7-22
SKIP_TEXT compiler option, C-26
SMALL_TEMPS compiler option, C-27
 source file, 9-4
 space considerations, 8-74
 special symbols, 2-2
 sqr function, 7-22
 sqrt function, 7-23
 stack area, 8-20
 stack management, 8-24
 standard files, 8-52
 standards, Pascal, 1-1
STANDARD_LEVEL compiler option, C-27
 statement labels, 5-4
 statements
 assignment, 4-30, 5-5
 CASE, 5-12, 8-73
 compound, 5-9
 conditional, 5-10
 empty, 5-23
 FOR, 5-17, 8-72
 GOTO, 5-22
 IF, 5-10
 procedure call, 5-7
 REPEAT, 5-16
 repetitive, 5-16
 simple, 5-2
 structured, 5-2, 5-9
 WHILE, 5-15
 WITH, 5-20, 5-31, 8-72
STATS compiler option, C-27
 str function, 7-12
 strappend procedure, 7-6
 strdelete procedure, 7-7
 string
 concatenation, 5-41
 constant, 4-9
 literal, 5-26
 relational, 5-40
 subscripts, 5-30
 string procedures and functions
 str, 7-12
 strlen, 7-13
 strltrim, 7-14
 strmax, 7-15
 strpos, 7-16
 strrpt, 7-17
 strrtrim, 7-18
 setstrlen, 7-5
 strappend, 7-6
 strdelete, 7-7
 strinsert, 7-8
 strmove, 7-9
 strread, 7-10
 strwrite, 7-11
 strlen function, 7-13
 strltrim function, 7-14
 strmax function, 7-15
 strmove procedure, 7-9
 strpos function, 7-16
 strread procedure, 7-10
 strrpt function, 7-17
 strrtrim function, 7-18
 structured constants, 4-5, 8-81
 structured statements, 5-9
 structured type, 4-17
 structured variables, allocations for, 8-1
 strwrite procedure, 7-11
 subexpressions, common, 8-70
SUBPROGRAM compiler option, C-30
 subprogram unit, 3-1, 3-7, 8-13
 subrange, 4-15
 subscripts
 array, 5-30
 string, 5-30
SUBTITLE compiler option, C-29
 subtraction operator, 5-35
 succ function, 7-29
 summary of the Pascal/1000 language, 1-2
 symbolic constants, 5-27
 predefined, 2-6
 symbols
 basic, 2-2
 special, 2-2
 system common access, D-26

T

TABLES compiler option, C-29
tag type, 4-20
text, 4-13
text files, 8-56
time considerations (CDS), 8-75
time considerations (Non-CDS), 8-76
TITLE compiler option, C-34
TRACE compiler option, C-34
TRACE__BACK compiler option, C-36
trace libraries, 9-26
trunc function, 7-23
type
 array, 4-17
 assignment compatible, 5-45
 base, 4-22
 compatible, 5-44
 definition, 4-10
 files, 4-23
 identical, 5-44
 packed, 4-25
 predefined, 2-6
 record, 4-20
 set, 4-22
 string, 4-24
 structured, 4-17
 tag, 4-20
 user-defined, 4-14

V

value parameters, 4-28
VAR, 4-29
variable declaration, 4-26
variable parameters, 4-29
variables
 accessing, 8-59
 component, 5-28
 dynamic, 4-16
 entire, 5-28
 predeclared, 2-6
 selected, 5-28
variant record, 4-20
VMA/EMA shared access, 8-68, 8-69

W

WARN compiler option, C-36
warning messages, I/O, A-9
warnings
 compile-time, A-12
 run-time, A-2
WHILE statement, 5-15
WIDTH compiler option, C-36
WITH statement, 5-20, 5-31, 8-72
WORK compiler option, C-37

U

unary + operator, 5-34
unary - operator, 5-35
union operator, 5-37
unpack procedure, 7-32
unpacked structures, examples of, 8-6
user-defined types, 4-14

READER COMMENT SHEET

PASCAL/1000
Reference Manual

92833-90005

March 1985

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 _____

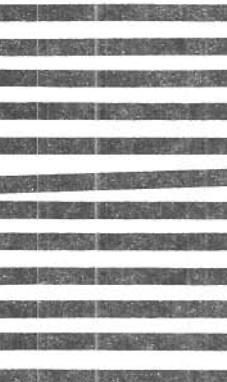
Address _____

Phone No. _____ Ext. _____

OLD



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

A series of twelve thick, dark horizontal bars arranged in three groups of four, used for postal processing.

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 141 CUPERTINO, CA.

— POSTAGE WILL BE PAID BY —

Hewlett-Packard Company

Computer Language Lab
19447 Pruneridge Avenue
Cupertino, California 95014
ATTN: Technical Publications

OLD



HEWLETT-PACKARD COMPANY
Computer Language Lab
19447 Pruneridge Avenue

MANUAL PART NO. 92833-90005
Printed in U.S.A. September 1984
U0385