64000

HP64000 Logic Development System

Model 64817A HP64000 HOST Pascal

CERTIFICATION

Hewlett-Packard Company certifies that this product met its published specifications at the time of shipment from the factory. Hewlett-Packard further certifies that its calibration measurements are traceable to the United States National Bureau of Standards, to the extent allowed by the Bureau's calibration facility, and to the calibration facilities of other International Standards Organization members.

WARRANTY

This Hewlett-Packard system product is warranted against defects in materials and workmanship for a period of 90 days from date of installation. During the warranty period, HP will, at its option, either repair or replace products which prove to be defective.

Warranty service of this product will be performed at Buyer's facility at no charge within HP service travel areas. Outside HP service travel areas, warranty service will be performed at Buyer's facility only upon HP's prior agreement and Buyer shall pay HP's round trip travel expenses. In all other cases, products must be returned to a service facility designated by HP.

For products returned to HP for warranty service. Buyer shall prepay shipping charges to HP and HP shall pay shipping charges to return the product to Buyer. However, Buyer shall pay all shipping charges, duties, and taxes for products returned to HP from another country.

HP warrants that its software and firmware designated by HP for use with an instrument will execute its programming instructions when properly installed on that instrument. HP does not warrant that the operation of the instrument, or software, or firmware will be uninterrupted or error free.

LIMITATION OF WARRANTY

The foregoing warranty shall not apply to defects resulting from improper or inadequate maintenance by Buyer, Buyer-supplied software or interfacing, unauthorized modification or misuse, operation outside of the environmental specifications for the product, or improper site preparation or maintenance.

NO OTHER WARRANTY IS EXPRESSED OR IMPLIED. HP SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

EXCLUSIVE REMEDIES

THE REMEDIES PROVIDED HEREIN ARE BUYER'S SOLE AND EXCLUSIVE REMEDIES. HP SHALL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER BASED ON CONTRACT, TORT, OR ANY OTHER LEGAL THEORY.

ASSISTANCE

Product maintenance agreements and other customer assistance agreements are available for Hewlett-Packard products.

For any assistance, contact your nearest Hewlett-Packard Sales and Service Office.

HP64000
HOST Pascal

LOGIC SYSTEMS DIVISION
COLORADO SPRINGS, COLORADO, U. S. A.

Printing History

Each new edition of this manual incorporates all material updated since the previous edition. Manual change sheets are issued between editions, allowing you to correct or insert information in the current edition.

The part number on the back cover changes only when each new edition is published. Minor corrections or additions may be made as the manual is reprinted between editions.

First Edition.....January, 1982 (P/N 64817-90902)
Second Edition.....May, 1982 (P/N 64817-90903)
Third Edition.....December, 1983 (P/N 64817-90904)

TABLE OF CONTENTS

Chapter 1: General Information
Introduction....1-1
HOST Pascal Compiler....1-1
HOST Pascal Extensions....1-1

Chapter 2: Source Code Considerations
Introduction....2-1
Character Set....2-1
  Alphabetic Characters....2-1
  Numeric Characters....2-1
  Special Characters....2-1
Reserved Words....2-4
Identifiers....2-5
Predefined Identifiers....2-6
  Predefined Procedures....2-6
  Predefined Functions....2-6
  Predefined Files....2-6
  Predefined Types....2-7
  Predefined Constants....2-7
Directives....2-7
String Literals....2-7
Comments....2-8
HOST Compiler Options....2-8

Chapter 3: General Form Of A Pascal Program
Introduction....3-1
Main Program Module....3-1
Program Heading....3-1
Program Block....3-2
Segment....3-4

Chapter 4: Pascal Program Declarations
Introduction....4-1
Declaration Section....4-2
  LABEL Declaration....4-2
  CONSTant Declaration....4-3
    Simple Constants....4-6
    Structured Constants....4-6
    Array Constants....4-7
    RECORD Constant....4-8
    SET Constant....4-9
    String Literal....4-10
TYPE Definitions....4-10
TYPE....4-10
Type Declaration....4-11
  Simple Types....4-11
  Ordinal Types....4-11
  Predefined Ordinal Types....4-12
    BOOLEAN....4-12
    CHAR....4-12
    INTEGER....4-13

Table of Contents (Cont'd)

User Defined Ordinal Types....4-13
Enumerated Type....4-13
Subrange Type....4-13
Real Types....4-14
REAL....4-14
LONGREAL....4-14
Pointer Types....4-15
Structured Types....4-15
Array....4-15
PAC....4-17
String Data Types....4-17
RECORD....4-19
SET....4-21
FILE....4-21
PACKED Type Modifier....4-22
VARiable....4-23
VARiable Declaration....4-23
Entire Variables....4-24
Component Variables....4-24
Indexed Variables....4-24
Field Designators....4-24
Buffer Variables....4-24
Referenced Variables....4-25
Routine Declarations....4-25
PROCEDURE Declaration....4-25
FUNCTION Declaration....4-26
Parameter Lists....4-27
Formal Parameter List....4-27
Actual Parameter List....4-28
Value Parameter....4-28
Variable Parameter....4-28
Procedure Parameter....4-29
Function Parameter....4-29
Parameter List Compatibility....4-29
Declarations Within Routines....4-29
Routine Body....4-29
Directives....4-30
Recursive Routines....4-30
Scope....4-30
Statements....4-31

Chapter 5: Statements and Expressions
Introduction....5-1
Statement Label....5-2
Assignment Statement....5-2
Procedure Statement....5-3
Compound Statement....5-5
IF Statement....5-5
CASE Statement....5-6
WHILE Statement....5-7
REPEAT Statement....5-8

Table of Contents (Cont'd)

FOR Statement....5-8
WITH Statement....5-9
GOTO Statement....5-10
Empty Statement....5-11
Expressions....5-11
Operands....5-13
Literals....5-13
  Integer Literals....5-14
  Real Literals....5-14
  String Literals....5-14
Symbolic Constants....5-14
Variables....5-15
Selectors....5-15
Array Subscripts....5-16
Field Selection....5-16
Pointer Dereferencing....5-16
File Buffer Selection....5-16
Operators....5-17
  Arithmetic Operators....5-17
  Boolean Operators....5-18
    NOT....5-18
    AND....5-18
    OR....5-18
  Set Operators....5-19
    Set Union....5-19
    Set Difference....5-19
    Set Intersection....5-19
Set Constructor....5-20
String Operators....5-21
Relational Operators....5-21
  Ordinal Relationals....5-21
  PAC Relationals....5-22
  String Comparison....5-22
  Pointer Relationals....5-22
  Set Relationals....5-23
Function References....5-23
Constant Expressions....5-24
  Operators....5-24
  Predefined Functions....5-24
  Operands....5-24
Type Compatibility....5-25
  Identical Types....5-25
  Compatible Types....5-25
  Assignment Compatible Types....5-26
Special Cases....5-27

Chapter 6: Files
Introduction....6-1
Logical Files....6-1
Sequential Files....6-2
Physical Files....6-2

Table of Contents (Cont'd)

Textfiles....6-2
Logical File Characteristics....6-2
File Buffer Variable....6-3
Current Position Pointer....6-3
File States....6-3
Opening Files....6-4
RESET....6-4
REWRITE....6-4
APPEND....6-5
Associating Logical and Physical Files....6-5
Associating Files Through the String Parameter....6-6
Sequential File Operations....6-6
Textfile Operation....6-6
GET(f)....6-6
PUT(f)....6-7
READ(f,v)....6-7
READ(f,v) With Textfiles....6-8
READLN(f,v)....6-9
WRITE(f,e)....6-10
WRITE(f,v) With Textfiles....6-11
WRITELN(f,p)....6-12
PAGE(f)....6-13
LINEPOS(f)....6-13
EOLN(f)....6-13
Closing Files....6-13
Summary of Procedures and Functions....6-14

Chapter 7: Standard Procedures and Functions
File-Handling Procedures....7-1
APPEND(f)....7-1
RESET(f)....7-2
REWRITE(f)....7-2
CLOSE(f)....7-2
GET....7-2
PAGE....7-3
PUT....7-3
READ....7-3
READLN....7-4
TIMEOUT(f,t)....7-4
WRITE....7-5
WRITELN....7-5
String Handling Procedures and Functions....7-6
SETSTRLEN....7-6
STRAPPEND....7-6
STRINSERT....7-6
STRDELETE....7-7
STRMOVE....7-7
STRLEN....7-8
STRMAX....7-8
STR....7-8
STRLTRIM....7-9

Table of Contents (Cont'd)

STRRTRIM....7-9
STRREAD....7-9
STRWRITE....7-10
STRRPT....7-10
STRPOS....7-10
Dynamic Allocation and De-allocation Procedures....7-10
General Information....7-10
NEW(p)....7-11
DISPOSE(p)....7-12
MARK(p)....7-12
RELEASE(p)....7-12
Transfer Procedures....7-12
PACK....7-12
UNPACK....7-14
Arithmetic Functions....7-15
Abs....7-15
Sqr....7-15
Sqrt....7-15
Exp....7-15
Ln....7-15
Sin, Cos....7-16
Arctan....7-16
Predicates....7-16
Odd....7-16
Eof....7-16
Eoln....7-16
Transfer Functions....7-16
Trunc....7-16
Round....7-17
Ordinal Functions....7-17
Ord....7-17
Chr....7-18
Succ....7-18
Pred....7-18
Numeric Conversion Functions....7-19
HEX....7-19
OCTAL....7-19
BINARY....7-19
File Handling Functions....7-19
LINEPOS....7-19
POSITION....7-19
IORESULT....7-19

Chapter 8: Implementation of HOST Pascal
Introduction....8-1
Data Allocation....8-1
Allocation for Scalar Variables....8-1
Allocation for Structured Variables....8-3
Allocation for Elements of Packed Structures....8-4

Table of Contents (Cont'd)

Examples of Packed and Unpacked Structure Allocation.....8-6
  Example 1: Unpacked structure allocation.....8-6
  Example 2: Packed Record Allocation With Unpacked Array.....8-8
  Example 3: Packed Record Allocation With Packed Array.....8-10
Memory Allocation.....8-11
I/O Error Handling.....8-12

Chapter 9: Using HOST Pascal
The Source File.....9-1
Compiling The Source File.....9-1
Command Parameters.....9-2
  Source File.....9-2
  Listfile.....9-3
  Options.....9-3
Running HOST Pascal Programs.....9-4
Run Command Parameters.....9-4
Additional Program Parameters.....9-6

Appendix A: Compile-Time and Run-Time Error Messages
Compile-Time Error Messages.....A-1
Run-Time Error Messages.....A-5
  Legend.....A-5

Appendix B: Sample Pascal Programs
Sample Programs.....B-1

Appendix C: Input/Output Characteristics
Introduction.....C-1
Disc Files.....C-1
Text Files.....C-1
Non-text Files.....C-2
I/O Devices.....C-2
  Null.....C-2
  Keyboard.....C-2
  Display.....C-3
  Printer.....C-3
  Display1.....C-3
  RS-232.....C-6
    Hardware Options.....C-7
    Receiver and Transmitter.....C-7
    Character and Protocol Transparency.....C-7
    Circular Buffers.....C-8
    TIMEOUT(F,T) Procedure.....C-8
    RS-232 Receiver Operation.....C-9
    RS-232 Transmitter Operation.....C-10
    Modem Control.....C-11
    Restricted Use of READ and READLN.....C-11
    Timing Considerations.....C-11
Example Program of RS-232 Implementation.....C-13

Index.....I-1

LIST OF ILLUSTRATIONS

2-1. Identifier Syntax....2-5
2-2. Compiler Options Syntax....2-9
3-1. Program Syntax....3-1
3-2. Heading Syntax....3-1
3-3. Program Block Syntax....3-3
4-1. Declaration Syntax....4-2
4-2. LABEL Declaration Syntax....4-3
4-3. CONSTant Declaration Syntax....4-4
4-4. String Literal Syntax....4-5
4-5. Integer Syntax....4-5
4-6. Real Syntax....4-5
4-7. Structured Constant Syntax....4-6
4-8. Array Constant Syntax....4-7
4-9. RECORD Constant Syntax....4-8
4-10. SET Constant Syntax....4-9
4-11. TYPE Declaration Syntax....4-11
4-12. TYPE Syntax....4-12
4-13. Enumerated Type Syntax....4-13
4-14. Subrange Type Syntax....4-14
4-15. Pointer Type Syntax....4-15
4-16. Array Syntax....4-16
4-17. String Type Syntax....4-18
4-18. Record Type Syntax....4-20
4-19. Field List Syntax....4-20
4-20. Set Type Syntax....4-21
4-21. File Type Syntax....4-22
4-22. PACKED Modifier Syntax....4-22
4-23. VARiable Declaration Syntax....4-23
4-24. Routine Declaration Syntax....4-25
4-25. PROCEDURE Heading Syntax....4-25
4-26. FUNCTION Heading Syntax....4-26
4-27. Formal Parameter List Syntax....4-27
4-28. Actual Parameter List Syntax....4-28
4-29. Statement List Syntax....4-31
5-1. Statement Syntax....5-2
5-2. Assignment Statement Syntax....5-3
5-3. Procedure Statement Syntax....5-4
5-4. Compound Statement Syntax....5-5
5-5. IF Statement Syntax....5-6
5-6. CASE Statement Syntax....5-7
5-7. WHILE Statement Syntax....5-8
5-8. REPEAT Statement Syntax....5-8
5-9. FOR Statement Syntax....5-9
5-10. WITH Statement Syntax....5-10
5-11. GOTO Statement Syntax....5-11
5-12. Expression Syntax....5-11
5-13. Simple Expression Syntax....5-12
5-14. Term Syntax....5-12
5-15. Factor Syntax....5-13
5-16. Selector Syntax....5-15
5-17. Set Constructor Syntax....5-20

List of Illustrations (Cont'd)

6-1. READ INTEGER Syntax....6-8
6-2. READ REAL or LONGREAL Syntax....6-9
8-1. Unpacked Structure Allocation....8-7
8-2. Packed Record Allocation With Unpacked Array....8-9
8-3. Packed Record Allocation With Packed Array....8-11
9-1. Compile Command Syntax....9-2
9-2. Source File Name Syntax....9-2
9-3. Run Command Syntax....9-4

LIST OF TABLES

6-1. Field Width Parameter Default Values....6-11
6-2. Procedure and Function to File Association....6-14
7-1. IORESULT Definitions....7-20
8-1. Allocations for Scalar Variables....8-2
8-2. Allocations for Structured Variables....8-3
8-3. Allocations for Elements of Packed Structures....8-4

Chapter 1
General Information

Introduction

This manual describes the Hewlett-Packard HOST Pascal compiler and its operation on the HP Model 64000 Logic Development Station.

HOST Pascal Compiler

The HOST Pascal compiler is an application program that translates Pascal source programs into a pseudo opcode file that will execute on the HP Model 64000.

The HOST Pascal language is a superset of "standard" Pascal. "Standard" here refers to the ISO (International Standards Organization) Pascal Standard.

It is assumed that the user has some knowledge of the Pascal language as defined by Jensen and Wirth and other reference books on standard Pascal.

HOST Pascal Extensions

HOST Pascal contains enhancements that provide more versatility for programming.

- The CASE statement can have an OTHERWISE clause.
- Constant-value expressions are allowed in places where a constant is allowed in standard Pascal.
- CONST, TYPE, and VAR declaration sections may be in any order after the optional LABEL declaration section and any number of CONST, TYPE, or VAR declaration sections is allowed.
- Dynamic memory procedures, MARK and RELEASE, may be used in the same program using the standard memory management procedures NEW and DISPOSE.
- Compiler directives may be specified in a flexible manner using the $ character as a delimiter.
- Constant lists in the CASE statement and RECORD variant definitions may use the ".." symbol to denote a range of values.
- Constant arrays, records, and sets may be defined in the CONST declaration section.

- Functions may return a value of any assignable data type.

- There is an extended syntax for string literals that uses the # character to allow non-printing characters to be incorporated into string literals.

- There is a predefined double precision real data type called LONGREAL.

- There is a predefined family of data types called STRINGs. These are similar to packed arrays of character but have a dynamic length attribute that allows the number of characters contained in the string to be variable.

- There are a number of predefined procedures and functions that perform useful operations with STRING data types.

- Predefined procedures APPEND and CLOSE, and predefined functions POSITION and LINEPOS enhance input/output capability.

- Predefined functions HEX, OCTAL, and BINARY perform number base conversions.

Chapter 2

Source Code Considerations

Introduction

The HOST Pascal Compiler will run on any HP 64000 system, provided that the memory expansion module is present in the logic chassis. The HOST Pascal Compiler allows the user the development of Pascal programs for the execution of repetitive tasks, performing mathematical calculations, and solving problems. Basically, the HOST compiler allows the 64000 system to be used as a mini-computer.

The HOST compiler accepts as input a sequence of statements from one or more source code files for conversion into a quasi-machine code which is stored in the 64000 for future use. This chapter discusses some source code features that must be considered when writing a source program.

Character Set

Alphabetic Characters

The alphabetic characters include all upper and lower case characters, i.e., 'A' thru 'Z' and 'a' thru 'z'.

Numeric Characters

The numeric characters include the digits 0 thru 9 for decimal numbers.

Special Characters

The special character set (symbols) and their use in Pascal programming are described as follows:

Character (Symbol)	Description	Use
'	Apostrophe	String Literal Delimiter.
*	Asterisk	Arithmetic operator-multiply; set intersection.
(*...*)	Asterisk/Parenthesis	Comment delimiters.
{...}	Braces	Comment delimiters.

NOTE

The "(*" and "{" comment delimiters are equal in power and can be used interchangeably without regard to the right delimiter used. The "}" and "*)" comment delimiters are equal in power and can be used interchangeably without regard to the left delimiter used.

[...]	Brackets	Set constructor; structured constant delimiter; array index delimiters.
(. ... .)	Parenthesis/Period	Set constructor; structured constant; array index delimiters.

NOTE

The "[" and "(." set delimiters are equal in power and can be used interchangeably without regard to the right delimiter used. The "]" and ".)" set delimiters are equal in power and can be used interchangeably without regard to the left delimiter used.

:	Colon	Case constant list delimiter; statement label delimiter; field width delimiter; identifier list delimiter.
,	Comma	Argument list separator; structured constant value list separator; enumerated type list separator.
$	Dollar sign	Compiler option (directive) delimiter.
=	Equal sign	Equality (relational operator).
-	Minus sign	Arithmetic operator-subtract/ negate; set difference.
(...)	Parentheses	Delimits a parameter list or an expression group; delimits an enumerated type.
.	Period	End of program; decimal point; field selector.
+	Plus sign	Arithmetic operator-addition; string concatenation; set union.

#	Pound sign	Indicates non-printing character in string constant.
;	Semicolon	Parameter separator; statement separator.
/	Slant bar	Arithmetic operator-real divide.
_	Underscore	Allowed in identifiers but not as first character.
^	Caret	Indicates file buffer accessing; indicates pointer dereferencing.
@	At	Indicates file buffer accessing; indicates pointer dereferencing.

NOTE

The caret and "at" are equal in power and can be used interchangeably.

:=	Symbol	Assignment indicator.
>	Symbol	Greater than (relational operator).
>=	Symbol	Greater than or equal to (relational operator); superset of.
<	Symbol	Less than (relational operator).
<=	Symbol	Less than or equal to (relational operator); subset of.
<>	Symbol	Not equal (relational operator).
..	Symbol	Subrange.

Reserved Words

The following words are reserved. They have special meaning to the HOST compiler and cannot be used as identifiers or user-defined symbols.

Reserved Word(s)	Description
AND	Boolean conjunction operator.
ARRAY	A structured type.
BEGIN, END	Delimit a compound statement.
CASE, OF, OTHERWISE	A conditional statement.
CONST	Indicates constant definition section.
DIV	Integer division operator.
FILE	A structured type.
FOR, TO, DOWNTO, DO	A repetitive statement.
FUNCTION	Indicates a function declaration.
GOTO	Control transfer statement.
IF, THEN, ELSE	A conditional statement.
IN	Set inclusion operator.
LABEL	Indicates label definition section.
MOD	Integer modulus operator.
NIL	Special pointer value.
NOT	Boolean negation operator.
OR	Boolean disjunction operator.
PACKED	Controls storage allocation for structured 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 scope(s).

Identifiers

Identifiers are selected by the programmer to denote constants, types, variables, procedures, functions, and programs. An identifier consists of a letter followed by any combination of upper-case or lower-case letters, digits, or underscore (_). The syntactical construction of an identifier is shown in Figure 2-1.

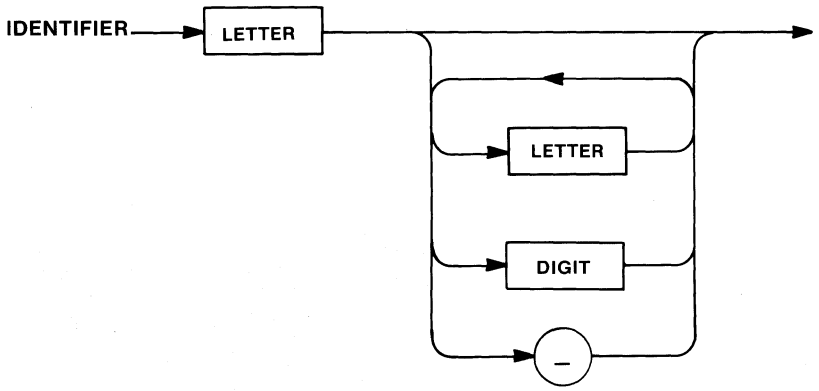

Figure 2-1. Identifier Syntax

When constructing identifiers, the following rules should be observed:

a. The first character of an identifier must be a letter.

b. Identifiers may contain any number of characters up to a source line in length. All characters are significant.

c. A reserved word cannot be used as an identifier.

d. Upper and lower case letters are not differentiated within identifiers.

e. Each identifier must be unique within its scope, i.e., with a procedure or function in which they are defined.

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

parameter may refer to a file variable that is declared in the program block.

Predefined Identifiers

There are certain predefined identifiers that will be recognized by the HOST compiler without being defined in the program. These predefined identifiers are listed in the following paragraphs.

Predefined Procedures

The following list of predefined procedures will be recognized by the compiler without further definition (refer to Chapter 7 for an explanation of each procedure):

append	new	reset	strread
close	pack	rewrite	strwrite
dispose	page	setstrlen	timeout
get	put	strappend	unpack
halt	read	strdelete	write
mark	readln	strinsert	writeln
	release	strmove	

Predefined Functions

The following list of predefined functions will be recognized by the compiler without further definition (refer to Chapter 7 for an explanation of each function):

abs	exp	position	strltrim
arctan	hex	pred	strmax
binary	ioresult	round	strpos
chr	linepos	sin	strrpt
cos	ln	sqr	strrtrim
eof	octal	sqrt	succ
eoln	odd	str	trunc
	ord	strlen	

Predefined Files

The following predefined files will be recognized by the compiler without further definition; they must, however, be indicated in the program parameter list if they are used in the program. (Refer to Chapter 6 for an explanation of each file.)

input	output

Predefined Types

The following list of predefined types will be recognized by the compiler without further definition (refer to Chapter 4 for an explanation of each type):

boolean	integer	real	text
char	longreal	string	

Predefined Constants

The following list of predefined constants will be recognized by the compiler without further definition (refer to Chapter 5 for an explanation of each constant):

false	minint
maxint	true

Directives

A directive introduces a procedure or function declaration for which there is no block specified.

forward	Indicates to the compiler that a block for the routine appears later in the program.

String Literals

A string literal is a sequence of ASCII characters enclosed by apostrophes. String literals are constants of the PAC type or the string type. String literals containing a single character may also be of the predefined type CHAR.

a. Non-printing (and other) characters are encoded after a pound symbol (#).

   Example:

   ```
   #48 is the representation for '0'.
   ```

b. If a string literal is to contain an apostrophe, the apostrophe is written twice.

c. String literals must be contained on a single line.

d. Characters between apostrophes must be in the range chr(32) thru chr(126), i.e., 'space' thru 'tilde'.

Additional information on string literals and character constants may be found in Chapter 5.

Comments

Words and messages contained in braces {...}, or parentheses/asterisks (*...*), are comments used to document the program. Comments are ignored by the compiler. A comment has the form:

{character string} or (*character string} or

(*character string*) or {character string*)

Conventions to be observed when using comments are listed below:

a. Since a comment is equivalent to a blank, it may be placed anywhere in the program that a blank is permitted.

b. A comment beginning with a left brace ({) need not terminate with a right brace (}). A comment beginning with the compound symbol (* need not terminate with the compound symbol *).

HOST Compiler Options

The compiler interprets the following construct as a compiler directive:

$<compiler_options>$

where <compiler_options> may be any or all of the following:

ANSI	PAGE	LINESIZE
INCLUDE	IOCHECK	TITLE
BUFFERS	RANGE	XREF
NOSAVE	CODE	SWAP
EXTENSIONS	PARTIAL_EVAL	SHORTID
LIST	WIDTH	SEGMENT

Compiler options may be inserted between any two tokens (identifiers, numbers, string literals, and special symbols). They are used to inform the compiler about changing needs within the program. A compiler option is a separator (as is a space or a comment) in the Pascal program. Compiler options must begin with a dollar sign and close with a dollar sign. A compiler error will result if no closing dollar sign appears on the line. A compiler option must exist entirely on one line.

The compiler option specification may include an option value. If no option value is specified, then, for options that require an ON - OFF value, ON is assumed. Otherwise, for options that require an integer or string literal value, the option is set to its default value.

The compiler option syntax is defined here, and the syntax diagram is shown in Figure 2-2.

```
<compiler options> ::= <option> {<separator> <option>}
<option>           ::= <identifier> <option value> | <empty>
<option value>     ::= ON | OFF | <signed integer> |
                                  <string literal> | <empty>
<separator>        ::= ; | ,
```

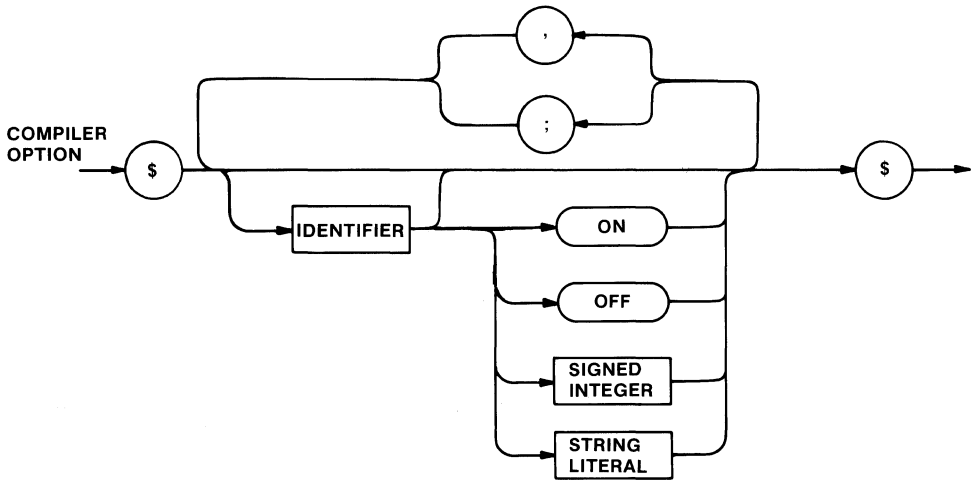

Figure 2-2. Compiler Options Syntax

The compiler options are described in the following paragraphs.

ANSI — The compiler option ANSI, when ON, causes warning messages to be issued for all features of HOST Pascal that are not part of ANSI Standard Pascal. Default is OFF.

INCLUDE — The compiler option INCLUDE is followed by a string literal that names a file containing text to be included at the current position in the program. The included code may not contain additional INCLUDE options. The remainder of the line, after the closing '$', must be blank. INCLUDE has no default.

BUFFERS — The compiler option BUFFERS specifies the number of 128 word blocks to be made available for the file buffer. Allowable values are 1, 2, 4, 8, or 16. This option applies to all succeeding variable definitions involving files. If the standard files INPUT and OUTPUT appear in the program heading and the BUFFERS option is to apply to them, the option must appear before the program heading. Default is 1.

NOSAVE — The compiler option NOSAVE specifies that the values of structured constants, i.e., constant arrays and constant records, will not be available to use in a later structured constant definition. This can result in a significant savings of memory at compile time. Default is OFF.

EXTENSIONS — The Compiler option EXTENSIONS allows language features that are extensions to HP Standard Pascal to be used. Default is OFF.

LIST — The compiler option LIST, when off, suppresses source code listing, except for lines that contain errors. Default is ON.

PAGE — The compiler option PAGE causes listing to continue at the top of the next page if LIST is ON. PAGE has no default value.

IOCHECK — The compiler option IOCHECK specifies that code will be emitted so that an I/O error will cause an immediate run-time error. Default is ON.

RANGE — The compiler option RANGE specifies that run-time checks of array index, parameter, and assignment values be carried out. Default is ON.

CODE — The compiler option CODE specifies that object code will be generated. If CODE OFF is ever specified, no object file will be generated regardless of any other CODE ON directives. Default is ON.

PARTIAL_EVAL — The compiler option PARTIAL_EVAL, when ON, causes evaluation of a Boolean expression to cease when the left operand evaluates to an opposing value. Evaluation ceases when the operand preceding AND evaluates false, and when the operand preceding OR evaluates TRUE. The option, when OFF, forces evaluation of the entire Boolean expression. Default is OFF.

WIDTH — The compiler option WIDTH specifies the number of significant characters in a source line. Additional characters are ignored. Default is 240 characters.

LINESIZE — The compiler option LINESIZE specifies the maximum number of characters in a line that a text file will be able to handle. It applies to all successive variable declarations that involve text files. If the standard files INPUT and OUTPUT appear in the program heading and the LINESIZE option is to apply to them, it must appear before the program heading. Default is 240.

TITLE — The compiler option TITLE specifies that the string parameter will be printed at the top of any subsequent pages of the listing, provided that LIST is ON. Default is all blanks.

XREF — The compiler option XREF specifies that a cross reference of the program will be generated at the end of the listing. Default is OFF.

SWAP — The compiler option SWAP, when ON, allows the compiler to compile a considerably larger program but at the cost of lower compilation speed. The compiler is roughly divided into two segments or overlays. One segment processes the declaration portion of a block, and the other segment processes the executable statements. These compiler segments are swapped in and out of memory as needed during the course of the compilation. Because all the compiler code is not in memory at once, more working space and symbol table space is available to the compiler. The SWAP option, if it is used, must be specified before the PROGRAM heading. Otherwise, it is ignored. Default is OFF.

SHORTID — The compiler option SHORTID, when ON, causes only the first six characters of identifiers to be significant. This can save memory space during compilation since only the first six characters of identifiers are stored in the symbol table. Ths SHORTID option must be specified before the PROGRAM heading if it is to be used. Otherwise, it is ignored. Default is OFF.

SEGMENT — The compiler option SEGMENT specifies that the p-code for the next procedure or function will be put into a separate segment or overlay. (See Chapter 3 for more detail.) There is no value associated with the SEGMENT option.

Chapter 3

General Form Of A Pascal Program

Introduction

Every Pascal program consists of a main program module and may contain as many procedure and function routines as necessary to properly execute the program.

Main Program Module

The compilation unit begins with the special compiler directive, "HOST", followed by the main program.

A main program contains a heading and a block, and concludes with a period (.) as indicated in Figure 3-1.

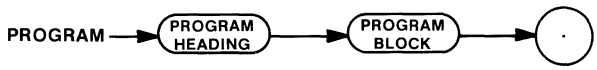

Figure 3-1. Program Syntax

Program Heading

The word PROGRAM is a reserved word and is always the first word of a Pascal program heading. The program heading may specify a list of program parameters, as shown in Figure 3-2.

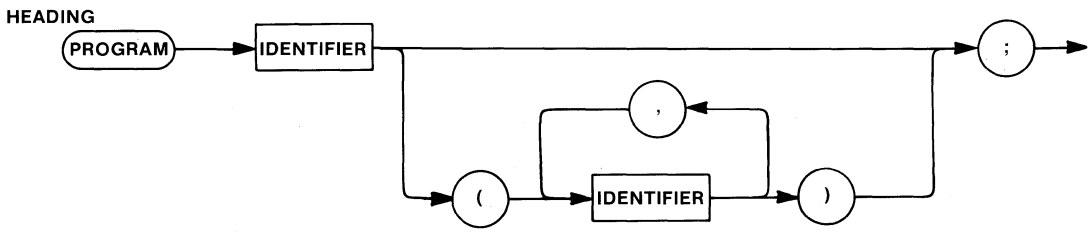

Figure 3-2. Heading Syntax

The identifier gives the program a name, but has no other significance within the program.

The program parameter list contains identifiers of type FILE through which the program communicates with the external environment. All of the file variables listed as program parameters will be associated with the name of a physical file or an I/O device when the program is executed. In the case of files INPUT and OUTPUT a file name is specified as part of the run command. For each additional program parameter, after the run command is issued and before program execution, the system will prompt the operator for the name of a physical file or I/O device.

A few examples of program headings follow:

```
"HOST"
PROGRAM execute;

"HOST"
PROGRAM applepie (input, output);

"HOST"
PROGRAM payroll (data, output);
```

NOTE

All identifiers must be defined before they are used, except that a pointer type may reference a type name that is defined later in the same declaration section, and a program parameter may reference a variable that is defined in the program block.

Program Block

The program block contains declarations and a compound statement. The declaration consists of labels, constants, types, variables, procedures, and functions. Identifiers and labels declared in the main program block are known as global identifiers and labels. The compound statement in the main program block is known as the program body. Execution of the program begins at the compound statement. The syntax diagram for the program block is shown in Figure 3-3.

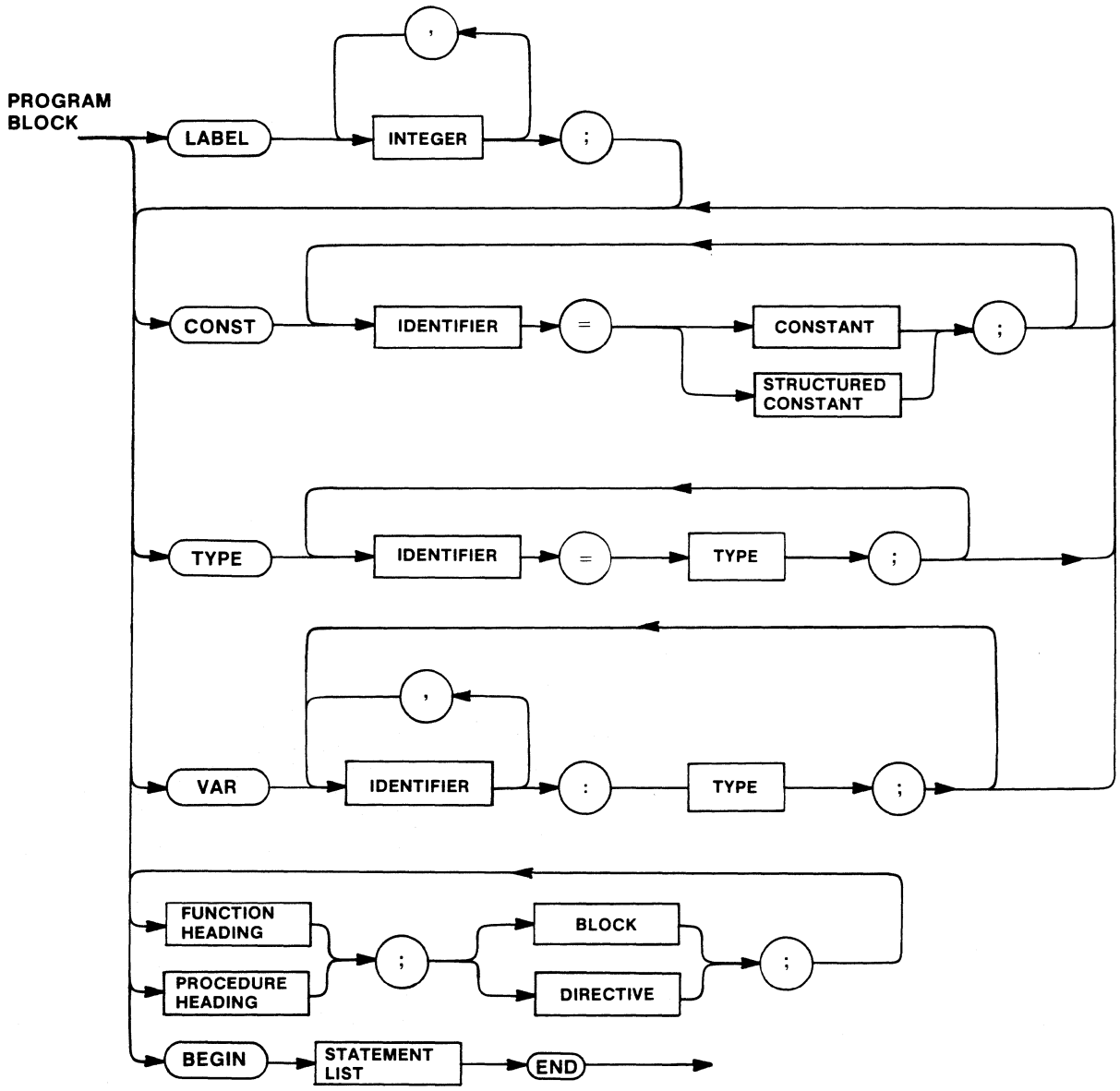

Figure 3-3. Program Block Syntax

Segment

SEGMENT

The compiler option SEGMENT provides a means of dividing a program's p-code into segments or overlays which may be loaded into memory individually. A program that is too large to fit into memory all at once may be divided and thus executed. The SEGMENT option has no value associated with it. It specifies that the next procedure or function encountered by the compiler will become a "segment procedure" or "segment function". The p-code for the segment procedure or function plus the p-code for any procedure or functions contained therein will be put into a separate p-code segment which can be loaded into memory as desired. P-code for the main program body and for any routines which are not part of segment procedure or functions is put into the main program segment which is always resident in memory during execution. When a program is run, the p-code for the main program segment is loaded into memory. The code for other segments is loaded from disc by simply calling the segment procedure or function in the usual way. The p-code for a segment remains in memory as long as the procedure or function is active. After a segment procedure or function returns, the memory containing the p-code for that segment is used for other purposes. Therefore, a subsequent activation of that segment requires that the p-code be loaded again. Recursive calls to segment procedures or functions require only one copy of the segment's p-code to be in memory. The SEGMENT option must be specified before the heading of the desired procedure function. A total of 14 segments may be specified in addition to the main program segment. A segment procedure or function may itself contain up to 255 procedures, functions, array constants, or record constants. A segment procedure or function may itself contain nested segment procedures or functions.

Chapter 4

Pascal Program Declarations

Introduction

Every program consists of a heading and a block. The heading has been discussed in Chapter 3. The block consists of a declaration section, and a statement section that specifies the action to be executed. Items identified in the program declaration are considered to be global in scope.

A complete block contains the following parts:

a. <LABEL declaration>

b. <CONSTant declarations>

c. <TYPE definitions>

d. <VARiable declarations>

e. <PROCEDURE and FUNCTION declarations>

f. <statements>

The declaration syntax is shown in Figure 4-1.

The LABEL declaration, if used, must be placed ahead of the other declarations in the block. The CONSTant, TYPE, VARiable, PROCEDURE, and FUNCTION parts may follow in any order and may be repeated as often as required. A procedure or function may be made into a segment or overlay by using the $SEGMENT$ compiler directive. A total of 14 segment procedures or functions, in addition to the main program segment, may be specified. Each segment may contain up to 255 procedures, functions array constants, or record constants. The repetition of CONST, TYPE and VAR declarations is an HP extension to standard Pascal.

Each part of the declaration section is discussed in detail in the following paragraphs. The statement section is discussed in Chapter 5.

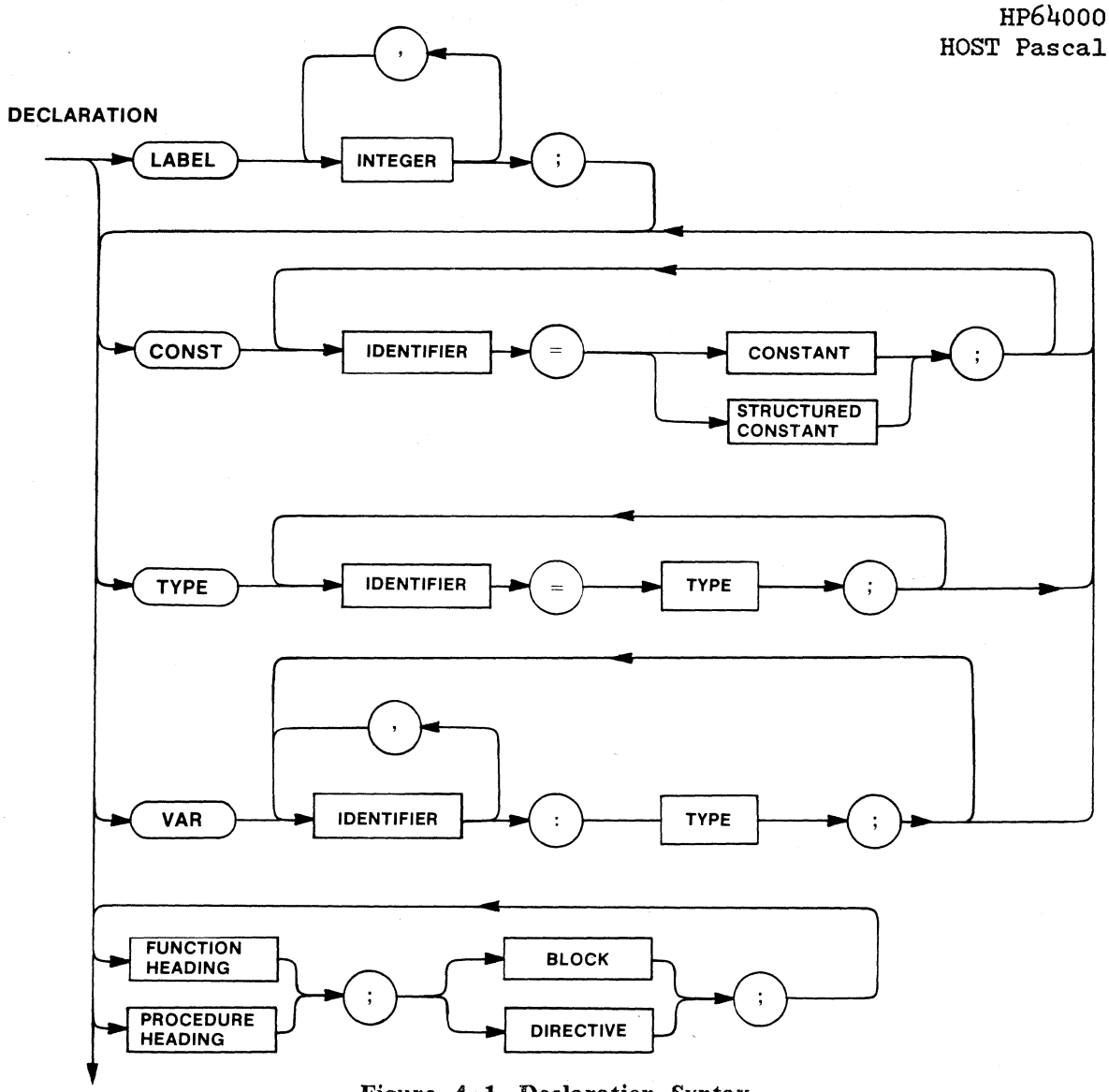

Figure 4-1. Declaration Syntax

Declaration Section

LABEL Declaration

A LABEL is an unsigned integer no more than four digits long (leading zeros are not significant); followed by a colon and a statement. The colon separates the integer from the statement.

Any statement in a program body may be identified by a LABEL. Prior to use, however, the integer must be identified in the LABEL declaration, and in addition, and must be declared first after the program heading.

The function of the LABEL is to transfer program control from one portion to another. The function is not complete, however, without a connecting GOTO <label> statement. Transfer of program control is initiated by the GOTO statement. Figure 4-2 shows the LABEL declaration syntax.

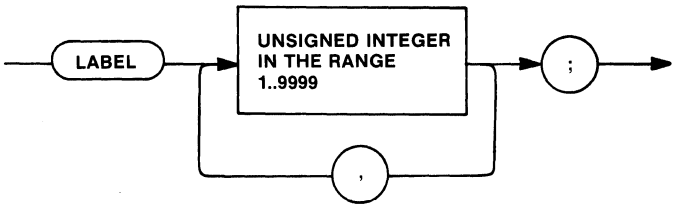

Figure 4-2. LABEL Declaration Syntax

The following example illustrates both LABEL declaration and LABEL use.

Example:

```
PROGRAM showlabel (input,output);

LABEL
    1234;
VAR
    a,b:integer;
BEGIN
          •
          •
          •
IF a > b
    THEN
         GOTO 1234
    ELSE
          •
          •
          •
1234: writeln ('overhead is exceeded');
END.  {showlabel}
```

CONSTant Declaration

CONSTants are identifiers of literal values. The values are assigned to the identifiers at compile time. A constant may denote a value for any data type except a file or a type that contains a file.

Those identifiers, or constants, may then be used throughout the program, making the program more readable for the user.

The literal values cannot be changed during program execution. Literal values can be changed, however, during a program editing session. A literal value assigned to a constant is used whenever the identifier is listed in the program.

If the literal value had been used in the program in place of the identifier, the old value would have to be changed to the new value at

each location in the program. If a constant identifier is used in the program, and not a literal, the only change needed is to the value in the constant declaration. The new value will be used in the program at each location of the identifier.

The syntax diagram for a CONSTant declaration is given in Figure 4-3, and the syntax diagram for literals is expanded in Figures 4-4 thru 4-6.

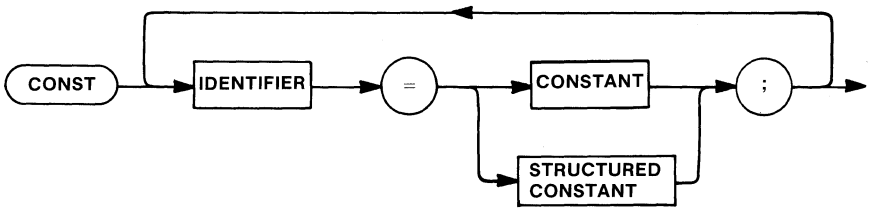

Figure 4-3. CONSTant Declaration Syntax

A string literal consists of a sequence of printable ASCII characters enclosed by apostrophes. If the string contains an apostrophe, the apostrophe must be written twice. The length of the string can range from no characters to a full line of characters. Any string literal must not exceed a line in length. Only printable ASCII characters are allowed between apostrophes. Other characters are denoted by using the # sign outside the apostrophes. The # may be followed by an unsigned integer in the range of 0 thru 255, or by a control character. A control character may be a letter or @, [,], \, ^, _ symbol. If a control character (c) is used, the value produced is: chr(ord(c)MOD 32).

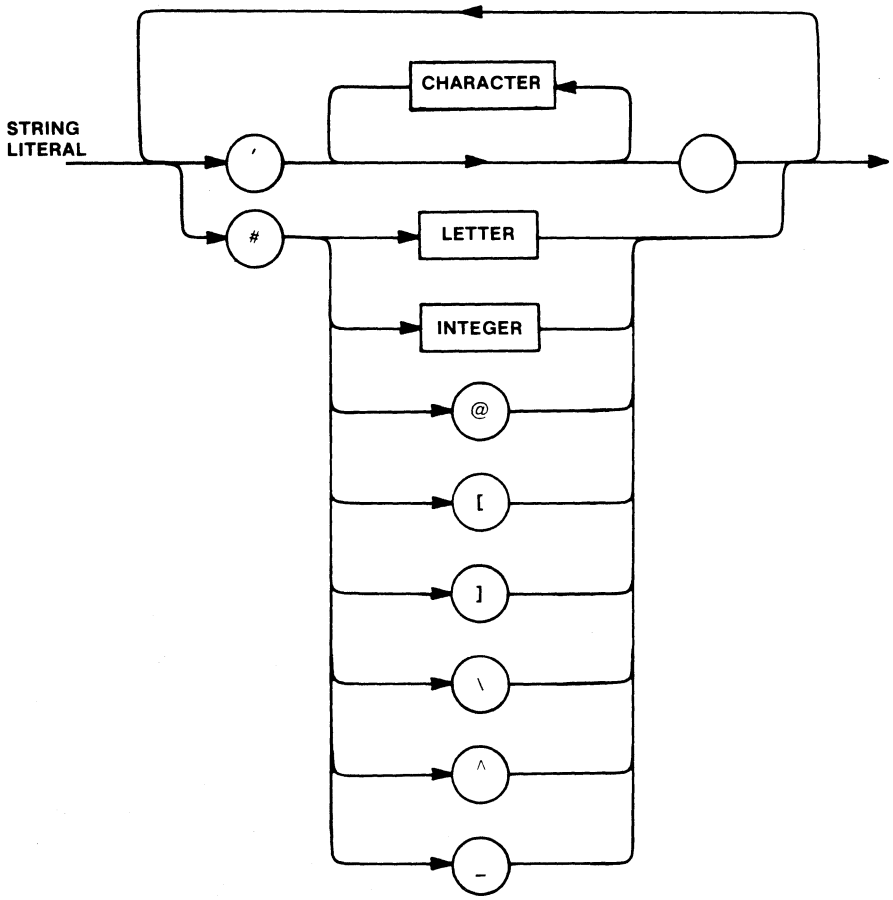

Figure 4-4. String Literal Syntax

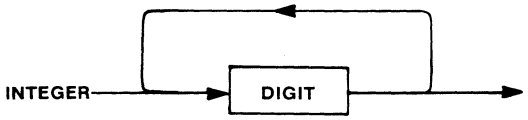

Figure 4-5. Integer Syntax

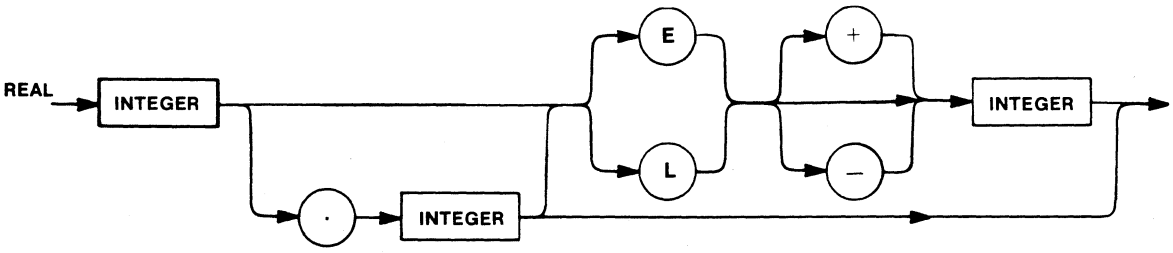

Figure 4-6. Real Syntax

Simple Constants

A simple constant can take the form of a string literal, a previously defined constant identifier, an integer, a REAL optionally preceded by + or -, or an expression of an ordinal type, i.e., INTEGER, BOOLEAN, CHARacter, subrange, or enumerated type. The use of ordinal expressions as constants is an HP extension.

Example:

```
CONST
     pagesize = 55;
     maxpages = 99;
     pi = 3.14159;
     pagenum = maxpages - pagecount;
     heading_a = 'List is now on.';
```

Constant expressions are constructed according to the rules defined for general expressions. The operands in an expression must be either literals or constants that have already been defined. The operators allowed are +, -, *, DIV, MOD, and the predefined functions ord, chr, pred, succ, abs, hex, octal, and binary.

The operands must be ordinal types except that a plus or minus is allowed before a REAL constant. Selection of elements of structured constants as operands is only allowed in executable expressions.

Structured Constants

A structured constant is a constant of a structured type, i.e., array, record, set, or string. Structured constants are an HP extension to Pascal. The definition consists of a previously defined TYPE identifier followed by a list of values. Values for all elements of the structured type must be specified and must have a type that is assignment compatible with the type of the corresponding element; with the exception that a string literal may be used to specify the elements of a PAC or a string data type. Structured constants can be used to initialize variables of structured types. The individual elements of a structured constant are also available as constants. The syntax of the structured constant is shown in Figure 4-7.

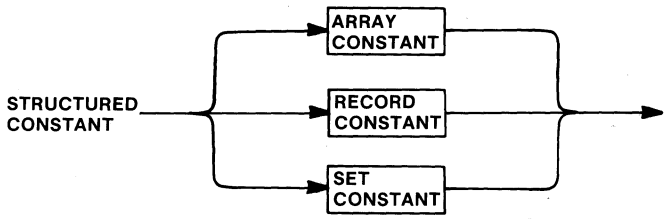

Figure 4-7. Structured Constant Syntax

The following paragraphs describe the structured constants array, record, set, and string.

Array Constants

The definition of an ARRAY constant consists of the ARRAY type identifier followed by the list of values that are to be included in the constant array. The Array constant syntax is shown in Figure 4-8.

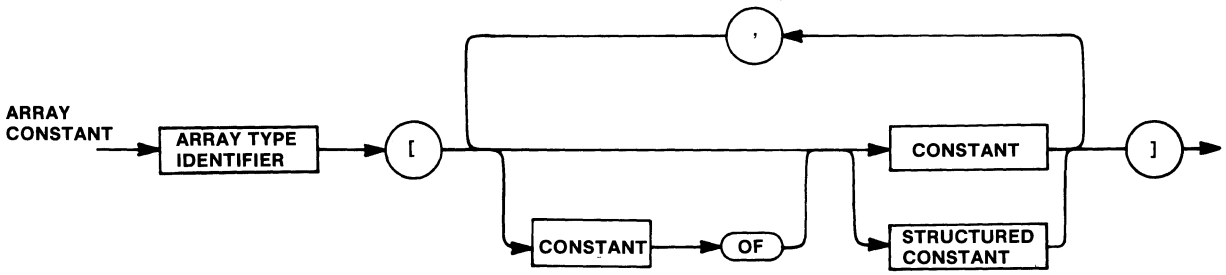

Figure 4-8. Array Constant Syntax

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_VALUE     = BOOLEAN_TABLE
                      [TRUE,TRUE,TRUE,TRUE,TRUE];
     INIT_VALUE1    = TABLE [100 OF 0];
     INIT_VALUE2    = TABLE [60 OF 0, 40 OF 1];
     IDENTITY       = MATRIX [ROW [1,0,0,0,0],
                              ROW [0,1,0,0,0],
                              ROW [0,0,1,0,0],
                              ROW [0,0,0,1,0],
                              ROW [0,0,0,0,1]];
     COLORS         = COLOR_ARRAY
                      [COLOR_STRING ['RED', 3 OF ' '],
                       COLOR_STRING ['YELLOW '],
                       COLOR_STRING ['BLUE', 2 OF ' ']];
```

Notice that in the last example, where the array was a PAC, that a combination of strings and characters was used. This is the only case where the constant (string) is permitted to be of a type that is not assignment compatible with the element type (char).

RECORD Constant

The definition of a RECORD constant consists of the RECORD type identifier followed by a list of the values to be assigned to the fields of the constant record. Each value is preceded by the name of the field that 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. The syntax for a record constant is shown in Figure 4-9.

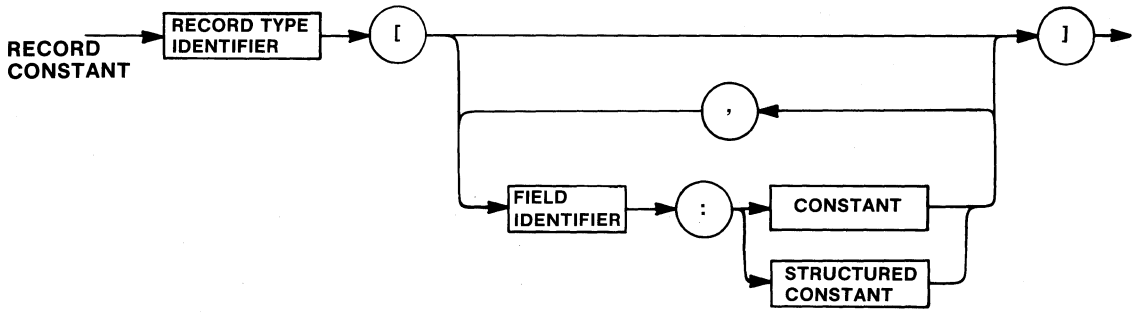

Figure 4-9. RECORD Constant Syntax

Examples:

```
TYPE
     COUNTER_RECORD = RECORD
          PAGES: INTEGER;
          LINES: INTEGER;
          CHARACTERS: INTEGER;
      END;
     REPORT_RECORD  =RECORD
          REVISION: CHAR;
          PRICE: REAL;
          INFO: COUNTER_RECORD;
          CASE SECRET: BOOLEAN OF
               TRUE: (CODE: INTEGER);
     END;

CONST
     NO_COUNT = COUNTER_RECORD [PAGES: 0, CHARACTERS: 0,
     LINES:0];
```

```
BIG_REPORT = REPORT_RECORD
                [REVISION: 'C',
                 PRICE: 27.50,
                 INFO: COUNTER_RECORD
                     [PAGES: 6, LINES: 28, CHARACTERS:
                     496],
                 SECRET: TRUE,
                 CODE: 8128];
```

SET Constant

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

If no type identifier is used, one of three possible results will occur depending on the type T of the elements in the set:

a. If T is an integer, then the set created is of type SET OF 0..255. Compile-time and run-time checks are performed to ensure that specified elements are in this range. Thus the set [25,0,255] is legal, but the set [-10, 256] is not legal.

b. 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'] has the type SET OF CHARacter.

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

The syntax for the SET constant is shown in Figure 4-10.

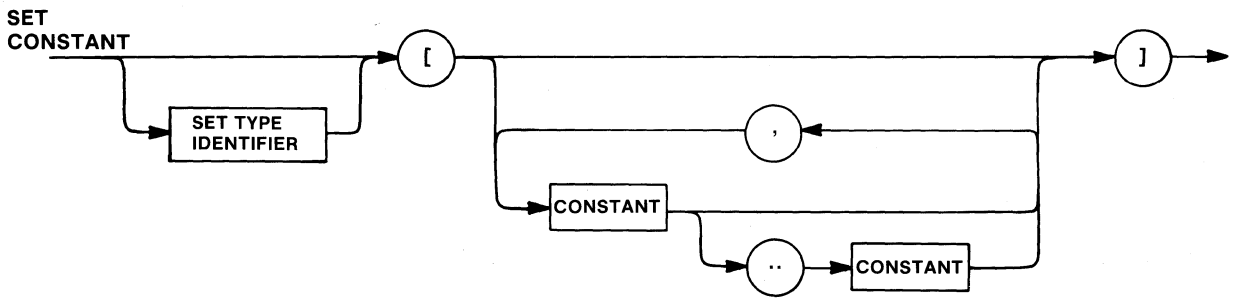

Figure 4-10. SET Constant Syntax

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 Literal

A string literal consists of a sequence of alphabetic or numeric characters in a specific order and enclosed by apostrophes. The string is associated with an identifier, i.e., declared as a CONSTant. Strings can be declared as part of the main program, or can be declared in any routine.

Example:

```
PROGRAM; {or routine}
     CONST
          Heada = 'INVALID INPUT';
                .
                .
                .
     Begin
          IF NOT (INPUT = 5)
               THEN writeln (heada)
     End.  {program}
```

TYPE Definitions

Types defined in this section of Chapter Four are the Predefined Types, Structured Types and the type modifier Packed.

TYPE

Data items can be characterized by their type. The TYPE determines a set of attributes:

a. the set of permissable operations that may be performed on an object of that type.

b. the set of values that may be assumed by an object of that type.

c. the amount of storage required by objects of that type.

Certain types are predefined. Other types can be defined by the user.

Type Declaration

In the type declaration section, an identifier can be associated with a type definition.

The TYPE declaration syntax is shown in Figure 4-11.

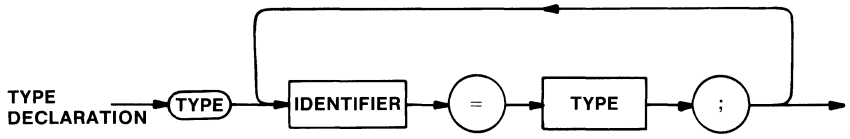

Figure 4-11. TYPE Declaration Syntax

Types can be further defined as:

Simple Types
Pointer Types
Structured Types

The TYPE syntax is shown in Figure 4-12.

Simple Types

All simple types define an ordered set of values. Simple types are divided into ordinal types and real types.

Ordinal Types

Ordinal types are the predefined ordinal types: BOOLEAN, CHAR, and INTEGER; the user defined ordinal types: enumerated and subrange, and type identifiers that have been equated to another ordinal type.

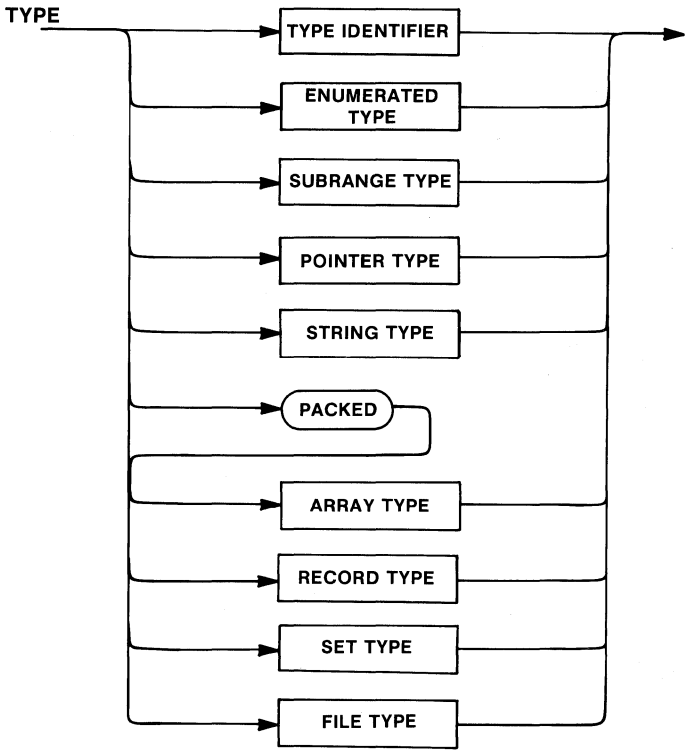

Figure 4-12. TYPE Syntax

Predefined Ordinal Types

BOOLEAN

The type Boolean is an ordinal type having two elements, false and true, and occupies one word of memory. Implicit is the concept that false < true. The operators applicable to Boolean operands are NOT, AND, OR. NOT takes precedence over AND; AND takes precedence over OR. The order of precedence may be altered by the use of parentheses. The relational operators always yield Boolean values.

CHAR

The type CHAR is an ordinal type consisting of the ASCII character set whose ordinality range is 0..255, and occupies one word in memory. The functions applicable to the type CHAR are ORD(C) which yields the ordinal integer corresponding to the character (C); and CHR(I) which yields the character corresponding to the ordinal value (I).

INTEGER

The type Integer is an ordinal subset of the set of whole numbers. The range of Integers is predefined by the terms MININT..MAXINT. MININT..MAXINT is the range -2^{31} to $2^{31}-1$. Each integer occupies two words in memory.

User Defined Ordinal Types

The user defined types discussed in the following paragraphs are the enumerated type and subrange type.

Enumerated Type

Enumeration defines an ordered set of values by listing the identifiers of the ordered values. The identifiers are constants that have ordinal values beginning with 0 for the first identifier, 1 for the second identifier, and so forth. The syntax diagram for the enumerated type is shown in Figure 4-13.

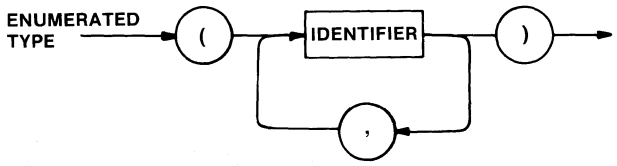

Figure 4-13. Enumerated Type Syntax

Examples:

```
color = (black, brown, red, orange);
day = (sunday, monday, tuesday);
```

The ordinal value of black is 0. The ordinal value of orange is 3. The ordinal value of monday is 1.

Subrange Type

Definition of a subrange (of an ordinal type) requires listing the lower bound constant and the upper bound constant of the subrange. In the case of a subrange of type INTEGER, the bounds must be between -2^{31} ..$2^{31}-1$. A variable of the subrange type possesses all of the properties of a variable of the base type, with the only restriction being that its value be in the specified range. The syntax for the subrange type is shown in Figure 4-14.

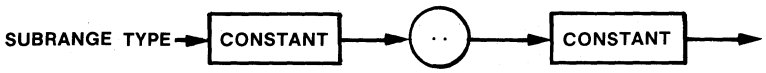

Figure 4-14. Subrange Type Syntax

Examples:

```
TYPE

   Dip = 1..99;
   Alpha = 'A'..'K';
```

Two words of memory will be occupied by a subrange of INTEGER if either the lower bound or the upper bound falls outside of the range -32768..32767. Only one word of memory will be occupied if both upper and lower bounds of the subrange fall within the range -32768..32767.

Examples:

```
Flop = 3900..39000; {will occupy two words of memory.}
Flip = -39000..-3900; {will occupy two words of memory.}
Floor = -99..99; {will occupy one word of memory.}
```

Real Types

Real types are the predefined types REAL, LONGREAL, and identifiers that have been equated to real types.

REAL

The set of Real numbers is a subset of whole numbers and is not an ordinal type. The Real number range includes values between +/- 10^{37} with six significant decimal digits. Each real number occupies two words in memory.

LONGREAL

The set of Longreal numbers is a subset of Real numbers. Each longreal number occupies four words in memory. The precision of Longreal is greater than that of Real. The values of Longreal range between +/- 10^{308} with 15 significant digits. Any of the set of operators applicable to Real numbers are also applicable to Longreal numbers.

Pointer Types

Variables that are declared in a program are accessible by their identifiers. These variables exist during the entire execution of the level of program to which they are local, and are therefore called static variables.

Dynamic variables can be generated without any correlation to program structure by using the standard procedure NEW(p). New memory space is allocated for the new dynamic variable, and the pointer variable (p) holds the address of the new dynamic variable. Thus a pointer variable may "point" to a dynamic variable. See Figure 4-15 for the pointer type syntax diagram.

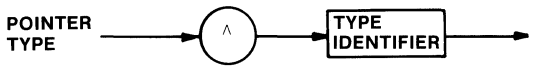

Figure 4-15. Pointer Type Syntax

A pointer may only refer to dynamic variables of a single type called the "base type". The base type is specified in the pointer definition. A pointer variable may be assigned the value NIL. NIL points to no location in memory. NIL is a reserved word used in a pointer at the end of a linked data structure to indicate the end of the data structure. Pointers occupy one word of memory space.

The base type identifier is an exception to the rule that all identifiers must be declared before they are used.

HOST Pascal does not allow dynamic variables to be type FILE or a type that contains a file.

Examples:

```
father, mother, child, sibling: ^person;
carbon, film, wirewound: ^resistor;
```

Structured Types

The structured types Array, Record, Set, and File are characterized by component type and by the structuring method. A structured type definition may contain an indication of the preferred data representation by use of the term PACKED. The term PACKED is an indication to the compiler that data storage is to be economized.

Array

An array is made up of a fixed number of components, each of which can be directly accessed. Each array has an index by which a component is

selected from the array. The index must be an ordinal type, e.g., [1..6]. The number of elements in the array is specified by the index. The components can be any type; but all components are of the same type, called the base type. It is illegal to use the form [INTEGER] as an index, even though INTEGER is an ordinal type. The syntax of the Array is shown in Figure 4-16.

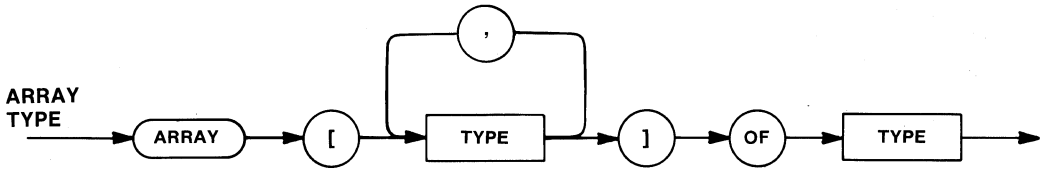

Figure 4-16. Array Syntax

Examples:

```
TYPE
     Root = Array ['1'..'6'] of Real;
     Freq = Packed Array ['1'..'6'] of Real;

CONST
     Flip = Root [1.1, 1.2, 1.3, 1.4, 1.5, 1.6];
     Flop = Freq [2.1, 2.2, 2.3, 2.4, 2.5, 2.6];
```

Components of an array can be assigned values during execution of the main program block as well as in the CONST declaration. The use of a packed array involves a choice between ease of access/execution time and conservation of memory space. A packed format conserves memory space but increases access/execution time because each access of the packed array requires a corresponding UNPACK effort.

Multi-dimensioned arrays, i.e., arrays of arrays, are possible by use of the following format:

```
TYPE
     row = ARRAY [1..5] OF real;
     matrix = ARRAY [1..10] OF row;
```

or shortened to the equivalent form:

```
TYPE
     matrix = ARRAY [1..5] OF ARRAY [1..10] OF real;
```

or reduced further to the form:

```
TYPE
     matrix = ARRAY [1..5, 1..10] OF real;
```

Bear in mind that each dimension of an array can accommodate no more than 16,383 elements. Multi-dimension arrays can, however, by their nature, accommodate more than 16,383 elements.

PAC

The term PAC is used to refer to arrays of the type:

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

PAC data types have the special property that they are compatible with string literals with the same or fewer components than are in the PAC data type.

When a string literal with fewer characters is assigned to a PAC variable, it is automatically extended on the right with blanks so that it has the same number of components as the PAC. When a string literal with fewer components is compared to a PAC data type, it is automatically extended on the right with blanks before the comparison is made. The use of PACs with shorter string literals is an HP extension to standard Pascal.

String Data Types

Strings are a family of standard data types that are similar to packed arrays of character, but have special properties. String data types are an HP extension to standard Pascal.

String data types have a dynamically variable length. While the maximum length of a string is fixed in the type definition, the actual length, i.e., the number of valid characters presently in the string, may vary from 0 to the maximum declared length of the string data type.

The predefined function STRLEN may be used to determine the present length of any string expression.

The components of a string data type are of type CHAR and may be accessed in the same way as components of other arrays. There is the additional restriction that the value of the index expression used to select a character must be greater than or equal to 1, and less than or equal to the current length of the string.

A string data type is compatible with any other string data type. In addition, string data types are compatible with string literals. The string literal '' represents the "null" string that is a string consisting of no characters.

The syntax for defining string types is different from other arrays. See Figure 4-17 for the string type syntax.

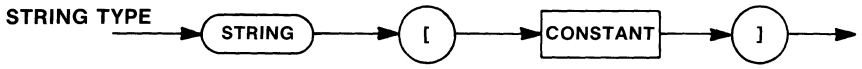

Figure 4-17. String Type Syntax

The constant is a constant integer expression specifying the maximum number of characters that may be contained in the string. In HOST Pascal the value of this constant must be between 1 and 255.

String variables may be used as VAR parameters in procedures and functions in a way that is fundamentally different from other data types.

Ordinarily the actual parameter substituted for a formal VAR parameter in a procedure or function must have the identical type as the formal parameter. With strings, however, the following type of parameter declaration is allowed:

```
PROCEDURE P (VAR  S:  STRING);
```

In this case, where the parameter type identifier is the predefined identifier STRING, any string variable may be substituted for S.

String expressions may be concatenated using the "+" operator and compared to other strings and string literals. Refer to Chapter 5 for information on string comparison.

When a string variable is created at program execution time, its value, including the dynamic length, is unpredictable. It is important for consistent execution to assign a value to the dynamic length before accessing the individual characters of the string variable. For example, the following will sometimes produce a run-time error:

```
VAR     S : STRING [100];
        I : INTEGER;
BEGIN
FOR I := 1 TO 100 DO   {Sometimes causes run-time error    }
 S[I] := ' ';          {because STRLEN(S) is unpredictable.}
```

The length of a string variable is set in only one of three ways.

1. Assign to the entire string variable. For example,

   ```
   S := 'ABC';
   ```

2. Use the procedure SETSTRLEN. For example,

   ```
   SETSTRLEN (S,100);
   ```

3. Use the variable as an actual variable parameter to a routine which assigns a value to it. For example,

   ```
   READ (S);
   ```

The example can be rewritten as follows:

```
VAR     S : STRING[100];
        I : INTEGER;
BEGIN
SETSTRLEN (S,100);
FOR I := 1 TO 100 DO
    S[I] := ' ';        {operates correctly}
```

RECORD

RECORD is a Pascal reserved word signifying a structured data type having a fixed number of elements. These elements, called fields, can be of differing types. The fields are enumerated and their types defined in the record TYPE declaration. Different records may have fields of the same name, but fields within a record must have distinct names. The field list follows each RECORD identifier. Each RECORD declaration is completed by END;. The syntax of a Record constant is shown in Figure 4-10.

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

The syntax for a record type is shown in Figure 4-18, and the syntax for a field list is shown in Figure 4-19.

Figure 4-18. Record Type Syntax

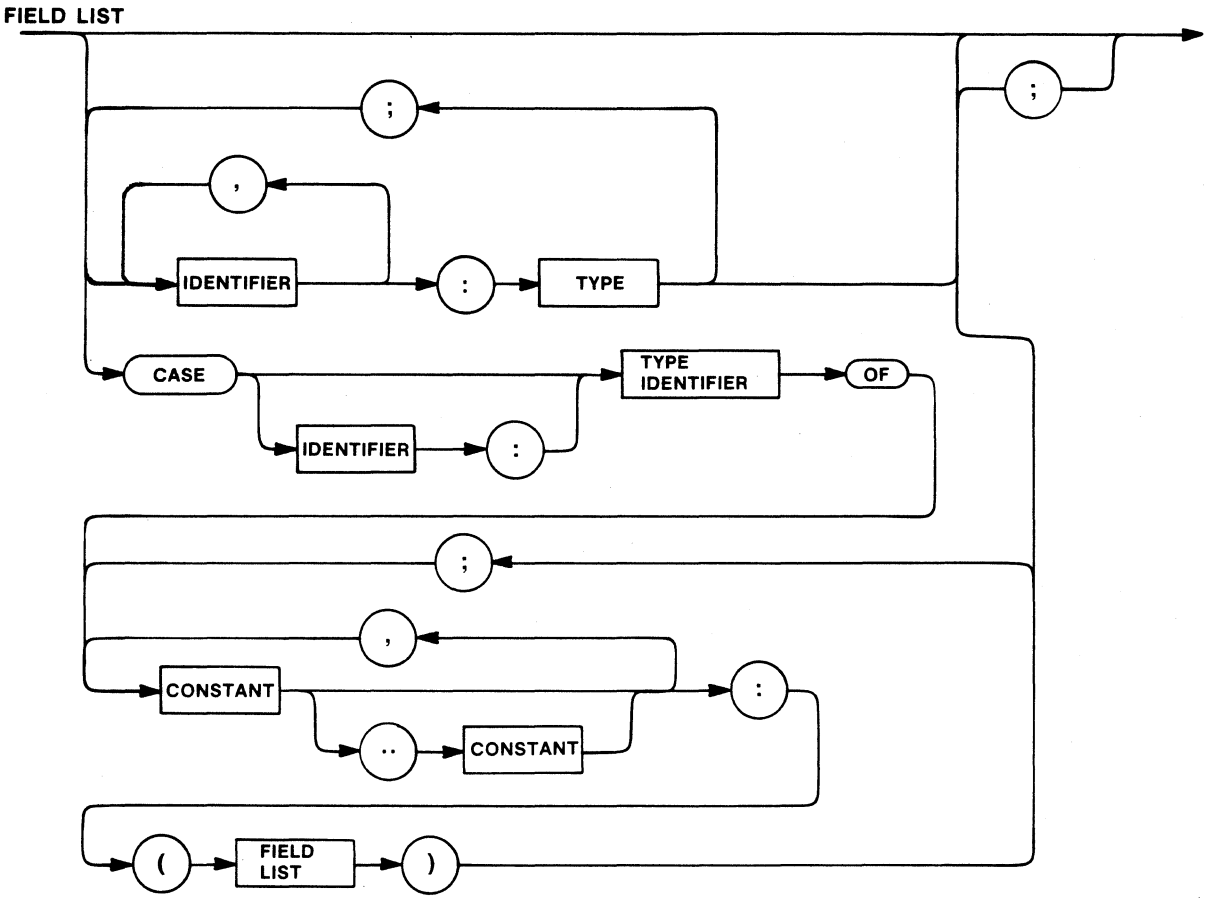

Figure 4-19. Field List Syntax

SET

A set is the powerset (set of all subsets) of an ordinal type called the base type. The base type may be identified by a predefined type, i.e., char or integer, or a subrange of a predefined type. The syntax of a Set type is shown in Figure 4-20.

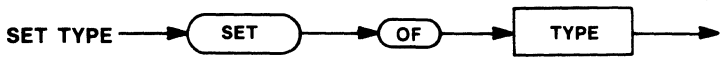

Figure 4-20. Set Type Syntax

The set base type must be an ordinal type. In the case of a subrange of integers, the low bound must be >= to 0 and the high bound must be <= 4079.

Relational operators for sets include =, >=, <=, and <>. These operators can be used between sets, with results that are Boolean. The symbol IN may be used between an ordinal expression and a simple set expression.

Examples:

```
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;
```

Sets can be manipulated by set union (+), set difference (-), and set intersection (*) to define new element groups. The maximum number of elements in a set is limited to 4080.

FILE

The file definition specifies a structure consisting of a sequence of components that are all of the same type. The number of components is not fixed by the file type definition. A file is usually associated with a peripheral storage device. A file having no components is an EMPTY file. The component of a file may not be a file or a structured type containing a file. The component type of a file must be assignable. The file type syntax is shown in Figure 4-21.

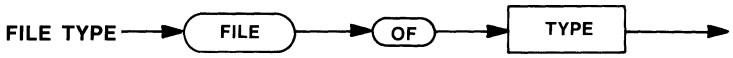

Figure 4-21. File Type Syntax

PACKED Type Modifier

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

There are times, however, when the programmer needs smaller amounts of storage allocated to certain data items, even if this requires less efficient access. The programmer can indicate this to the compiler by prefixing the definition of a structured type with the symbol PACKED. The syntax of the PACKED modifier is shown in Figure 4-22.

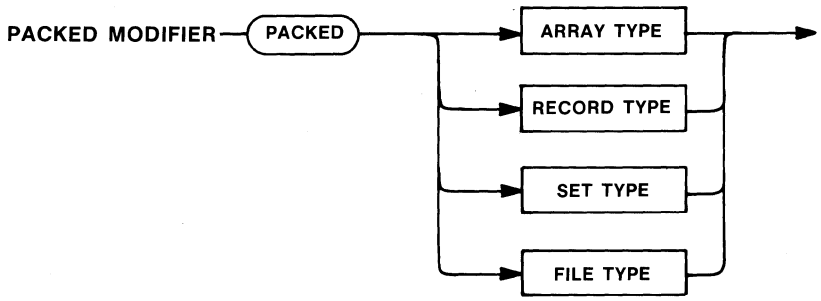

Figure 4-22. PACKED Modifier Syntax

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

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

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

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

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

The standard procedures PACK and UNPACK can be used to assign components from an unpacked array to a packed array, and vice versa.

VARiable

VARiables are locations in memory that are identified by name, and exist during the entire execution of the level of program to which they are local. Variables that have been declared are called static variables.

Dynamic variables, on the other hand, can be generated without any correlation to program structure. Variables contain values that can be changed during the execution of the program. The VARiable declaration syntax is shown in Figure 4-23.

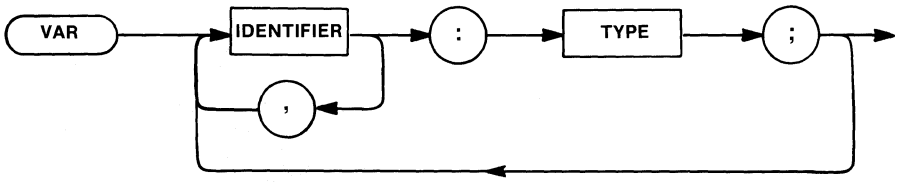

Figure 4-23. VARiable Declaration Syntax

VARiable Declaration

The variable declaration consists of a list of identifiers followed by the applicable type definition.

Examples:

```
VAR

     aset : char;
     bset, dset, flop : integer;
     freq : PACKED ARRAY [1..15] of REAL;
     root : ARRAY [(alpha, beta)] of COLOR;
     cset : FILE OF char;
     nset : SET OF noun;
     p1, p2 : ^person;
```

The value of a variable is undefined at the time of declaration.

Variables denoted are either entire variables, component variables, or variables referenced by a pointer.

Entire Variables

An entire variable is denoted by its identifier.

Component Variables

A component variable is denoted by the variable followed by a selector specifying the component. The form of the selector depends on the structure type of the variable. The selector can be an index, a field designator, or a buffer variable.

Indexed Variables

A component of an n-dimensional array variable is denoted by the variable identifier followed by n index expressions. The index expressions must be assignment compatible with the index types declared in the definition of the array type. Indices separated by commas are equivalent to indices in separate brackets, and combinations of commas and brackets. For example, [a,c,e] is equivalent to [a,c] [e], equivalent to [a] [c,e], and equivalent to [a] [c] [e].

Field Designators

A component of a record variable is denoted by the record variable followed by a period and the field identifier of the component.

Examples:

```
a.color
b[red,true].nset
```

Buffer Variables

A buffer variable is associated with each file. The current file component may be read into this buffer variable and inspected prior to a READ of the component. The next file component may be placed in the buffer variable and written from that location.

Example:

```
input^
```

Referenced Variables

A referenced variable is a pointer expression followed by the ^ symbol. If p is a pointer variable with base type T, then p^ denotes a variable of type T.

Examples:

```
p^ p^.sibling^
```

Routine Declarations

PROCEDURE and FUNCTION declarations may take place in the declaration portion of the main program, or within other procedures or functions. Routines must be declared before they are used. Each routine, whether procedure or function, is declared in a similar fashion. The routine heading is followed by either the directive FORWARD or by a block that contains the declarations and statements comprising the routine. The routine declaration syntax is shown in Figure 4-24.

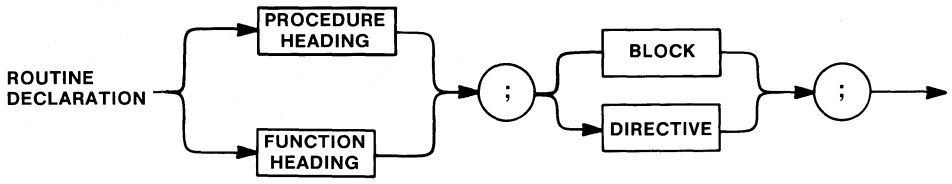

Figure 4-24. Routine Declaration Syntax

PROCEDURE Declaration

Procedures perform specific tasks or algorithms by execution of the statements within the procedure. Each procedure must be declared before its use. The procedure heading syntax is shown in Figure 4-25.

Figure 4-25. PROCEDURE Heading Syntax

Example:

```
PROGRAM REG21 (INPUT,OUTPUT);
    VAR  I,N  : INTEGER;
         X,Y  : REAL;
    PROCEDURE SWAP (VAR P,Q :REAL);
    VAR  TEMP : REAL;
    BEGIN
         TEMP := P;
         P    := Q;
         Q    := TEMP
    END;
BEGIN
    READ(N);
    FOR I := 1 TO N DO
         BEGIN
              READ (X,Y);
              IF X > Y THEN SWAP (X,Y)
         END;
    WRITELN ('ARE THE ORDERED PAIRS');
END.
```

FUNCTION Declaration

The function declaration consists of a heading and a main block. The function heading consists of the function identifier, a formal parameter list, and the type of function result. The type of the function result can be any type except a file, or a type containing a file. Within the function main block at least one statement must assign a value to the function identifier. See Figure 4-26 for the FUNCTION heading syntax.

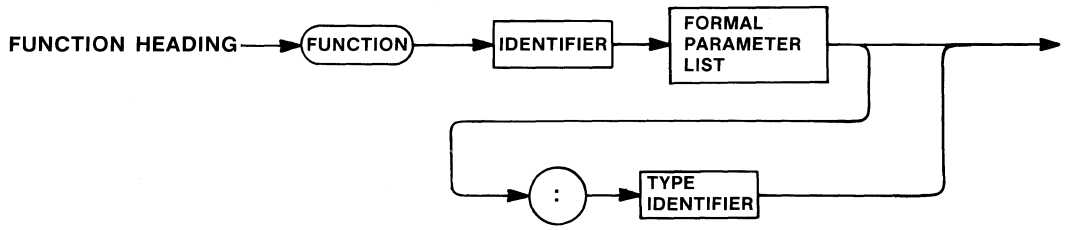

Figure 4-26. FUNCTION Heading Syntax

Functions perform specific tasks or algorithms by execution of statements within the FUNCTION. The function is further identified by a type; and the value generated by the function must be of that type, and assignable to the function identifier.

Example:

```
     {function declaration}
FUNCTION Sqrt (x:real):real;
VAR X0, X1:real;
BEGIN X1 :=X; {X >1, Newton's method}
    REPEAT X0 := X1; X1 := (X / X0 + X0) * 0.5
    UNTIL abs (x1-x0) < eps * x1;
    Sqrt := x0
END;

          {function call}
     BEGIN  {start of program}
            •
            •
            •
        writeln ('The following is a square root matrix.');
        y := sqrt (3.5);
        writeln ('End of square root matrix.');
            •
            •
            •
     END.  {end of program}
```

Parameter Lists

Formal Parameter List

The formal parameters for both functions and procedures can be value parameters, variable parameters, procedure parameters, and function parameters. The syntax diagram of the formal parameter list is shown in Figure 4-27.

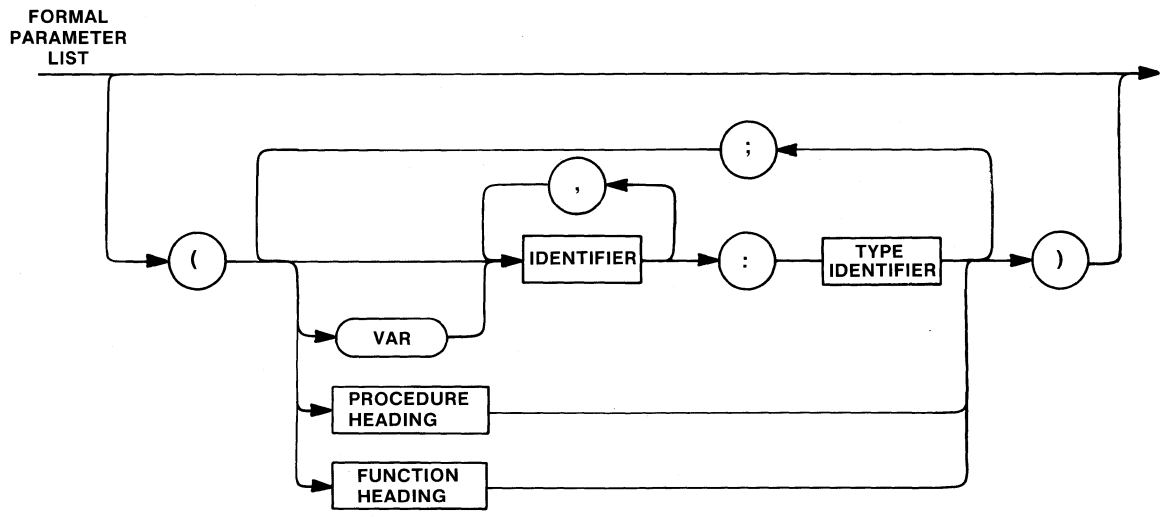

Figure 4-27. Formal Parameter List Syntax

Actual Parameter List

Actual parameters are the values used in the execution of procedures and functions. The syntax diagram of the actual parameter list is shown in Figure 4-28.

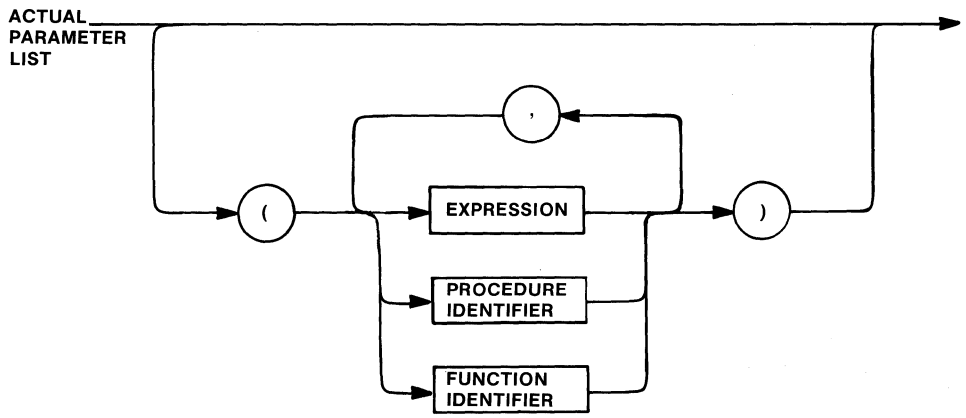

Figure 4-28. Actual Parameter List Syntax

Value Parameter

The actual parameter must be an expression, i.e., something found on the right side of an assignment statement. The corresponding formal parameter represents a local variable in the called routine, and the current value of the expression is initially assigned to this variable. Actual value parameters must be assignment compatible with the type of the corresponding formal parameter.

Variable Parameter

The actual parameter must be a variable, and the corresponding formal parameter represents the actual variable during the entire execution of the routine.

An actual variable parameter must have the same type as the corresponding formal parameter.

There is one exception to this rule of identical types. The type of a variable parameter may be the predefined type STRING. In this case, a variable of any string data type may be the actual parameter.

Procedure Parameter

The formal parameter is a procedure heading. The actual parameter is a procedure identifier. The actual procedure and formal procedure have compatible formal parameter lists.

Predefined procedures may not be used as actual procedure parameters.

Function Parameter

The formal parameter is a function heading. The actual parameter is a function identifier. The formal function and actual function have compatible formal parameter lists and identical result types.

Predefined functions may not be used as actual function parameters.

Parameter List Compatibility

Two formal parameter lists are compatible if they contain the same number of parameters, and if the corresponding parameters match. Corresponding parameters match if any of the following is true:

a. They are both value parameters of the same type.

b. They are both variable parameters of the same type.

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

d. They are both function parameters with compatible parameter lists and the same result type.

Declarations Within Routines

The declaration part of a procedure or function contains the declarations of constants, types, labels, variables, and other routines. These declarations are local to the routine in which they are declared. Declarations within routines take the same form as the program declaration.

Routine Body

The body of a routine is a compound statement that describes the execution of the routine toward an end result. The result of a function is a value. The result of a procedure is an action. The syntax for the routine block is the same as that of a main program block.

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 FORWARD must be used to inform the compiler of the location of the block. A FORWARD declaration is composed of the routine heading, including the parameter list if used, the function result type if applicable, followed by the directive. The routine must be fully declared before the end of the current scope. The parameter list, and result type for a FUNCTION, may be respecified. If the parameter list and function type is respecified, they must be identical with the original declaration.

Example:

```
FUNCTION exclusive_or (x,y:boolean) : boolean;
     FORWARD;
          .
          .
          .
     FUNCTION exclusive_or;
          BEGIN
               exclusive_or := (x and not y) or (not x and y)
          END;
```

Recursive Routines

A routine that calls itself is a recursive routine. Use of the routine identifier within the routine body indicates recursive execution of the routine. If a FUNCTION identifier appears on the left of an assignment statement, however, only the assignment is executed. It is also possible for a first routine to call a second routine in which the first routine is called. That action is an indirect recursion.

Scope

Certain objects in HOST Pascal programming have a related scope of utility. Those objects are:

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

The scope of an object pertains to the level of the program in which the object is declared or defined. Within a routine declaration the declaration part specifies local labels, constants, types, variables, and routines. Execution of the routine may access labels, variables, constants, types, and parameters declared at the same or outer levels of declaration. No outer level program execution, however, can access an

inner level identifier. In the case of two identifiers having different scopes but having the same spelling, the outer identifier will be inaccessible to the inner identifier. No two identifiers having the same scope can have the same spelling.

Statements

A compound statement is a statement list consisting of other compound statements and simple statements. Each statement must be separated from other statements by a semicolon (;). The statement list syntax is shown in Figure 4-29. Refer to Chapter 5 for more information on compound statements.

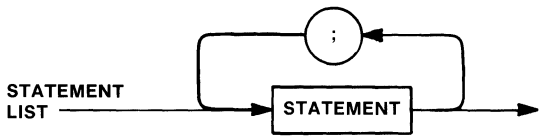

Figure 4-29. Statement List Syntax

Chapter 5

Statements and Expressions

Introduction

The body of a program, procedure, or function consists of a compound statement, meaning the BEGIN/END delimiters enclose the program or routine body. Following is the type and general function of HOST PASCAL/64000 statements:

Label	-	statement identifier.
Assignment statement	-	assigns a value to a variable.
Procedure statement	-	calls a procedure.
Conditional statements	-	IF and CASE statements choose a set of actions based on a condition.
Repeat statements	-	WHILE, REPEAT, and FOR repeat a set of actions.
Field dependent statement	-	WITH statement allows a reference to a record field without naming the record.
Control transfer statement	-	GOTO statement transfers action to another part of the program.
Grouping statement	-	Compound statement is a group of statements.
No Op statement	-	Empty statement is a do nothing statement.

The assignment, PROCEDURE, GOTO, and empty statements are commonly called "simple" statements; the IF, CASE, WHILE, REPEAT, FOR, and WITH statements are commonly called structured statements for they may contain other structured statements, simple statements, and statement labels. The syntax for statements is shown in Figure 5-1.

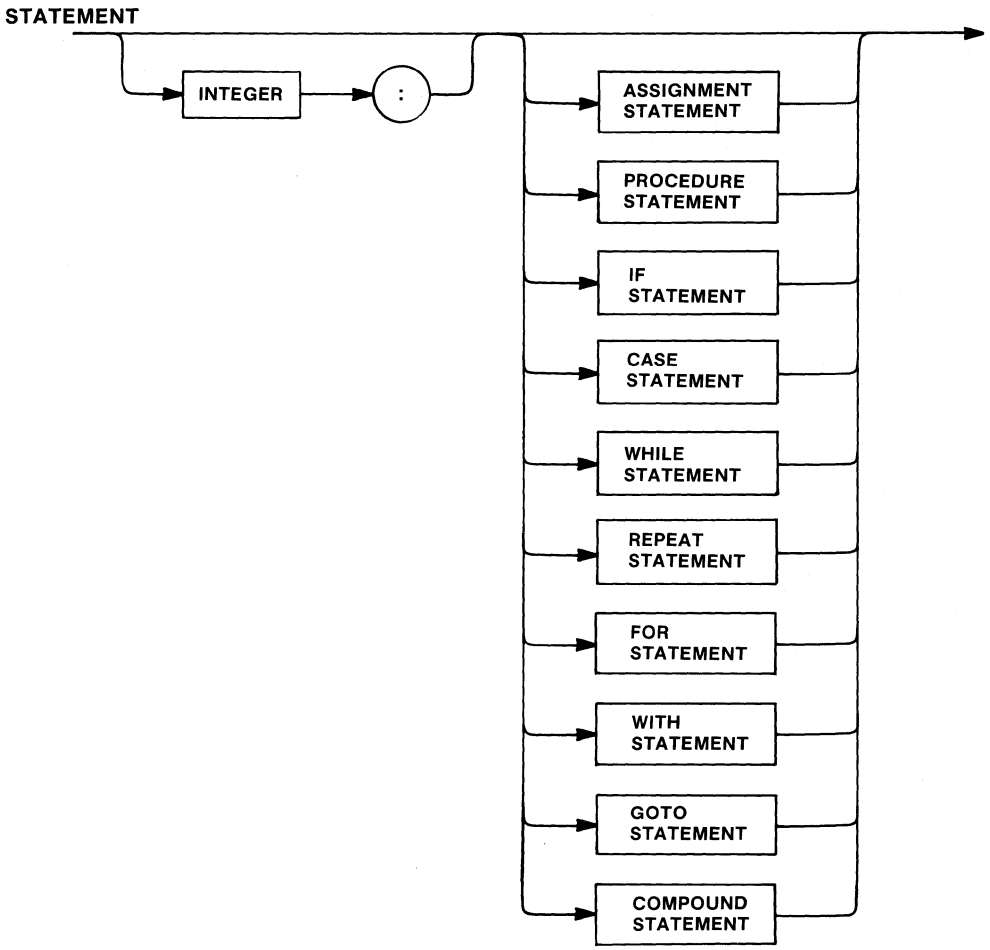

Figure 5-1. Statement Syntax

Statement Label

A statement label may be associated with any statement in a program body. The label must have appeared in the LABEL declaration section of the program or routine in which the label is defined.

Assignment Statement

The assignment statement is used to change the value of a variable. The variable can be of any type except a file type, or a structure containing a file. The type of the variable and the type of the expression must be assignment compatible; e.g., a variable of type INTEGER cannot be assigned a value of type CHARacter. The identifier on the left side of the assignment symbol may be either a variable identifier, a field identifier, or a function identifier. If the identifier is a FUNCTION

identifier, the assignment statement must be made within the block of the FUNCTION. The assignment statement syntax is shown in Figure 5-2.

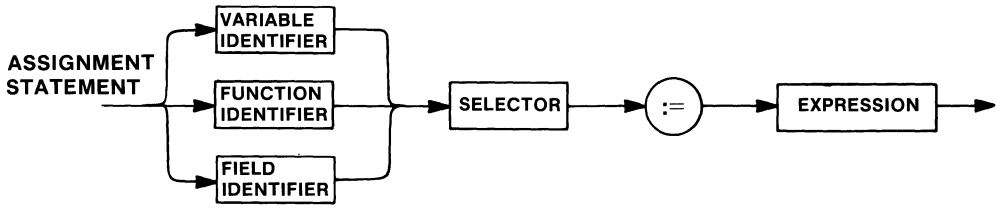

Figure 5-2. Assignment Statement Syntax

Example:

```
fctr := 31.25;
flop := flim * fctr;
```

Procedure Statement

The procedure statement transfers program execution to a procedure. Upon completion of the procedure, program execution is transferred to the statement that follows the procedure statement. The procedure identifier must be the name of either a predefined procedure or a procedure declared previously in a procedure declaration. The declaration may have been an actual declaration (i.e., heading and body), a forward 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 have the actual parameters. The actual parameter list must agree in number, order, and type with the formal parameter list. The procedure statement is illustrated in the following examples, and the syntax is shown in Figure 5-3.

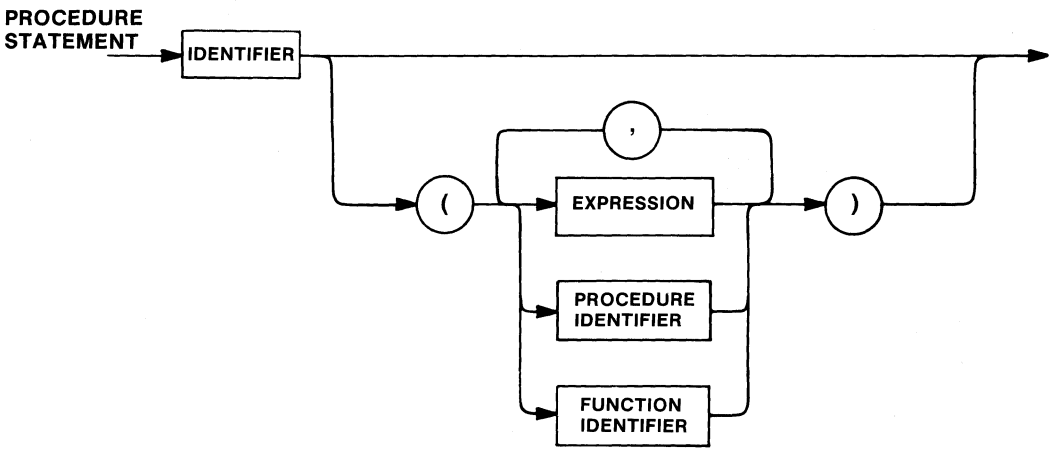

Figure 5-3. Procedure Statement Syntax

Example:

```
PROCEDURE freqgen (VAR fctr,rctr:integer); {procedure
                                            declaration}
     BEGIN
          •
          •
          •
     END;   {procedure declaration}

BEGIN  {program}
     •
     •
     •
     freqgen (apron,ramp);
     •
     •
     •
END.   {program}
```

Compound Statement

The compound statement is used as a means of treating a group of statements as a single statement. The compound statement is delimited by the reserved words BEGIN and END. The statements enclosed by BEGIN and END are executed in the order written. The compound statement has two primary uses:

a. as the body of a procedure, function, or program;

b. as a structured statement that may contain other statements. Usually where a substatement is allowed, the default is only one statement. The compound statement is useful if more than one statement is to be executed.

Compound statements can be used as part of IF, CASE, WHILE, REPEAT, FOR, and WITH statements. Delimiters are required with each of the compound statements. The BEGIN/END pair is used in all cases except REPEAT and CASE statements.

REPEAT/UNTIL delimit the REPEAT statement, and CASE/END delimit the CASE statement. The compound statement syntax is shown in Figure 5-4.

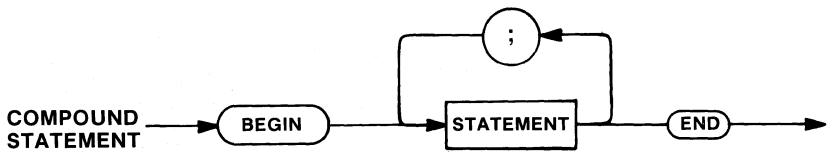

Figure 5-4. Compound Statement Syntax

IF Statement

The IF statement chooses one of two possible responses, based on a given condition. The two responses possible are THEN and ELSE. The expression that follows IF must be a boolean type. When the IF statement is executed, the expression is evaluated to be either TRUE or FALSE. If the value is true, the action following THEN is performed. If the value is false, the action following ELSE is performed. If the value is false and no ELSE action is specified, no action is taken. By implication, however, the remainder of the program becomes the ELSE action. ELSE parts that appear to belong to more than one IF statement are always associated with the nearest IF statement. Note that a semicolon may not separate a THEN statement from the related ELSE statement. The IF statement syntax is illustrated in Figure 5-5.

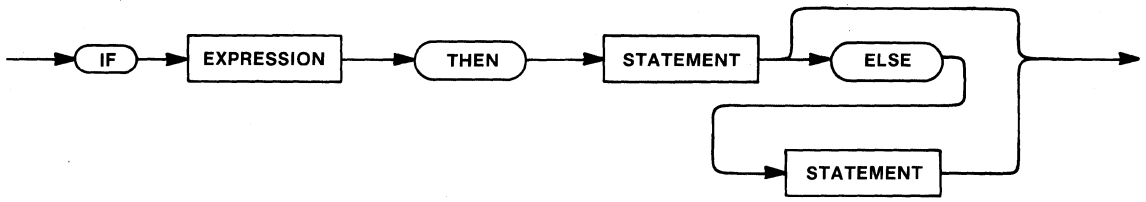

Figure 5-5. IF Statement Syntax

CASE Statement

The CASE statement, like the IF statement, is used to select a certain action based upon the value of an expression. The CASE statement, however, can select from more than two courses of action. If none of those courses of action are selected an OTHERWISE statement is executed. The OTHERWISE portion of the CASE statement is an HP extension of standard Pascal. CASE statement syntax is illustrated in Figure 5-6. The CASE expression may be any ordinal type, including boolean, integer, character, and user-defined enumeration and subrange types. The expression, called the selector, is used to choose which statement is to be executed. Each constant expression in the list 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 part is executed if the selector does not match any of the labels. Specifically, when a CASE statement is executed:

a. The selector expression is evaluated.

b. If the value appears in a label list within the CASE statement, the statement associated with that label is executed and main program execution continues with the statement following the CASE statement.

c. If the value does not appear in any label list the statements appearing between OTHERWISE and END are executed, and program execution resumes with the statement following the CASE statement.

d. If the value does not appear in any label list and no OTHERWISE clause exists, the result will be a run time error.

CASE statements may be nested to any level.

Because of the way the CASE statement is implemented in HOST Pascal, it is possible to generate a large amount of object code with a small amount of source code.

Specifically, when the values in the labels of the CASE statement are widely separated, the use of nested IF statements is much more memory efficient and will achieve the same result. The compiler generates a table whose length, in bytes, is twice the difference between the smallest label value and the largest label value. The compiler will produce a warning if the difference between label values is greater than 1000.

Example:

```
CASE I OF
     0 : J := 1;
     10000 : J := 2;
     OTHERWISE
          J := 3;
     END;
```

The above example CASE statement would generate over 20,000 bytes of object code, and very likely would cause the compiler to run out of memory.

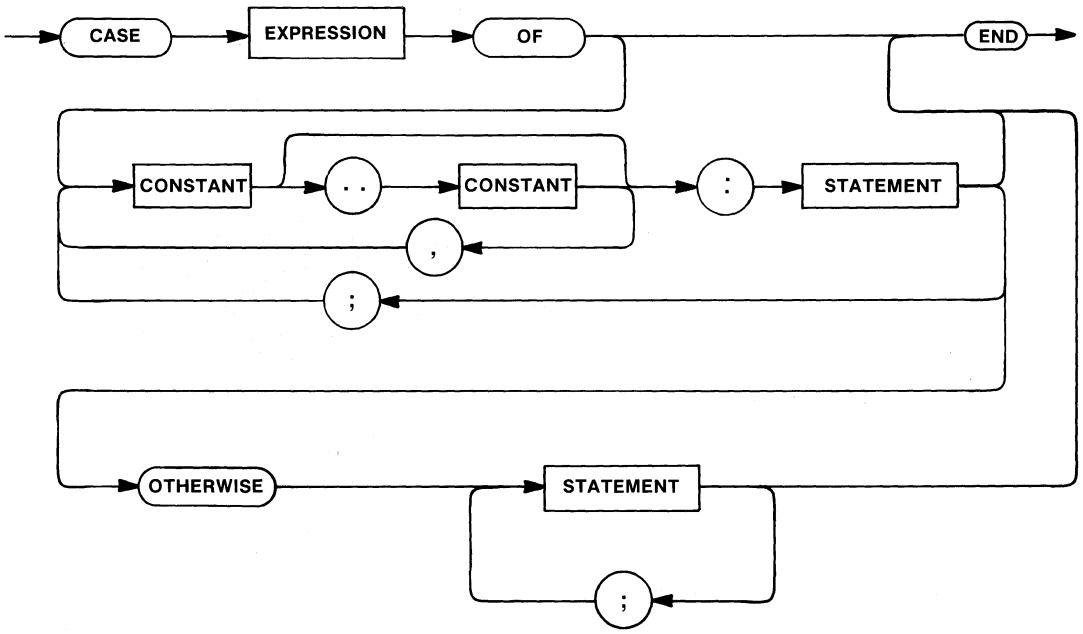

Figure 5-6. CASE Statement Syntax

WHILE Statement

The WHILE statement is a repeating statement used to execute an action so long as a given expression is true. The expression is evaluated before execution, in contrast to the REPEAT statement which is evaluated after execution. The expression must be of the boolean type. Each time the evaluation is true the WHILE statement is executed. When the evaluation becomes false the statement following the WHILE statement is executed and program action is continued.

It is necessary that execution of a WHILE statement causes a change in data such that the evaluation result becomes false. Otherwise the WHILE statement is never exited, an endless loop exists and execution of the program is never concluded. The WHILE expression syntax is illustrated in Figure 5-7.

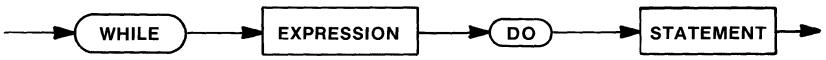

Figure 5-7. WHILE Statement Syntax

REPEAT Statement

The REPEAT statement is executed so long as the UNTIL Boolean expression is false. The expression is evaluated after execution of the statement enclosed by the REPEAT/UNTIL delimiters, in contrast to the WHILE expression which is evaluated before execution. The expression must be of the boolean type.

Each time the evaluation returns false the REPEAT statement is executed. When the evaluation returns true the statement following the REPEAT is executed and program action is continued. It is necessary that execution of a REPEAT statement causes a change in data such that evaluation results in a true value. Otherwise the REPEAT statement is never exited, an endless loop exists and execution of the program is never concluded. The REPEAT statement syntax is illustrated in Figure 5-8.

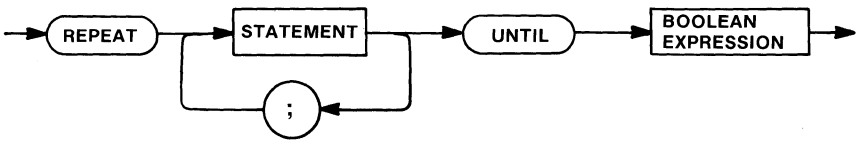

Figure 5-8. REPEAT Statement Syntax

FOR Statement

The FOR statement executes 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. The control variable must be a local variable, and it also must be an entire variable. 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, typically: FOR n := 1 TO 10 DO. The range specified is 1 TO 10. The FOR statement is not executed, and the control variable is not changed if the initial expression is greater than the final expression with the FOR..TO statement (or less than the final expression with the FOR..DOWNTO statement). The range expressions must be assignment compatible with the type of the control variable. These expressions are evaluated only once, before any assignment is made to the control variable. The FOR statement syntax is illustrated in Figure 5-9.

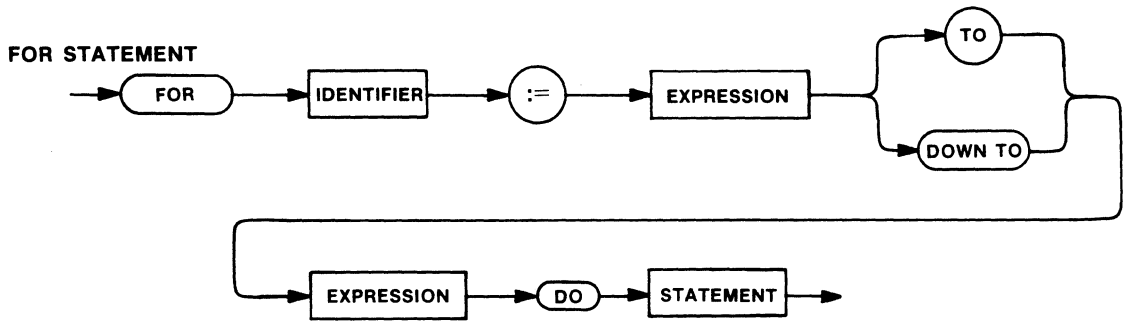

Figure 5-9. FOR Statement Syntax

WITH Statement

The WITH statement allows access to record fields without naming the record. 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.

When the record expression is a function returning a record, the fields of the record may only be used in other expressions. (i.e., they may not be used on the left side of an assignment statement.)

Example:

```
TYPE
     R = record
          aa:integer;
          b,c:real;
     END;
FUNCTION  A : R;
     BEGIN
          A.aa := 3;
          A.b := 0;
          A.c := 0;
     END;
     •
     •
     •
     BEGIN
          with A do
               V := aa;
               •
               •
               •
     END;
```

WITH statement syntax is illustrated in Figure 5-10.

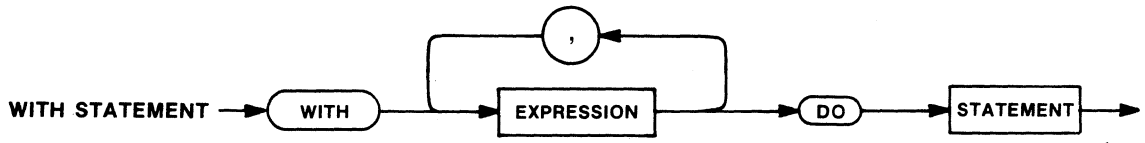

Figure 5-10. WITH Statement Syntax

GOTO Statement

The GOTO statement is used in conjunction with a label. The label must be an integer in the range of 1..9999. Program execution is transferred to the statement named by the label. The label in a GOTO statement may be defined in the same body as the GOTO statement, or it may be defined in an enclosing block. The latter case is referred to as an "out-of-block" GOTO. The execution of an "out-of-block" GOTO automatically closes any files that were local to any of the exited blocks. The GOTO statement syntax is illustrated in Figure 5-11.

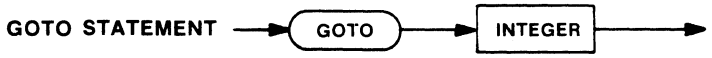

Figure 5-11. GOTO Statement Syntax

In HOST Pascal the GOTO statement should not direct program execution into the middle of any FOR or WITH statement because results may be undefined.

Empty Statement

The empty statement is denoted by no symbol and performs no action. It is used to indicate that no action is to be taken as the result of a condition evaluation.

Example:

```
IF a < b
     THEN
     ELSE
          writeln ('a is greater than or equal to b.');
```

THEN has no action statement associated with it, therefore an empty statement exists.

Expressions

An expression is a construct composed of operators and operands, and is used to compute a value of some type. An operator defines an action to be performed on its operands. Operands denote the objects that operators will use in obtaining a value. An operand may be a literal, a constant identifier, a variable, or it may be a reference to a function. The syntax diagram for expressions is shown in Figure 5-12, and is expanded into greater detail in Figures 5-13 thru 5-15.

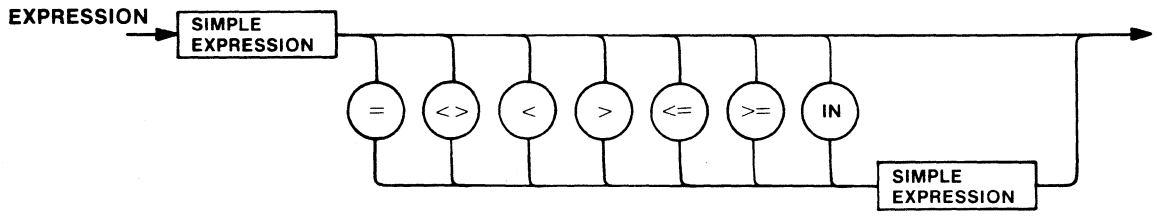

Figure 5-12. Expression Syntax

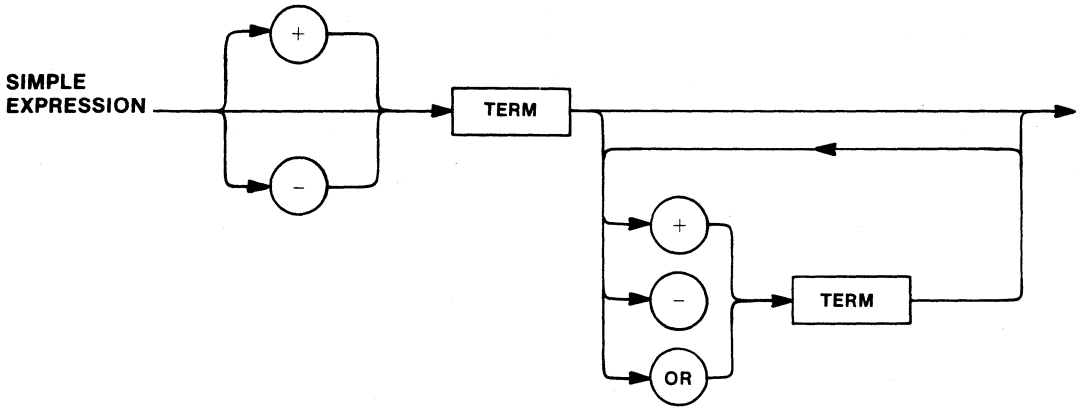

Figure 5-13. Simple Expression Syntax

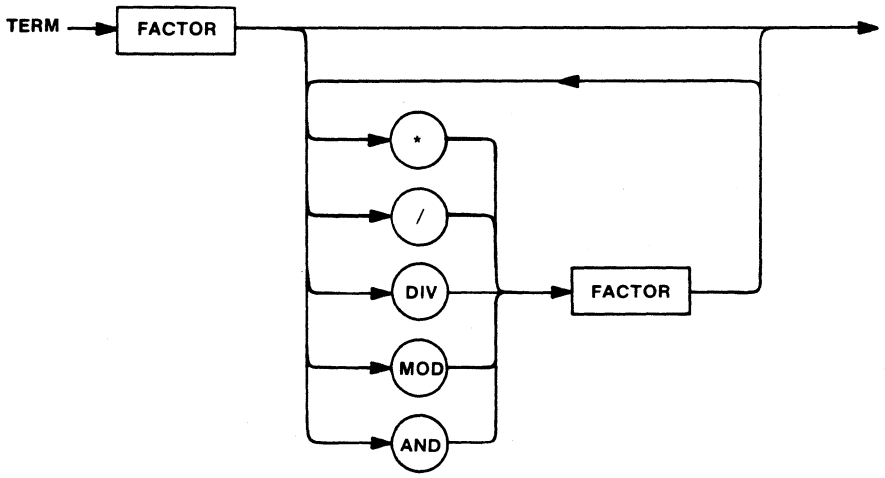

Figure 5-14. Term Syntax

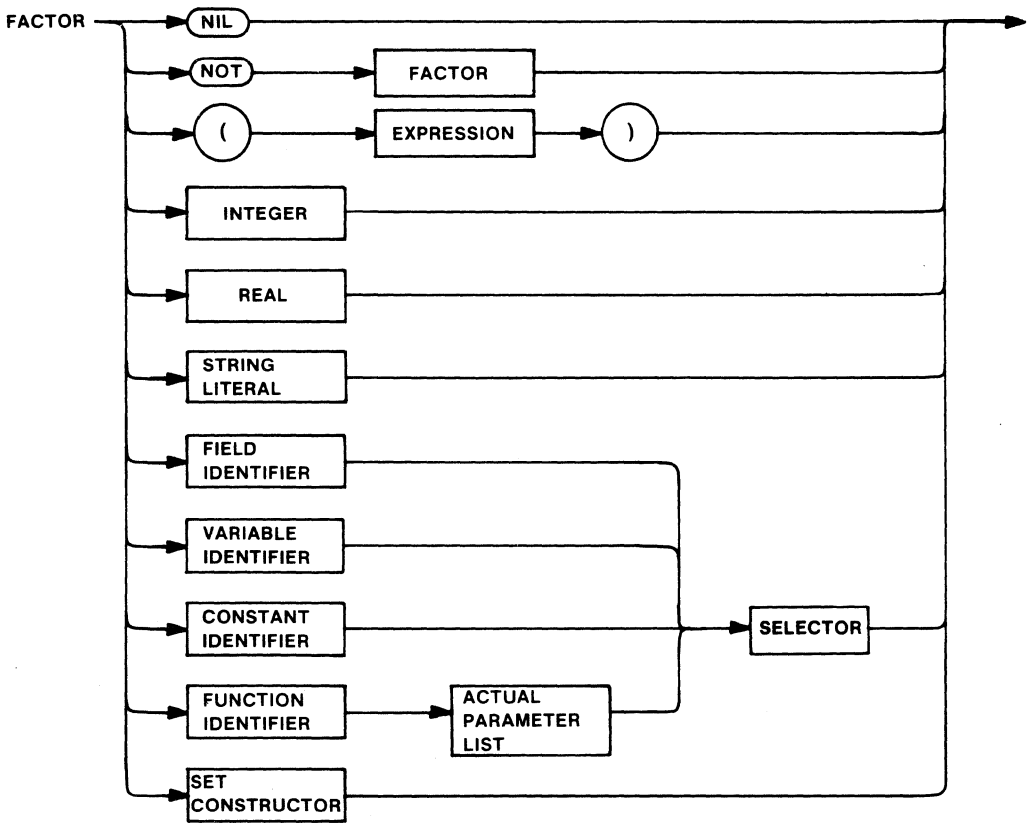

Figure 5-15. Factor Syntax

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 is a literal, symbolic constant, variable, function or the value of another expression, that can be acted upon by an operator.

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

Integer literals consist of numbers of the type integer. Spaces may not be used within an integer literal. Integers can be represented only in decimal notation.

Real Literals

Literals of the types REAL and LONGREAL consist of numbers of the type REAL. Exponential notation may be used to represent real or longreal values. The letter "E" preceding a scale factor specifies an exponent with a real number, and is read "times 10 to the power of". The letter "L" preceding a scale factor specifies an exponent with a longreal number. Decimal points must be preceded and followed by at least one digit. Spaces can not be used in real or longreal numbers.

Examples:

```
0.5  3.79E-3  3.79L-3  8E+4  8L4
```

String Literals

String literals consist of groups of characters set off by single quotation marks. A single character can be considered as a type CHAR or as a type string. If a single quotation mark is included in a string, it must be shown twice. Printable ASCII characters appear in strings in the normal manner with the exception of the apostrophe ('), which must be inserted twice.

Examples:

```
'DON''T USE THIS CONTAINER'.
'This is a string.'
```

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

Examples:

```
#27'that was an ESC character, as this is, too.'#[
'This string has 5 bells'#g#g#g#7#7' in it.'
```

Symbolic Constants

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

Variables

A variable is an identifier that represents a changeable data item. The variable must be declared and associated with the type of data it represents. The declaration takes place in the VAR portion of the block to which it is local. The 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, the variable is 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.

Selectors

A selector specifies a particular component of a structured expression. The selector may be applied to a structured variable or symbolic constant, or to a reference to a function that returns a structured type. The syntax for selectors is shown in Figure 5-16.

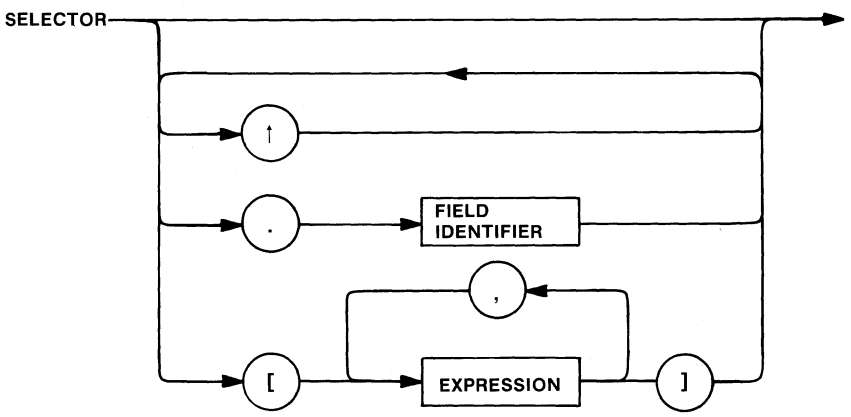

Figure 5-16. Selector Syntax

Array Subscripts

Array and string components are selected using subscripts, denoted by square ([]) brackets, and an expression. The subscript, or index, type must be compatible with the expression type appearing in the array type definition. The values of constants and non-constants are checked at run time to make sure those values lie in the range specified in the index type, 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.

Field Selection

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

Pointer Dereferencing

A pointer points to, or "references" a variable in the heap. To access that variable, the pointer is followed by the caret (^). At run time the pointer value is checked to make sure it isn't NIL before accessing the heap variable. The pointer may itself be a selected variable, or function reference. It may not be a selected constant, for the only pointer constant is NIL.

Examples:

```
p^  r.q^  ra[i].g^  ra[i].q^p^  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 base type, is denoted by following the file identifier with a caret (^). The file identifier appearing before the caret may itself be a selected variable, but may not be a selected constant or a selected function reference.

Examples:

```
f1^  f.ff^
```

Operators

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

a. number, order, and type of operands
b. result type
c. precedence

Operator precedence is used to determine the order of element evaluation in an expression. The higher precedence operators are evaluated first. Grouping of operators can alter the precedence value of the group; however, precedence within the group still follows the rules of precedence. The following list shows operators with their order of precedence, from highest to lowest.

```
NOT
*, /, DIV, MOD, AND
+, -, OR
<, <=, <>, =, >=, >, IN
```

Operators may either be predefined or user-defined. Predefined operators are the arithmetic, boolean, set, string, 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.

Arithmetic Operators

Arithmetic operators take numeric operands and produce a numeric result. A numeric type is the type REAL, LONGREAL, INTEGER, or any INTEGER subrange. If either operand is REAL or LONGREAL, the result will be of type LONGREAL. If both operands are integers, the result will be a two-word INTEGER. Operands are converted to match the type of the result before operation takes place. Integer values, for example, are converted to real values before an operation involving integer and real numbers.

In the case of real division (/), if both operands are integers, the operands are converted to REAL type before division takes place. The result is of type LONGREAL.

Operands for both DIV and MOD must be integers.

Integer division (DIV) calculates the truncated quotient of two integers. The sign of the result is positive if both operands have the same sign, and negative if the operands have opposite signs.

A div B is equivalent to trunc (A/B).

For the MOD operation, the sign of the result is always positive. I MOD J is defined only for j > 0. An error occurs if j <= 0.

A mod B is equivalent to (A - (k * b)) for integer k such that 0 <= A mod B < B, where B > 0.

Boolean Operators

The Boolean operators perform logical functions on Boolean operands. The Boolean operators are: NOT, AND, OR.

NOT

The NOT operator takes one Boolean operand and produces a Boolean result equal to the inverse of the operand.

AND

The AND operator yields a Boolean result of true only if all operands are true.

OR

The OR operator yields a Boolean result of true if any one of the operands is true; or yields a Boolean result of false only if all of the operands are false.

a	b	NOT a	a AND b	a OR b
T	T	F	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	F

All Boolean expressions are evaluated using either partial or full evaluation, depending on the setting of the PARTIAL_EVAL option. The default state of the PARTIAL_EVALuation option is off; however, the option may be changed at any time in the program.

Under PARTIAL_EVAL, the evaluation proceeds from left to right. The evaluation ceases for the AND operator when a "false" boolean value is detected. The evaluation process ceases for the OR operator when a "true" boolean value is detected.

Relational operators with Boolean operands are always fully evaluated. NOT, AND, OR cannot be used on operands of non-Boolean types.

Set Operators

Three infix operators are defined which manipulate two expressions having compatible set types and result in a third set. The set operators are: set union (+), set difference (-), and set intersection (*).

Set Union

The union operator creates a set whose members are all of those elements present in the first set plus those in the second set. Simply, the combining together of two sets into one set.

Set Difference

The difference operator creates a set whose members are those elements that are members of the first set but are not members of the second set.

Set Intersection

The intersection operator creates a set whose members are all of those members present in both sets.

The two operands of a set operator must be expression compatible. The distance between the lower and upper bounds of a set's base type is referred to as "width". One set is wider than a second set if every element in the second set is represented in the first.

Example:

```
set of 0..100 is wider than 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 set operation is performed, if either operand has a width other than the result's width, it is automatically widened prior to the operation.

Example:

```
TYPE
    HIGH = SET OF 50..100;
    LOW = SET OF 1..10;
    CROSS = SET OF 5..75;

VAR
    pos:HIGH;
    neg:LOW;
    crs:CROSS;
```

The result of the expression (neg + pos) is a set whose base type is the range 1..100. The result of the expression (crs + pos) is the range 5..100.

Narrowing of sets is also automatic for set assignments and actual value parameters. Run-time range checks are made 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 the type SET OF 5..100 and thus must be narrowed to SET 5..75. A run-time error will occur if the result has as members any of the numbers in the range 76..100.

Set Constructor

The set constructor notation is considered to be an operator and it too creates a set. Each expression value in the constructor is entered into the set. Every element between two expressions may be included by using the range (..) symbol between the expressions. A type identifier may be used to specify exactly what type of set is to be created. See Figure 5-17 for the Set Constructor Syntax diagram.

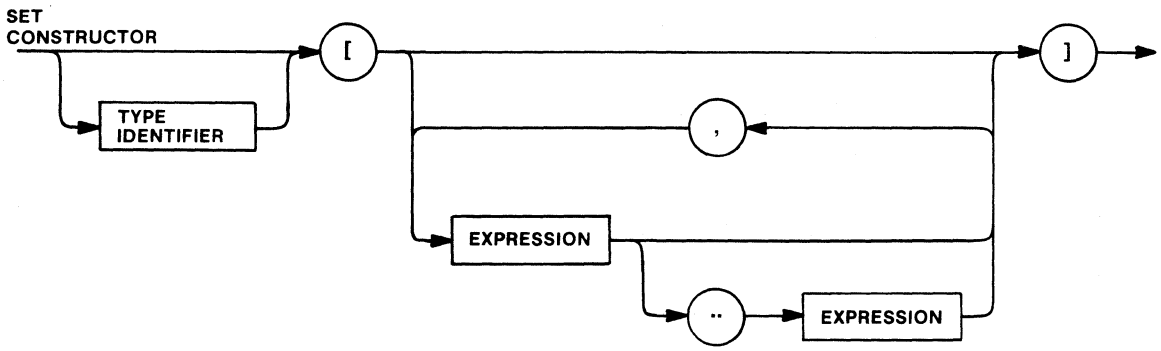

Figure 5-17. Set Constructor Syntax

If no type identifier is used, one of three possible results will occur depending on the type T of the elements in the set:

a. If T is an integer, then the set created is of type SET OF 0..255. Compile-time and run-time checks are performed to ensure that specified elements are in this range. Thus the set [25,0,255] is legal, but the set [-10, 256] is not legal.

b. 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'] has the type SET OF CHARacter.

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

The type identifier 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 run time.

String Operators

The operator + specifies the concatenation of two string expressions. An error will result if the combined length of the two expressions is greater than 255.

Example:

```
VAR
     s : string[100];
          •
          •
          •
BEGIN
     s := 'part 1';
     write (s + 'part 2');
END.
```

Relational Operators

Relational operators are used to compare two operands and return a boolean result. The operands may be INTEGER, REAL, LONGREAL, sets, boolean, or pointers. Relational operators appear between two expressions, that must be compatible, and always result in a value of type boolean. The relational operators are:

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

Ordinal Relationals

The relationals that can be used with operands of the types integer, boolean, char, or any enumeration or subrange type, are: <, >, <=, =, <>, and >=. These operators carry the normal definition of ordering for numeric types, and char relationals are defined by the ASCII collating sequence. The order of enumerated constants is defined by the order in which the constant identifiers are listed in the TYPE definition. The predefinition of boolean is: BOOLEAN = (false, true) and means 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 following truth table:

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

<= is the implication operator, = is the equivalence operator, and <> is an exclusive OR operator.

PAC Relationals

PACs (packed arrays of characters) may be compared using the operators =, <>, <, >, <=, or >=. The two PACs must have the same number of components. In addition, PACs may be compared to string literals that have the same number or fewer characters than the PAC.

PACs are compared character by character until either a pair of unequal characters are found, or until all characters have been compared. The ordering of two PACs is determined by the first pair of unequal characters according to the ASCII collating sequence.

Comparing a PAC to shorter string literal is an HP extension to standard Pascal. When a PAC is compared to a shorter literal, the literal is extended on the right with blanks until the two are equal in length.

String Comparison

A string expression can be compared to any other string expression, including string literals. Strings are compared character by character until a pair of unequal characters are found or until all the characters in the shorter string are used. If two characters are unequal, the ordering of the string is determined by the ASCII collating sequence. If all the characters in two strings are equal up to the length of the shorter string, then the longer string is greater than the shorter string. Two strings are equal only if they have the same length and all characters contained in that length are equal.

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 be identical.

Set Relationals

Two sets can be compared for equality 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 set if every element in the first set is also a member of the second set. Also, the second set is said to be a superset of the first set. Sets are widened, if necessary, before the relational operation. The < and > operators are not allowed on sets.

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

Function References

A reference to a function can be thought of as an operator whose operands are the actual parameters passed to the function; or as an operand whose value is determined by the process in the function.

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. 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, if several components of the structure will be accessed.

Functions may be recursive.

Constant Expressions

A constant expression is one that the compiler is able to evaluate at compile time. The syntax is no different from ordinary expressions, but there are restrictions on the operators and operands of a constant expression. Allowed in constant expressions are the following:

Operators

+ (unary and binary)
- (unary and binary)
*
DIV
MOD

Predefined Functions

pred
succ
ord
chr
odd
abs (except for REAL or LONGREAL operands)
hex
octal
binary

Operands

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

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

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

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

Type Compatibility

A set of compatibility requirements for the operands of each operator is based both on the operator and the types of its operands.

Relative to each other, two types in Pascal are either:

a. identical,
b. compatible,
c. assignment compatible,
d. expression compatible, or
e. incompatible.

Identical Types

Two types are identical if either of the following is true:

a. their types have the same type identifier.

b. if the two type identifiers have been equivalenced by a definition in the form T1 = T2.

Compatible Types

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

a. T1 and T2 are identical types.

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

c. T1 and T2 are set types with compatible base types.

d. T1 and T2 are string types.

e. T1 and T2 are both PAC types with the same number of components.

f. T1 and T2 are both real types.

Assignment Compatible Types

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

a. T1 and T2 are compatible types which are not files or structures that contain files.

b. T1 is a real type and T2 is compatible with INTEGER.

c. T1 is REAL and T2 is LONGREAL. In this case the LONGREAL is rounded before being assigned. A run-time error will occur if the LONGREAL value is outside the range of the REAL.

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

e. 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.

f. T1 is a PAC and T2 is a string literal with the same or fewer components as T1. In the case where T2 is shorter than T1, T2 will be extended on the right with blanks.

For operations that require assignment compatibility, a compile-time 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 that 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 REAL values.

d. INTEGER values are converted to LONGREAL values.

e. LONGREAL values are rounded to REAL values.

f. Set values are widened or narrowed to the type T1.

g. Shorter string literals are extended on the right with blanks to become compatible with longer PACs.

Two types, T1 and T2, are expression compatible if either of the following is true:

a. T1 is assignment compatible with T2.

b. T2 is assignment compatible with T1.

Special Cases

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

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

Chapter 6

Files

Introduction

Files, although declared in the VARiable section, are different from other variables. The main purpose of a file is to allow the program to communicate with the program environment.

Logical Files

All file variables used in HOST Pascal programs are logical files. Each logical file has an identifier associated with it. The logical file structure consists of a sequence of components of the same type. These components may be of a simple type, such as INTEGER, REAL, or CHAR; or they may be of structured types, such as arrays or records. The components cannot be of type FILE or a structured type that contains a file.

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

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

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

Examples:

```
TYPE
     STR70 = PACKED ARRAY [1..70] OF CHAR;
     person = RECORD
          name : STR70;
          age_in_years : 0..120;
          employee_number : 0..5000
     END;
VAR
     people_file : FILE OF person;
     string_file : FILE OF STR70;
     int_file : FILE OF CHAR;
     num_file : FILE OF INTEGER;
```

A logical file must be opened if it is to be accessed. The procedure used in opening the file determines how the file components may be accessed.

The compiler automatically generates code for the files INPUT and/or OUTPUT, if the file was listed in the program heading. RESET code is

applied to file INPUT, and REWRITE code is applied to file OUTPUT, at the beginning of the program body.

Sequential Files

Sequential files are logical files that have been opened through the procedures RESET, REWRITE, or APPEND. Components in these files must be accessed in sequence.

An existing sequential file can be changed only by rewriting the entire file (REWRITE), or by appending new data to the file (APPEND).

The process of opening associates the logical file with a physical file.

Physical Files

A physical file is either an I/O device or a disc file that exists outside the program environment on the 64000 system. I/O devices that may be accessed are: display, printer, keyboard, display1, rs232, and null. Disc files are identified with file names. A file name has the syntax:

```
<name>[:[<userid>]][:<digit>][:<type>]
```

Textfiles

The identifier TEXT is predefined as follows:

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

Textfiles are similar to other files but are further structured into lines that are separated by line markers. Line markers may be generated by the standard procedure WRITELN and be detected by the standard function EOLN.

A textfile type definition specifies a structure consisting of a sequence of components that are all of type CHAR. The number of components is not fixed by the file type definition. Sequential operations applicable to variables of the type File of CHAR also apply to textfiles. Line markers are not of type CHAR. If a READ is performed when the end-of-line has been reached, a blank character will be stored into the file variable.

Logical File Characteristics

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

File Buffer Variable

The file buffer variable is of the same type as the file's component type, denoted: f^, where f is the identifier associated with the file.

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

Once the file has been opened, the file buffer may be accessed by the program as a variable of the file component type. The contents of the file buffer may be assigned to a variable through the assignment statement. For example: variable_id := file_id^ would assign the contents of the file_id buffer to "variable_id". The variable must be of a type that is assignment compatible with the file's component type. (Files of the predeclared type TEXT have file buffers variable that are assignment compatible with variables of the predeclared type CHAR).

Example:

```
TYPE
     Book_info = RECORD
          title : packed array [1..50] of char;
          author : packed array [1..50] of char;
          number : 1..32000;
          status : (on_shelf, checked_out,lost,ordered)
     END;
VAR
     book : book_info;
     book_file : FILE OF  book_info;
          •
          •
          •
     BEGIN
          book := book_file^;
               •
               •
               •
     END;
```

Current Position Pointer

The current position pointer marks a component of the file. It is used with the file buffer to access components of the file. The first component of a file is number 1.

File States

Logical files are either OPEN or CLOSED. Logical files, when open, will be in either a readable or writable state. Files are opened into a readable state by the procedure RESET. Files are opened into a writable state by the procedure REWRITE, or by the procedure APPEND. Files that are closed are not accessable.

Opening Files

A file cannot be accessed, although previously declared, until it has been opened. There are three predefined procedures that can be used to open a file. Those procedures are: RESET, REWRITE, and APPEND. Files opened by these procedures are sequential files.

RESET(f)
RESET(f,s)
RESET(f,s,t)

A file opened by the procedure RESET(f) is opened in a readable mode.

If the file (f) is already open when RESET is called, then file (f) is automatically closed and then reopened.

Then the logical file (f) is associated with a physical file. Association is performed by use of a file name. The name is obtained according to the rules given in the section on "Associating Logical and Physical Files". If the physical file does not exist, an error occurs. If the file is not empty, the file pointer will be positioned at the first component of the file.

After the procedure RESET(f) is called, the current position points to the first component if file (f) is not empty. Then a call to the standard procedure GET occurs. In addition, the function EOF(f) remains FALSE. If the file is empty, the contents f^ is not defined and EOF(f) becomes TRUE.

An optional second parameter, (f,s), may be included in the procedure call. This is a string parameter that may be used to specify the physical file by name. The string parameter is described in the section titled "Associating Files Through The String Parameter", in this chapter.

A third parameter, (f,s,t), is also optional and of type string. The string may contain system dependent information. The third parameter is not used by HOST Pascal but is allowed for compatibility with other HP Pascal systems.

REWRITE(f)
REWRITE(f,s)
REWRITE(f,s,t)

A file opened by the procedure REWRITE(f) is in the writable mode. Rewrite(f) discards any previously existing components (file is now empty) and the current position points to the first position of the file. The content of the file buffer f^ is undefined, and the function EOF(f) becomes TRUE.

If the file (f) is open when REWRITE is called, then file(f) is automatically closed and reopened.

If the physical file associated with (f) exists and is a disc file, the physical file is purged, any existing components of the file are lost, and a new physical file is created.

A second parameter, (f,s), described in the section "Associating Files Through The String Parameter", may be included in the procedure call.

A third parameter, (f,s,t), may be included, and is interpreted as in the procedure RESET.

APPEND(f)
APPEND(f,s)
APPEND(f,s,t)

A file opened by the procedure APPEND(f) is in the writable mode. The components of file(f) are not discarded, however. The current position pointer is set to just after the last existing component in the physical file. The content of the file buffer is undefined and the function EOF(f) is TRUE.

If the file(f) is open before APPEND is called, file(f) is automatically closed and reopened.

If the physical file associated with (f) does not exist, it will be created, as described in the section REWRITE.

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

A third parameter, (f,s,t), may also be included. This parameter is interpreted as by RESET.

Associating Logical and Physical Files

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

a. An external name may be supplied as a second parameter to the predeclared procedures APPEND, RESET, and REWRITE.

Example:

```
RESET (T,'DATA:MINE:data');
```

b. If the logical file was open before the call to APPEND, RESET, or REWRITE, then the physical file associated with (f) will be the same physical file that was associated with (f) before the call.

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

d. If no external name is supplied, a file name will be created. This name will be the same as the first nine characters of the logical file identifier. Any lower case alphabetic characters will be converted to upper case.

Associating Files Through the String Parameter

One method of associating logical and physical files is through the string parameter. This method involves the use of the optional second parameter in the predefined procedures APPEND, RESET, and REWRITE.

The second parameter of the procedures is a string expression that names a physical file to be associated with the logical file named by the first parameter. Note that apostrophes must be treated as those in a string literal.

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

Sequential File Operations

A file can be accessed only after it has been opened. There are six predefined procedures in HOST Pascal that can be used to access the components of a file. Those procedures are GET, PUT, READ, READLN, WRITE, and WRITELN.

Textfile Operation

Textfiles may be used as parameters with any of the procedures used to access non-text sequential files. In addition, several standard procedures and functions can be used exclusively with textfiles. The procedures are READLN, WRITELN, and PAGE; the functions are LINEPOS, and EOLN.

GET(f)

The procedure GET(f) advances the current-position pointer one component, but does not cause the file component to be assigned to the file variable. Rather, after performing GET(f), a subsequent reference to the

file buffer f^, or a call to eof(f) or eoln(f), will actually cause the input operation and the assignment of the file component to the file variable. This so-called "deferred GET" implementation is useful when the physical file being accessed, such as a keyboard, is interactive in nature.

If the component is the last one of the file, the function EOF(f) will be TRUE the next time GET(f) is called. If the file was not opened in the readable mode, or if the EOF(f) was TRUE prior to the procedure call, an error will occur.

PUT(f)

The procedure PUT(f) assigns the contents of the file buffer into the current component of (f), and advances the current-position pointer to the next component. Following the procedure call, the content of the file buffer is undefined. An error occurs if the file was not in a writable mode prior to the procedure call.

READ(f,v)

For a file(f) that is not a textfile, and a variable (v) of a type that is assignment compatible with the type of the file's components, the procedure call READ(f,v) will assign the contents of the file buffer to variable (v), and advance the current-position pointer one component. If file(f) is omitted, the file INPUT is assumed.

Errors will occur if the file was not opened in the readable mode, or if EOF(f) was TRUE prior to the call to READ(f,v).

The procedure READ(f,v) may contain additional parameters, v1,...,vn. READ(f,v1,...,vn) is equivalent to:

```
READ(f,v1);
     •
     •
     •
READ(f,vn);
```

READ(v) is equivalent to READ(input,v); READ(f,v) transfers from file (f) to variable (v). If (f) is not a text file, READ(f,v) is equivalent to:

```
v := f^;
GET(f);.
```

The file component type must be assignment compatible with (v). (v) may be a component of a PACKED structure.

READ(f,v) With Textfiles

Although the textfiles contain only components of type CHAR, the variable (v), in the procedure call READ(f,v), may be of type INTEGER, REAL, LONGREAL, PAC, string, a subrange of INTEGER, CHAR, or a subrange of CHAR. This is possible because the procedure does an implicit conversion from the ASCII form that appears in the text file to the actual form stored in the variable.

If variables of type REAL, LONGREAL, INTEGER, or INTEGER subrange are included as parameters in the procedure READ, the file will be searched for characters that satisfy the syntax of the variables.

If (f) is a textfile, even though the textfile contains only characters, (v) may be compatible with CHAR and may be a component of a PACKED structure; (v) may be compatible with INTEGER; (v) may be REAL or LONGREAL; (v) may be a PAC; or (v) may be a string.

The action performed depends on the type of the variable.

If (v) is compatible with CHAR, then READ(f,v) is equivalent to v := f^; GET (f).

If (v) is compatible with INTEGER, then READ implies reading from (f) a sequence of characters forming a number according to the syntax shown in Figure 6-1.

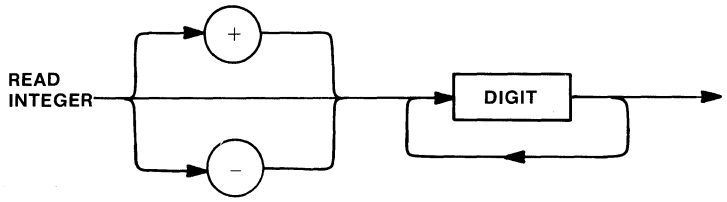

Figure 6-1. READ INTEGER Syntax

An implicit conversion from the ASCII representation to the internal representation takes place if the variable is INTEGER or REAL.

An error will occur if the proper sequence of characters is not found or the value read is not in the range of integers.

Preceding blanks and line markers are skipped. The value will be assigned to (v) after reading the digits. f^ will then contain the character immediately following the last digit read.

The READ syntax, if (v) is REAL or LONGREAL, is shown in Figure 6-2.

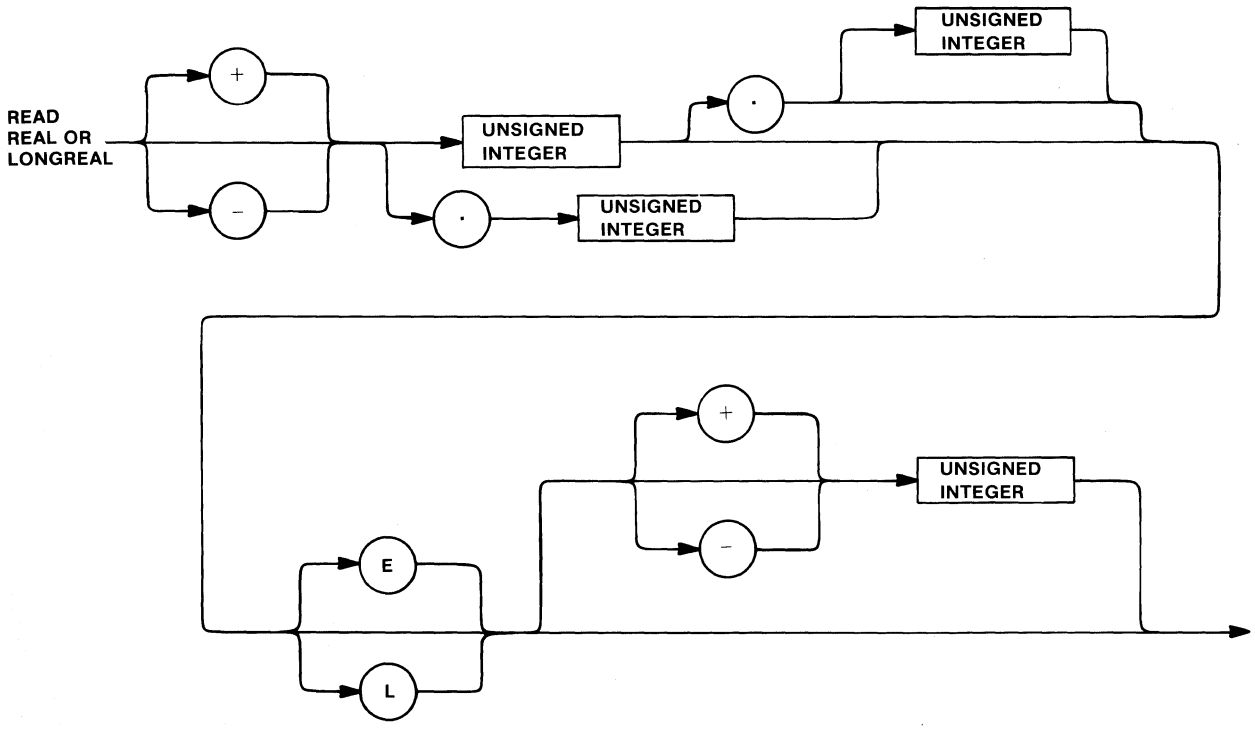

Figure 6-2. READ REAL or LONGREAL Syntax

An error will occur if the proper sequence of characters is not found or the value read is not in the range of REAL or LONGREAL.

Preceding blanks and line markers are skipped. The value will be assigned to (v) after reading the digits. f^ will then contain the character immediately following the last digit read.

If (v) is a PAC type, then characters are read into (v) until either (v) is full or EOLN(f) becomes TRUE. If EOLN becomes TRUE before (v) is full, then the remainder of (v) is filled with blanks. If EOLN(f) is TRUE when READ(f,v) is called, then an initial READLN(f) is done to skip past the line marker.

If (v) is a string variable, then characters are read into (v) until either (v) is full or EOLN(f) is true. The length of the string will be set to the number of characters that were read. If eoln(f) is true when read(f,v) is called, an initial readln(f) is done to skip past the line marker.

READLN(f,v)

The parameter list for READLN(f,v) is similar to that of the procedure READ. The file INPUT is assumed if the file identifier is missing. Several variable parameters of the types CHAR (or subrange of CHAR),

REAL, LONGREAL, INTEGER (or a subrange of INTEGER), PAC, or string can be used.

READLN(v) is equivalent to READLN(input,v). READLN(f,v1,...,vn) is equivalent to:

```
READ(f,v1,...,vn);
READLN(f);.
```

READLN(f) is equivalent to:

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

READ and READLN differ in their use of the end-of-line. After a call to READLN, the current-position pointer is positioned after the end-of-line. The variable parameters are filled followed by a skip to the next line, ignoring whatever remains in the line.

WRITE(f,e)

The procedure call WRITE(f,e) will assign the value of the expression (e) to the file buffer, assign the contents of the file buffer into the current component of (f), and advance the current-position pointer to the next component.

If the file parameter is not included in the procedure call, then (f) is assumed to be the OUTPUT file.

An error will occur if the file was not in a writable mode prior to the procedure call.

The procedure WRITE(f,e1) may contain additional parameters, e2,...,en, of a type compatible with the file's component type. The procedure call WRITE(f,e1,...,en) is equivalent to:

```
WRITE(f,e1);
     •
     •
     •
WRITE(f,en);
```

WRITE (e) is equivalent to WRITE(output,e).

If (f) is not a textfile, then WRITE(f,e) is equivalent to f^ := e; PUT(f);. Note that e must be assignment compatible with the component type of (f).

If (f) is a textfile, then write parameters may have one of the following forms:

a. e
b. e:m
c. e:m:n

where e, m, and n are expressions. The form e:m:n is legal only if the type of e is REAL or LONGREAL.

WRITE(f,v) With Textfiles

The procedure WRITE(f,v), when used with textfiles, uses the variable identifiers occurring as "write parameters". The program ouput can be formatted through the use of these write parameters to display program results in a more readable form.

Write parameters may be listed in one of three different forms:

a. expression

b. expression:m

c. expression:m:n

where m and n are field width parameters and must be expressions compatible with integer.

If default formatting is desired, the first form is used. The type of the expression may be an integer, real, longreal, char, Boolean, PAC, or string. The field-width parameter (m) will be defaulted depending on the type of the expression. The field width default values are shown in table 6-1.

Table 6-1. Field Width Parameter Default Values

PARAMETER TYPE	(m) FIELD WIDTH DEFAULT
CHAR	1
PAC	length of PAC
string	current length of string
INTEGER	12
REAL	12
LONGREAL	20
BOOLEAN	length of TRUE or FALSE

Example:

```
flop := 20;
write('This string contains');
write(flop);
write(' characters.');
```

will produce output of the form:

```
This string contains          20 characters.
```

Field width parameters can be used to adjust the space into which a value is written when formatting is desired. For expressions of type INTEGER, CHAR, or string, only the first field width parameter can be specified, therefore, the second form of write parameter is used.

The field width parameter (m) is an integer expression specifying the number of characters that will be used to represent the value in the textfile. If (m) is greater than the number of characters actually needed, the additional characters will be represented as blanks preceding the value. Thus the value is right justified.

For integer or real expressions (m) is ignored and text remains unabridged if parameter (m) is less than needed. Text is truncated, however, if (m) is less than needed for CHAR, string, and enumerated types. If (m) < 0, an error will occur.

Both (m) and (n) parameters can be used to format REAL values. If parameter (n) is present, a fixed-point representation with (n) digits after the decimal point is obtained. If (n) is 0 the decimal point will be omitted. No more significant digits will be written than are contained in the internal representation. If (n) is less than the number of significant digits in the internal representation, the number will be rounded off. When the parameter (n) is missing or the value cannot be expressed with a fixed-point representation, a floating point representation consisting of a coefficient and scale factor will be chosen.

A text file has a maximum line length that may be controlled using the LINESIZE compile option. The Host Pascal system insures that longer lines are not written by automatically performing WRITELN operations if necessary.

When writing to a text file, the system counts the number of characters written since the last WRITELN operation.

When writing each field, the system checks to see if the new field will fit on the present line without exceeding the maximum line length. If the field will not fit, the system performs a WRITELN operation and then writes the field on the next line.

WRITELN(f,p)

The procedure WRITELN is the same as the procedure WRITE, except that an end-of-line marker is placed immediately after writing the values of the write parameters. As with the procedure WRITE, parameters in WRITELN may be of type CHAR, subrange of CHAR, INTEGER, REAL, LONGREAL, BOOLEAN, PAC, string, or a subrange of INTEGER.

WRITELN is equivalent to writeln(output). writeln(f,p1,...pn) is equivalent to:

```
WRITE(f,p1,...pn); WRITELN(f);
```

PAGE(f)

The procedure PAGE(f) will cause the next element written to textfile (f) to appear at the top of the next page. PAGE causes the line printer to skip to the top of form so it will only effect a printed listing. If the file (f) is not a textfile, an error will occur. If the procedure call contains no parameters, the predefined file OUTPUT is assumed.

LINEPOS(f)

LINEPOS is a function that returns the number of characters read from, or written to the file since the last end-of-line. The component currently in the file buffer is not included in this count.

LINEPOS(f) is an HP Pascal extension.

EOLN(f)

The function EOLN will return the Boolean value TRUE if the position pointer is located at the line marker, and will be FALSE at any other position.

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

Closing Files

A file is closed by use of the procedure CLOSE. Any attempt to access a closed file will produce an error.

Opening a file will implicitly close any physical file previously associated with that logical file.

If any physical file was associated with a logical file, the procedure CLOSE will save the physical file unless the file is purged through the string parameter PURGE. The procedure CLOSE(<file_name>, PURGE) will purge any physical file associated by the program with the file (file_name) at the time of the call.

The procedure CLOSE is an HP Pascal extension.

Summary of Procedures and Functions

Table 6-2 provides a brief summary of procedures and functions listed in chapter 6, and the types of file with which they can be associated.

Table 6-2. Procedure and Function to File Association

	Sequential File	
	text	non-text
PUT	x	x
GET	x	x
RESET	x	x
REWRITE	x	x
APPEND	x	x
CLOSE	x	x
EOF	x	x
EOLN	x	
LINEPOS	x	
READ	x	x
READLN	x	
WRITE	x	x
WRITELN	x	
PAGE	x	

Chapter 7

Standard Procedures and Functions

File Handling Procedures

The following procedures are used to manipulate files in the HOST Pascal 64000 compiler.

The procedures APPEND, RESET, and REWRITE may have first, second, and third parameters (f,s,t). The optional second parameter is a string that specifies a system file to be associated with (f). The optional third parameter is also a string specifying implementation dependent options relating to the file. If s and t parameters are all blanks they are treated as if they were omitted.

The following actions occur when APPEND, RESET or REWRITE are called:

a. The file (f) is closed if it was open.

b. An actual file or I/O device is associated with (f). The name of the file or device is determined as follows:

   1. The second parameter is used as a file name.

   2. If the second parameter is all blanks, or was omitted, then the following takes place:

      a) If the file was open previous to the call to APPEND, RESET, or REWRITE, the same file is used.

      b) If the file was not open previous to the call, and if the file was a program parameter, then the name given to the program parameter is used as file name.

      c) If the file was not a program parameter, or if the file name associated with the program parameter was all blanks, then the file name becomes the same as the file variable identifier.

APPEND (f)
APPEND (f,s)
APPEND (f,s,t)

The procedure APPEND opens the file as a sequential file in the writable mode. Any components previously existing in the file remain, and the pointer is positioned directly after the last component of the file. The content of the file buffer is undefined. The boolean eof(f) will be TRUE. New data will be added to the file, at the end of the file.

The parameter (f) must be a file that has been previously declared. It need not be closed before the call is made. If the file was open before the procedure call, the file is automatically closed and reopened in a writable mode.

RESET(f)

The procedure RESET opens the file as a sequential file in the readable mode. The current-position pointer is initially positioned at the first component of the file, followed by a GET.

The parameter (f) must be a file that has been previously declared. It need not be closed before the call is made. If the file was open before the procedure call, it is automatically closed and then reopened in the readable mode.

REWRITE(f)

The procedure REWRITE opens the file as a sequential file in the writable mode. The current-position pointer is placed at the first component. Any components previously existing in the file are discarded and the file buffer is undefined. The function eof(f) is true.

If the Pascal file is associated with a system file by the procedure REWRITE, all contents of that system file are destroyed.

CLOSE(f)
CLOSE(f,s)

The procedure CLOSE makes the file unavailable for accessing, and association with a system file is dropped. The content of the file buffer is undefined. The boolean eof(f) will be true.

The parameter (f) must be a file which has been previously declared. It need not be open before the call is made. If the file was closed before the procedure call, no error will be produced.

The second parameter is optional and is of type string. This parameter can take on one value, PURGE. If the string PURGE is used, the file is destroyed when it is closed. If the second parameter is omitted or has any value except PURGE, the file will be saved.

GET

The procedure GET advances the current file position. A following reference to the buffer variable will actually move this component into the buffer variable. If the component does not exist, the content of the file buffer is undefined and eof(f) will be true. An error will occur if eof(f) was true before the call or if the file was not readable.

The parameter (f) must be a file that has previously been opened in a readable mode.

PAGE

The procedure PAGE causes printer orientation to the top of the next page when the text file (f) is printed.

The file (f) must have been previously declared as a text file. The file must be in a writable mode.

The file OUTPUT is assumed if the parameter (f) is omitted.

PUT

The procedure PUT writes the value of the buffer variable f^ to the current component of (f) and advances to the next component. Following the call, the content of the file buffer is undefined.

The parameter (f) must be a file that has previously been opened in a writable mode.

READ

The procedure READ accepts input from a file that has previously been opened in a readable mode. Data from the file is then assigned to variables specified as parameters in the procedure call.

The file INPUT is assumed if the file identifier is not included as a parameter.

The procedure call READ (f,v1,...,vn) is equivalent to the procedure calls READ(f,v1), READ(f,v2),...,READ(f,vn).

The procedure READ can be used to read from a file that is not a textfile. In that case READ (f,x) is equivalent to x := f^, GET (f). READ (f,x1,...,xn) is equivalent to READ (f,x1);...;READ(f,xn). If v is a variable of type CHAR, then READ(f,v) is equivalent to v := f^, GET(f).

If (f) is a textfile and v is a variable of type INTEGER (or subrange of integer) or REAL or LONGREAL, then READ(f,v) implies the reading from (f) of a sequence of characters that form a number according to the following syntax:

```
<integer number> ::= <sign><unsigned integer>;
<real number> ::= <sign><input real number><exponent>;
<input real number> ::= <digit sequence>|
    <unsigned integer>.<digit sequence>|<unsigned integer>|
     <unsigned integer>.;
<exponent> ::= E<scale factor>|L<scale factor>| E | L |
     <empty>;
```

The value of the number read is assigned to the variable v. Preceding blanks and line markers are skipped. Following the read, f^ will contain the next character immediately following the characters read. The result of reading a longreal number is independent of the letter preceding the scale factor.

READ(f,v), if v is a variable of type PAC, implies the reading of characters into v. If eoln(f) becomes true before v is filled, then the remainder of v is filled with blanks. If f^ contains the line marker when this call is made, an initial READLN is performed. In this case v will contain all blanks if the next component is the line marker.

READ (f,v), if v is a variable of string type, implies the reading of characters into v until either EOLN is true, or v is filled. The current length of v is set to the number of characters read. If EOLN is true when this call is made, an initial READLN is performed. In this case, v will contain the null string if the next component is a line marker.

READLN

The procedure READLN is used to read and skip to the next line. It is similar to the procedure READ used with text files in that the input is received from the file and assigned to the variable parameter(s). Once this action has been completed, however, the procedure READLN will ignore any remaining characters on the line and the next access to the file will begin on the following line.

The parameter (f) must be a file that has been previously declared as a text file and opened in the readable mode. The file INPUT is assumed if the file identifier is not included as a parameter.

The variables v1 thru vn may be of type CHAR, REAL, LONGREAL, INTEGER (or subrange of INTEGER), PAC, or string. Their values will be assigned in the same way as variable parameters of the procedure READ used with text files.

TIMEOUT(f,t)

The procedure TIMEOUT is used to specify the time interval for waiting for a character to be received on the "rs 232" I/O device. TIMEOUT has an effect only if text file f is open for reading to the RS232 device. The integer expression t specifies the time interval as a number of ticks of the real time clock. A clock tick is either 1/60th or 1/50th of a second depending on the AC line frequency. If the value of t is negative, timeout timing is disabled and the RS232 receiver will wait forever for a character. When the time interval is set, it remains in effect for all subsequent input operations until TIMEOUT is called again or the file is closed.

TIMEOUT is an extension to HP Standard Pascal and requires the compiler directive $EXTENSIONS ON$ to be in effect.

Example:

```
$EXTENSIONS ON$
    RESET(F,"RS232");
    TIMEOUT(F,120);
$IOCHECK OFF$
    READ(F,CH);
$IOCHECK ON$
    IF IORESULT = 0 THEN
      {Handle valid data}
    ELSE
      {Handle errors including timeout};
```

In the above example, the timeout interval is set to 120 ticks which is 2.0 seconds if the AC line frequency is 60 Hertz. The READ procedure will complete either when a character is received or when 2 seconds have elapsed, whichever comes first. The function IORESULT is used to determine which happened.

WRITE

The procedure WRITE places the values of its write parameters into a file (f) previously opened as a sequential file in a writable mode.

The file OUTPUT is assumed if the file identifier (f) is not included as a parameter.

WRITELN

The procedure WRITELN places the values of its write parameters into the text file (f), and appends a line marker to the file immediately following the last character. The statement:

```
WRITELN(f,p1,...,pn);
```

is equivalent to:

```
WRITE(f,p1,...,pn);
WRITELN(f);
```

The file identifier (f) must be a text file that has previously been opened. The predeclared file OUTPUT is assumed If the parameter (f) is not included.

String Handling Procedures and Functions

SETSTRLEN

The procedure SETSTRLEN(s,l) sets the current length of string variable s to l without changing any characters in the string. A run-time error occurs if the value of integer expression l is less than zero or greater than STRMAX(s).

Example:

```
VAR
    s1 : string [100];
              •
              •
              •
         setstrlen(s1,10);
```

STRAPPEND

The procedure STRAPPEND(s1,s2) concatenates string expression s2 to the value of string variable s1 and stores the result in s1. A run-time error occurs if STRLEN(s1) + STRLEN(s2) is greater than STRMAX(s1). This procedure is equivalent to:

```
s1 := s1 + s2;
```

Example:

```
    VAR
         s1 : string [10];

BEGIN
         s1 := 'ABC';
         STRAPPEND(s1, 'DE');
    END.
```

s1 now has the value 'ABCDE'.

STRINSERT

The procedure STRINSERT(s1,s2,startpos) inserts string expression s1 into string variable s2 starting at the position startpos. A run-time error occurs if the value of integer expression startpos is less than one or greater than strlen(s2) + 1. A run-time error occurs if strlen(s1) + strlen(s2) is greater than strmax(s2).

Example:

```
s := 'ABCDE';
STRINSERT('xy',s,4);
```

The value of s is now 'ABCxyDE'.

STRDELETE

The procedure STRDELETE(s,startpos,nchars) deletes nchars characters from string variable s starting at position startpos. If the value of integer expression nchars is less than 1, then no change is made in s and the values of the other arguments are not checked. Otherwise, a run-time error occurs if the value of integer expression startpos is less than 1 or if startpos + nchars-1 is greater than strlen(s).

Example:

```
s := 'ABCDE'; strdelete(s,3,2);
```

The value of s is now 'ABE'.

STRMOVE

The procedure STRMOVE copies characters from one part of a string or PAC expression into part of a string or PAC variable. STRMOVE requires five parameters:

```
STRMOVE(nchars, source, sourcepos, dest, destpos);
```

Nchars is an integer expression specifying the number of characters to be moved. If the value of nchars is less than 1, no change is made to the destination and no run-time errors occur regardless of the value of other parameters.

Source may be a string literal, a string expression, or a PAC expression.

Sourcepos is an integer expression specifying the first character in the source to be moved. A run-time error occurs if the value of sourcepos is less than 1. If source is a string, a run-time error occurs if sourcepos + nchars-1 is greater than strlen(source). If source is a PAC, a run-time error occurs if sourcepos + nchars-1 is greater than the upper bound of the PAC type.

Dest is either a string variable or a PAC variable.

Destpos is an integer expression specifying the index of the first character in dest to be changed. A run-time error occurs if the value of destpos is less than one.

If dest is a string, run-time errors occur if either destpos is greater than strlen(dest)+1, or if destpos + nchars-1 is greater than strmax(dest). If destpos + nchars-1 is greater than the current length of dest, then the current length of dest will be set to destpos + nchars-1.

If dest is a PAC, a run-time error will occur if destpos + nchars-1 is greater than the upper bound of the PAC type.

STRMOVE will properly handle the case of moving part of a string or PAC variable onto an overlapping part of itself.

Example:

```
VAR
     P :  packed array [1..10] of char;
          •
          •
          •
     P := 'ABCEDFGHIJ';
          strmove(3,'123',1,p,5);
```

The value of p is: 'ABCD123HIJ'.

STRLEN

The function STRLEN(s) returns an integer that is the current length of string expressions.

Examples:

```
STRLEN('') returns 0.
STRLEN('ABCDE' + 'FGHIJ') returns 10.
```

STRMAX

The function STRMAX(s) returns an integer that is the maximum declared length of string variable (s).

Example:

```
VAR
     S1 : string [100];
          •
          •
          •
     strmax (s1) returns 100.
```

STR

The function STR(s,startpos,nchars) returns a new string that is a copy of some portion of string expression (s). The string returned has a length equal to the value of integer expression nchars, and begins with the character s[startpos].

If nchars is less than 1, the string returned is the null string and no further checks are made on the values of the other parameters. Otherwise, a run-time error occurs if the value of integer expression startpos is less than 1, or if startpos + nchars-1 is greater than strlen(s).

Example:

```
str('ABCDE',4,2) returns 'DE'.
```

STRLTRIM

The function STRLTRIM(s) returns a new string that is a copy of string expression (s) with leading blanks removed.

Example:

```
strltrim ('     ABC     ') returns 'ABC     '.
```

STRRTRIM

The function STRRTRIM(s) returns a new string that is a copy of string expression (s) with trailing blanks removed.

Example:

```
strrtrim ('     ABC     ') returns '     ABC'.
```

STRREAD

The procedure STRREAD (s,startpos,nextpos,v1,...,vn) performs symbolic to internal conversion from the contents of string expression s into variables v1...vn. It is similar to the procedure READ (f,v1,...,vn) where f is a textfile. READ obtains its input characters from textfile f while STRREAD obtains its input from string expression s.

The string s is treated as a single line of a TEXT file with an implicit line marker at its end followed by an implicit end of file. The integer expression startpos indicates the starting character in s. After the operation, the integer variable nextpos will contain the index of the next component of s that would be read from. The conversion rules and allowable types for variables v1...vn are the same as for READ from TEXT files.

A run-time error will occur when STRREAD is called if the value of startpos is less than 1 or greater than STRLEN(s) + 1. A run-time error will occur if STRREAD attempts to access data beyond the implicit line marker at the end of the string s.

STRWRITE

The procedure STRWRITE (s,startpos, nextpos, p1,...,pn) performs internal to symbolic conversion from the values of write parameters p1...pn into the string variable s. It is similar to the procedure WRITE (f,p1,...,pn) where f is a text file. WRITE places its output characters into text file f while STRWRITE places its output characters into string variable s.

The integer expression startpos indicates the first character position in s where output characters will be placed. After the operation, the integer variable nextpos will contain the index of the next component of s that STRWRITE would access. The conversion rules and allowable forms for write parameters p1...pn are the same as the parameters to WRITE when applied to text files.

A run-time error will occur when STRWRITE is called if the value of startpos is less than 1 or greater than STRLEN(s) + 1. A run-time error will occur if the routine attempts to write more than STRMAX(s) characters into s.

STRRPT

The function STRRPT(s,n) returns a new string that is composed of n copies of string expression (s). The null string is returned if n is less than 1. A run-time error occurs if n * strlen(s) is greater than 255.

Example:

```
strrpt('ABC',3) returns 'ABCABCABC'.
```

STRPOS

The function STRPOS(s1,s2) searches string expression s2 for string expression s1 and returns the integer index of the beginning of the first occurrence of s1 within s2, or returns 0 if the string s1 was not found.

Example:

```
strpos('DE','ABCDEF') returns 4.
```

Dynamic Allocation and De-allocation Procedures

General Information

HOST Pascal allows variables to be created during program execution. The space, called the "heap", allocated to dynamic variables can then be de-allocated and later re-allocated to another variable. Dynamic

allocation and de-allocation 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. The standard procedure DISPOSE is normally used to de-allocate variables. In HOST Pascal, however, the procedure DISPOSE does not actually allow space occupied by the dynamic variable to be reused. The procedures MARK and RELEASE are used in order to reclaim space used for dynamic variables.

When it is known in advance that a group of dynamic variables may be needed on a short term basis, the state of the heap, before the short term variables are allocated, can be recorded by using the predefined procedure MARK. When the short term variables are no longer needed, the heap can be returned to the original condition by using the predefined procedure RELEASE. All variables allocated after the MARK are removed.

An attempt to allocate variables that require more space than available in the heap will cause an error message and aborting of the program.

The following paragraphs describe in greater detail the dynamic procedures NEW, DISPOSE, MARK, AND RELEASE.

NEW(p)

The procedure NEW is used to allocate memory space for a dynamic variable. (p) is a variable of type pointer. (p) can only point to a variable of a particular type T, and therefore is said to be bound to T.

When the procedure NEW(p) is called, a section of the heap large enough for a variable of type T is allocated and the address of that space is held in pointer (p).

If T is a record with variants, then the amount of space allocated is the amount required by the fixed part of the record, plus the amount required by the largest variant.

An alternative form of NEW can be used if type T is a record with variants. NEW(p,v1,...,vn), where v1...vn are constants used to select variants and subvariants of the record. The constants must be listed contiguously and in the order of their declaration. The amount of memory allocated is determined by the size of the variants selected. The tag field constant 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(p)

DISPOSE(p) indicates that the dynamic variable p^ is no longer needed. The value of the pointer p is set to NIL.

If the second form of NEW was used to allocate p^, then the alternate form of DISPOSE(p) must be used. DISPOSE (p,v1,...,vn). The tag field values should be identical to those used when the variable was allocated. A run-time error occurs if the value of p is NIL when DISPOSE is called.

DISPOSE does not allow the space used by p^ to be reused. In order to reclaim and reuse space previously occupied by dynamic variables, the standard procedures MARK and RELEASE must be used.

MARK(p)

MARK(p) is a predefined procedure having one parameter, a pointer variable, that records the HEAP state at the time MARK is executed. Calling MARK(p) causes assignment of the first free address in the HEAP to (p). The value of (p) may not change between MARK and RELEASE. Any execution of the procedure NEW will build new data structures, starting with the address held in (p).

RELEASE(p)

RELEASE is a predefined procedure having one parameter, a pointer variable, that restores the HEAP to the state present at the time of MARK(p). The value of (p) may not change between MARK and RELEASE. All dynamic variables created after MARK are effectively destroyed, and the memory space occupied by those variables is available for allocation to new dynamic variables. Be sure that no pointer variables point to dynamic structures created after the MARK procedure.

Transfer Procedures

PACK

PACK transfers components of an unpacked array to a packed array. The number of components in the unpacked array must be greater than or equal to the number of components in the packed array. Economy of memory is achieved by use of the procedure PACK.

PACK is used in the form: PACK (a,i,z), where:

```
a is of type ARRAY [m..n] of t;
i is of a type compatible with the index type of array a;
z is of type PACKED ARRAY [u..v] of t.
```

The procedure successively assigns the values of the elements of array a, starting with a[i], to the elements of array z, starting with z[u]. All elements of array z, i.e., z[u]..z[v], are assigned values of elements from array a.

The following example 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, given that:

```
a : Array [m..n] of t;
z : Packed array [u..v] of t;
```

where m-n >= u-v. Then PACK[a,i,z] means:

```
k := i;
for j := u to v do
     BEGIN
          z[j] := a[k];
          k := succ(k);
     END;
```

where k is a variable compatible with the index type of a, and j is a variable compatible with the index type of z.

UNPACK

The procedure UNPACK transfers components of a packed array to an unpacked array.

UNPACK is used in the form: 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 of a type compatible with the index type of array a.

The procedure successively assigns the values of the elements of array z, starting with z[u], to the elements of array a, starting with a[i]. All elements of array z, i.e., z[u]..z[v], are assigned to elements in array a.

The following example 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;
          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. 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 i has any other value, an error occurs when unpack attempts to index array z beyond the range of its index type.

In general, given that:

```
a : Array [m..n] of t;
z : PACKED ARRAY [u..v] of t;
```

where m-n >= u-v. Then UNPACK[a,i,z] means:

```
k := i;
for j := u to v do
     BEGIN
          a[k] := z[j];
          k := succ(k);
     END;
```

where k is a variable compatible with the index type of a, and j is a variable compatible with the index type of z.

Arithmetic Functions

There are eight predefined arithmetic functions in HOST Pascal. Each of these functions is passed in an arithmetic expression as a parameter and returns a numeric value.

In some cases 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 longreal values if integer values are passed to them. All of the functions return a longreal value when a real or longreal parameter is passed.

To compute the values of the functions, HOST Pascal uses system routines and compiler-defined algorithms.

Abs

`abs(X)` - Computes the absolute value of X.

Sqr

`sqr(X)` - Computes the value of X squared.

Sqrt

`sqrt(X)` - Computes the square root of X. If X < 0 then a run time error occurs.

Exp

`exp(X)` - Computes e (base of the natural logarithms) to the power of X.

Ln

`ln(X)` - Computes the natural logarithm of X. If X < 0 then a run time error occurs.

Sin, Cos

sin(X), cos(X) - Computes the sine and cosine of X, in radians.

Arctan

arctan(X) - Computes the arctangent of X, in radians.

Predicates

The following three procedures return Boolean results.

Odd

odd(X) The procedure odd returns TRUE if the value of the integer expression X is odd, FALSE otherwise.

Eof

eof, or eof(f) where (f) is a file that has previously been declared. The procedure eof(f) returns TRUE if the file F is not open, F is open in a writable mode, or F is open for reading and the file is positioned beyond last existing component. If the parameter F is omitted, the standard file INPUT is assumed.

Eoln

eoln, or eoln(f) where (f) is a text file that has previously been declared and opened in the readable mode. The procedure eoln(f) returns TRUE if the text file f is positioned at the end of a line. If the parameter f is omitted, the file INPUT is assumed.

Transfer Functions

Trunc

trunc(X) - The function trunc returns an integer result that 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
```

Round

round(X) - 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
```

Ordinal Functions

The ordinal functions are: ord, chr, succ, and pred.

Ord

ord(X) - where X is an expression of ordinal type. The function ord returns the ordinal number associated with the value of X. If the result can be contained in one word, a one-word result is returned, otherwise a two-word result is returned. If the parameter is compatible with INTEGER, then the parameter 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 ord value of -1 is -1; the ord value of 1000 is 1000. The ord value of 'a' is 97, the ord value of 'A' is 65.

The predefined type Boolean, for example, is defined:

TYPE BOOLEAN = (false, true) and therefore ord(false) returns 0, and 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); the ord(red) returns 0, ord(blue) returns 1, and ord(yellow) returns 2.

Chr

chr(X) - where X is an integer expression. The function chr returns the character value whose ordinal number is equal to the value of the integer expression X. If the RANGE compiler option is ON, a run-time error will occur if the value of X is outside the range 0..255. If the RANGE option is OFF, a character will be formed by zeroing all but the least significant 8 bits of the value of X. For any character ch, the following is true: chr(ord (ch)) = ch.

Examples:

The value of 63 returns the chr '?', the value 100 returns the chr 'd', the value 13 returns the chr 'carriage return', the value 75 returns the chr 'K'.

Succ

succ(X) - where X is an expression of ordinal type. The function succ returns a result having an ordinal one greater than 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. Given the declaration: TYPE color = (red, blue, yellow); succ (red) returns blue, and succ (Yellow) returns a value that is not of type color. If X is 1, then succ(X) is 2. If X is -5, then succ(X) is -4. If X is 'a', then succ(X) is 'b'. If X is false, then succ(X) is true.

Pred

pred(X) - where X is an expression of ordinal type. The function pred returns a value having an ordinal value one less than 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, and pred(monday) returns a value that is not of type day.

Examples:

The pred(1) is 0, the pred(-5) is -6, the pred('b') is 'a', the pred(true) is false.

Numeric Conversion Functions

HEX(s)
OCTAL(s)
BINARY(s)

HEX, OCTAL, and BINARY are functions that take the string expression s and return an integer. The string is interpreted as a hexadecimal, octal, or binary number. Except for leading and trailing blanks, all characters in the string must be legal digits in the indicated base. Blanks embedded between the digit characters are not allowed.

File Handling Functions

The functions LINEPOS, and POSITION enable the program to determine the current position in a file relative to the beginning-of-file and end-of-line.

LINEPOS

The function LINEPOS(f) is applicable if (f) is a previously opened text file. The function returns, as an integer, the number of characters read from or written to the text file (f) since the last line marker. This does not include the character in f^.

The meaning of LINEPOS is altered if (f) is open to the I/O device "RS232". In the case where (f) is open for reading to RS232, LINEPOS returns the number of characters that are presently in the RS232 receiver circular buffer. In the case where (f) is open for writing to RS232, LINEPOS returns the number of characters that are in the RS232 transmitter circular buffer plus the number of characters contained in the transmitter hardware registers. See the discussion in Appendix C for further information.

POSITION

The function POSITION returns, as an integer, the index of the current component of the file(f) starting with 1. This component is the next to be accessed by read or write. The file(f) must have previously been opened. File(f) may not be a text file. If the buffer f^ is full, the result is the index of that component.

IORESULT

The function IORESULT, used in conjunction with the compiler directive $IOCHECK OFF$, allows input/output exceptions to be handled by the program.

IORESULT is an extension to HP Standard Pascal and requires the compiler directive $EXTENSION ON$ to be in effect.

IORESULT returns an integer that is the result code of the last input/output operation. The meaning of the integer returned is shown in Table 7-1.

Table 7-1. IORESULT Definitions

Integer	Definition
0	No error
1	End of file
2	Invalid disc number
3	File not found
4	File already exists
5	No disc space available
6	No directory space available
7	File linkage is corrupt
8	Read/Write-Only device opened in wrong mode
9	Non-textfile opened text-type device
10	Two files opened the same disc file or device
11	Illegal syntax reading integer or real
12	Illegal value reading integer or real
13	File is not open in proper mode
14	Ill formed file name
The following result codes are returned only when reading from the I/O device "RS232".	
15	Timeout interval has elapsed without receiving character
16	Circular buffer overflow; data has been lost
17	Line break (i.e., continuous spacing) signal received
18	Character framing error (i.e., stop bit was 0)
19	Character overrun error (interrupts were not serviced fast enough)
20	Character parity error

Chapter 8

Implementation of HOST Pascal

Introduction

A practical knowledge of the implementation of HOST Pascal is useful for efficient programming. This chapter describes data allocation, memory configuration, data and stack management, HEAP management, and efficient programming.

Data Allocation

The HOST Pascal compiler converts source code into pseudo code instructions and data definitions. The space reserved by the data definitions is used to represent 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 is given data allocation that optimizes the use of memory space. An unpacked data type is given an allocation that allows faster data access.

This section describes the size in bits and words of the data allocation for a variable or structured constant of a particular type and the boundary alignment conventions for that allocation.

Allocations for structured constants are identical to the allocations for variables of the same type as the structured constant.

Allocation for Scalar Variables

The allocations for variables of scalar, subrange, and pointer types is shown in Table 8-1. All allocations begin on word boundaries.

Table 8-1. Allocations for Scalar Variables

Type	Size	Notes
BOOLEAN	1 word	FALSE is represented by 0. TRUE is represented by 1.
INTEGER	2 words	Bit 15 of the first word is the sign bit.
Subrange of INTEGER	1 or 2 words	Subranges contained in -32768..32767 require 1 word to represent variables of that type. All other subranges require 2 words to represent variables of that type. Examples: Subrange — Allocation 0..8 — 1 word -32768..32767 — 1 word 10..40000 — 2 words -70000..-1 — 2 words
Enumeration	1 word	The values are represented internally as 1-word integers in the range 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 byte (the left byte contains 0).
Pointer	1 word	

Allocation for Structured Variables

The allocation for variables of ARRAY, RECORD, FILE, and SET types is shown in Table 8-2. All allocations begin on word boundaries.

Table 8-2. Allocations for Structured Variables

Type	Size
ARRAY	The size of an array allocation is the sum of the allocations of its elements: (product of cardinalities of index types) * (allocation of one element) The elements are stored in row major order.
RECORD	The size of a record allocation is the sum of the allocations of the fixed part and, if any, the allocations of the tag field and the largest variant.
FILE	The FILE variable needs memory in two areas: a. Space for file variable. b. Space for the Device Control Block (DCB). If the file is a textfile, the variable allocation is 131 words. If the file is not a textfile, the allocation is 2 words + size of the file component type. Assignment of a buffer (DCB) area is made when the block in which the file is declared becomes active. The DCB area is taken from a group of buffer areas allocated when the program was executed. The size of a DCB is approximately 40 words + (128 * b) where b can be 1, 2, 4, 8, or 16. The value of b is determined by the value of the BUFFERS compiler option when the file is declared. Default value of BUFFERS is 1.
SET	The maximum number of elements possible in a set is 4080. The allocation for a set is (n + 16) DIV 16, where n is the ordinal value of the largest element in the set, and the allocation is in words.

Allocation for Elements of Packed Structures

Arrays, records, sets, and files may be packed by prefixing the type definition with 'PACKED'. This indicates to the compiler that the non-structured elements of the type are to be allocated in a memory conserving format. In general, the following guidelines are used in the interest of data 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. (The array may be packed, however, by prefixing the array definition with 'PACKED'.)

PACKED ARRAY [1..6,1..3] of T, however, is equivalent to PACKED ARRAY [1..6] of PACKED ARRAY [1..3] of T.

Table 8-3. Allocations for Elements of Packed Structures

Type	Allocation	
BOOLEAN	Size: Alignment:	1 bit Bit boundary
INTEGER	Size: Alignment:	2 words Word boundary
Subrange of INTEGER	Size:	-32768 <= lower bound..upper bound <= 32767 requires no more than 1 word. If the high bound <= 32767 and if the low bound >= 0, the packed subrange of integer will occupy the minimum number of bits required to represent the highest value of the subrange. For example: 0..7 requires 3 bits, 0..8 requires 4 bits, 0..31 requires 5 bits, 25..31 requires 5 bits, 0..32767 requires 15 bits,
	Alignment:	Bit boundary

Table 8-3. Allocations for Elements of Packed Structures (Cont'd)

Enumeration	Size: Alignment:	Minimum number of bits necessary to represent the value (cardinality of type - 1) Bit boundary
Subrange of Enumeration	Size: Alignment:	Minimum number of bits necessary to represent the value (cardinality of subrange - 1) Bit boundary
REAL	Size: Alignment:	2 words Word boundary
LONGREAL	Size: Alignment:	4 words Word boundary
CHAR	Size: Alignment:	1 byte (8 bits) Bit boundary
Pointer	Size: Alignment:	1 word Word boundary
Set	Size: Alignment: Size: Alignment:	n bits, n <= 16 Bit boundary where n is the cardinality of the base type. n bits, 16 < n <= 4080 The maximum number of elements possible in a set is 4080. The allocation for a set is (n + 16) DIV 16, where n is the number of bits in the set. Word boundary

Examples of Packed and Unpacked Structure Allocation

Example 1: Unpacked structure allocation

```
TYPE
    SUIT = (club, diamond, heart, spade);

VAR
    r : RECORD
         a : INTEGER;
         b :  1..13;
         c :  SUIT;
         d :  REAL;
         e :  CHAR;
         f :  'A'..'Z';
         g :  BOOLEAN;
         h :  LONGREAL;
         i :  SET OF SUIT;
         j :  ARRAY [SUIT] OF 1..13;
         CASE t :  BOOLEAN OF
              true:  (k,l,m : CHAR);
              false: (n     : INTEGER);

         END;
```

Variable r is allocated as follows:

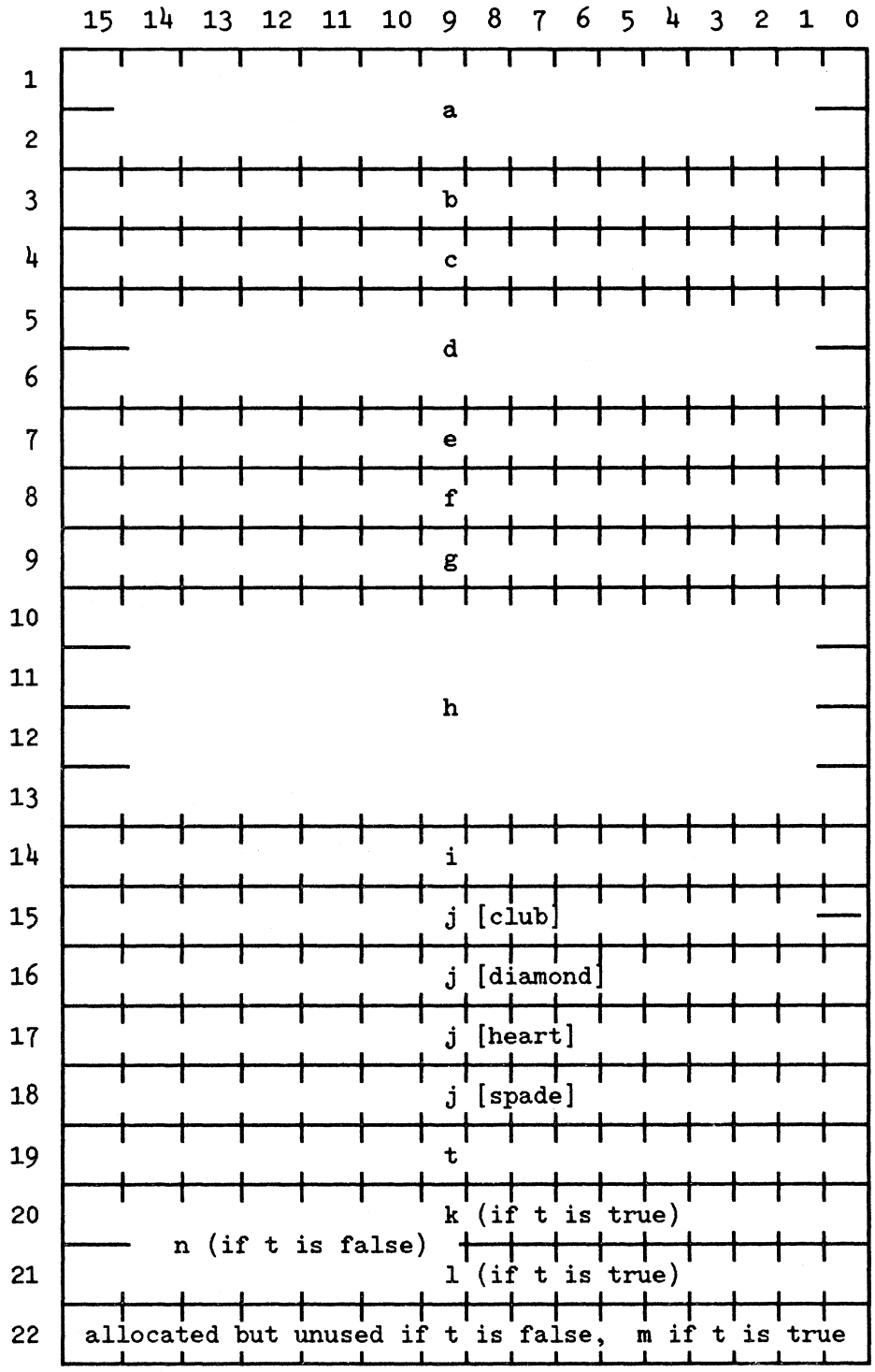

Figure 8-1. Unpacked Structure Allocation

Example 2: This example shows a packed record allocation. Note, however, 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;
```

Variable r is allocated as follows:

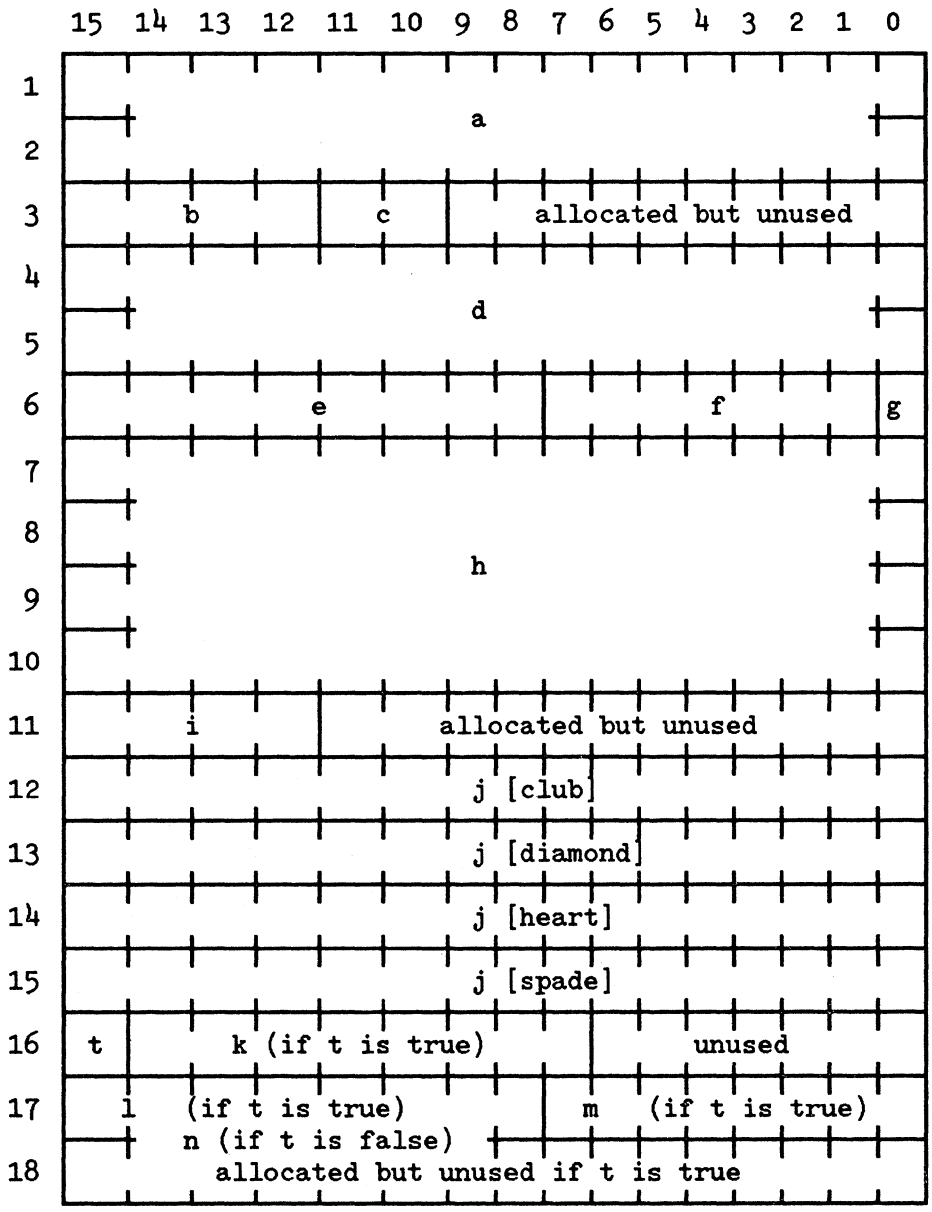

Figure 8-2. Packed Record Allocation With Unpacked Array

Note that the elements of array j are not packed, but the array as a whole is treated as a field of the packed record.

Example 3: This example shows a packed record allocation, including field j as a packed array.

```
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:

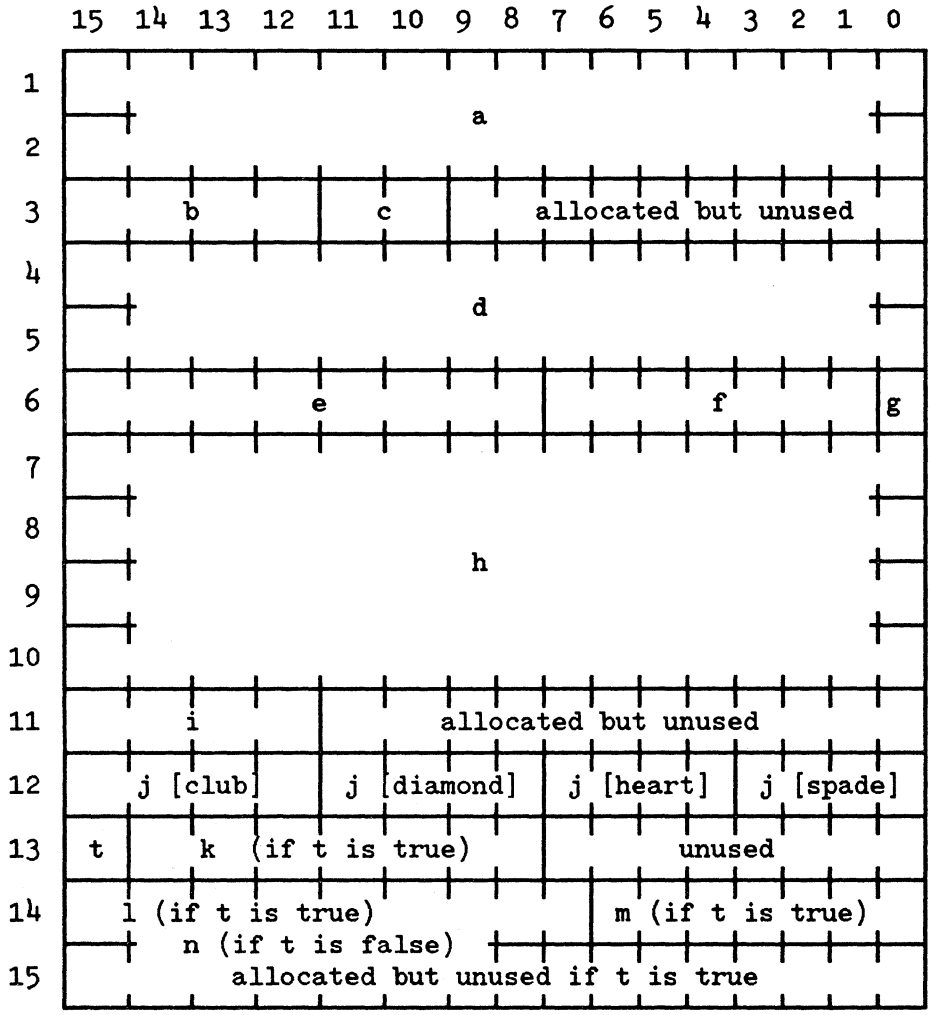

Figure 8-3. Packed Record Allocation With Packed Array

Memory Allocation

The memory layout employed in the HOST Pascal system is composed of a DCB area, an upper STACK/HEAP area, and a lower STACK/HEAP area if the Memory Expansion Module is present in the logic system.

The DCB area varies in size, based on the number of files declared in the program and the value of the BUFFERS option at the time of file declaration. DCB area is allocated when a program is run and does not change in size during program execution.

One DCB is pre-allocated at the beginning of run-time for each file variable declared in the program. It is possible, through recursive calls to procedures or functions that contain local file variables, to exceed the number of DCB's that have been preallocated. In this case, a run-time error, "UNABLE TO ALLOCATE FILE BUFFER", will occur.

The upper stack/heap area accommodates 25k bytes of memory minus the area allocated to DCB. The lower stack/heap area accommodates an additional 32k bytes of memory.

The stack contains p_code and static variables; the heap contains dynamic variables allocated with the standard procedure NEW. Implementation of the stack begins with the high address and increases in size toward the low address. The heap is implemented at the low address and increases in size toward the high address. An overlapping of stack and heap areas is possible and causes an error, "OUT OF MEMORY. CAN'T EXPAND STACK OR HEAP."

Each activation of a procedure or function creates a new space for the static data area of the procedure or function. The procedure or function code is never duplicated.

I/O Error Handling

I/O errors cause termination of program execution. Processing of I/O errors can be implemented by using the compiler option IOCHECK and the standard function IORESULT.

IORESULT is an extension to the standard Pascal, and its use is enabled by the EXTENSIONS compiler option. IOCHECK OFF disables the generation of code that checks for I/O errors. IORESULT returns an integer that is the result code from the previous I/O operation. The value returned by IORESULT is defined in Table 7-1.

```
$IOCHECK OFF$
RESET(f);
IF IORESULT <> 0
     THEN
     BEGIN {exception handling}
          •
          •
          •
     END;
```

Chapter 9

Using HOST Pascal

The Source File

The Pascal compiler takes as input a program source file created with the 64000 Editor function. The basic form of a program source file is as follows:

```
"HOST"

PROGRAM Name (parameters);

{Declarations}
    •
    •
    •

BEGIN {program}
    •
    •
    •
END.  {program}
```

The first line of the source file must be the special compiler directive "HOST". Then follow the declaration and statement sections of the program.

Compiling the Source File

After the source file has been edited, and before the program can be executed, the source file must be compiled. Pressing the soft key labeled [compile], and adding the source file name, causes translation of the source file into an absolute file. The absolute file contains a pseudo-machine code version of the source file. The source file name is given also to the absolute file. After the "compile" command has been entered, some choices must be made by the operator. The "compile" command syntax is shown in Figure 9-1.

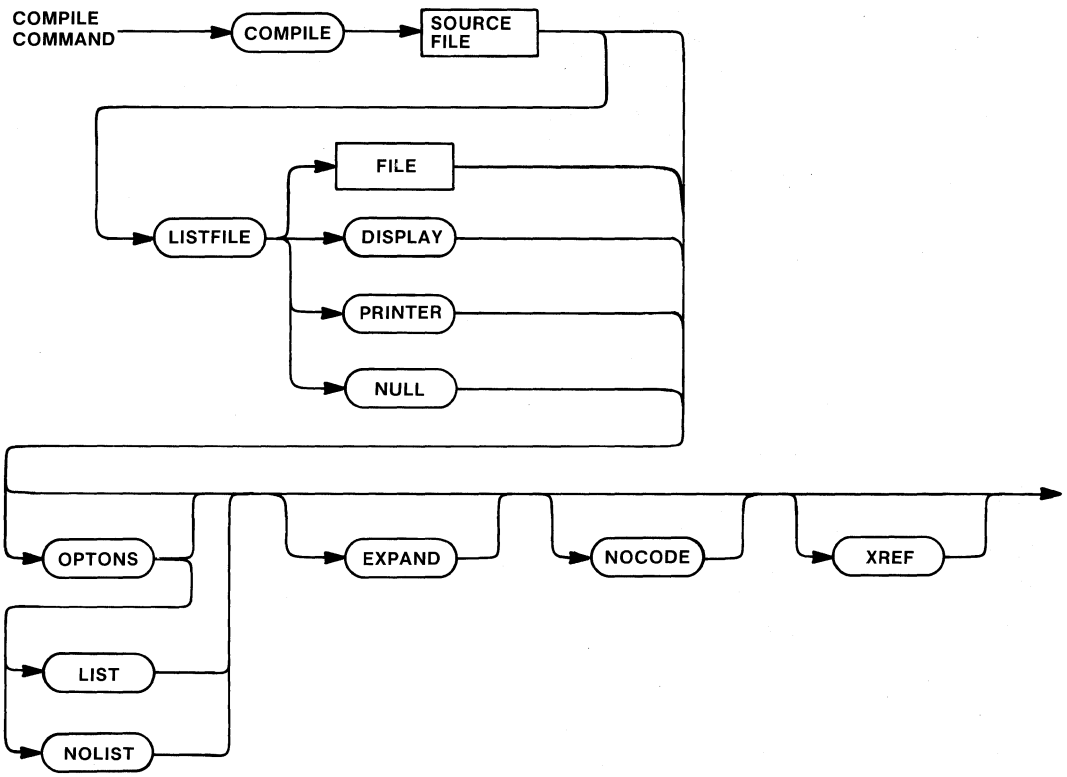

Figure 9-1. Compile Command Syntax

Command Parameters

Source File - The name of the source file to be compiled. The syntax for the source file name is shown in Figure 9-2.

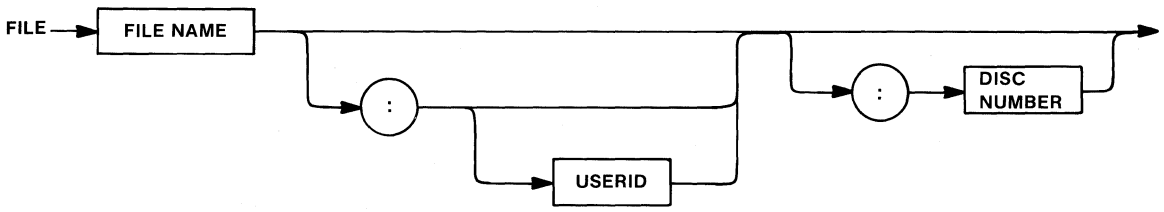

Figure 9-2. Source File Name Syntax

where:

file name is comprised of 1 to 9 alphanumeric characters beginning with an upper case alphabetic character.

userid — is comprised of 1 to 6 alphanumeric characters beginning with an upper case alphabetic character. If userid is omitted, the present userid will be the default value.

disc # — represents the logical unit number of the system disc. Only decimal numbers are allowed. If disc # is omitted, all discs will be searched for the file.

The absolute file will be written onto the same disc from which the source file was taken. If another absolute file has the same name as the current absolute file, the old file will be overwritten by the current file.

The pseudo machine code will not be written into the absolute file if errors are detected during the compiling of the source file, or if the nocode option is specified, or if any CODE OFF directives are encountered in the source file.

Listfile - If the listfile keyword is not used, then the default is the predefined listfile for the present userid. If no predefined listfile exists, a null listfile is the default. If the listfile keyword is used, one of the following must also be specified:

<FILE>
display
printer
null

The syntax for <FILE> is the same as for source file. The type of the listing file is "listing". If no disc # was specified, then the listing file will be written to the disc from which the source file was taken.

Options - If the options keyword is used, then one or more of the following may be specified:

list
nolist
expand
nocode
xref

list — List specifies a complete source code listing with error messages,

nolist — Nolist specifies that only error messages will be listed. All LIST directives in the source code are ignored.

expand — The expand option has no effect.

nocode — specifies that no pseudo code file will be written. CODE directives in the source file will be ignored.

xref — specifies a cross reference listing of the source file. All XREF directives in the source file are ignored.

Running HOST Pascal Programs

The "run" command causes execution of the pseudo-machine code in the absolute file.

When a HOST Pascal program is executed, a file name may be associated with each program parameter. A program parameter is a file variable that was listed in the program heading of the Pascal program. The file name is the name of a disc file or an I/O device.

File names for the standard files INPUT and OUTPUT may be specified in the context of the run command. For other program parameters, the system prompts the user for a file name for each additional program parameter. The run command syntax is shown in Figure 9-3.

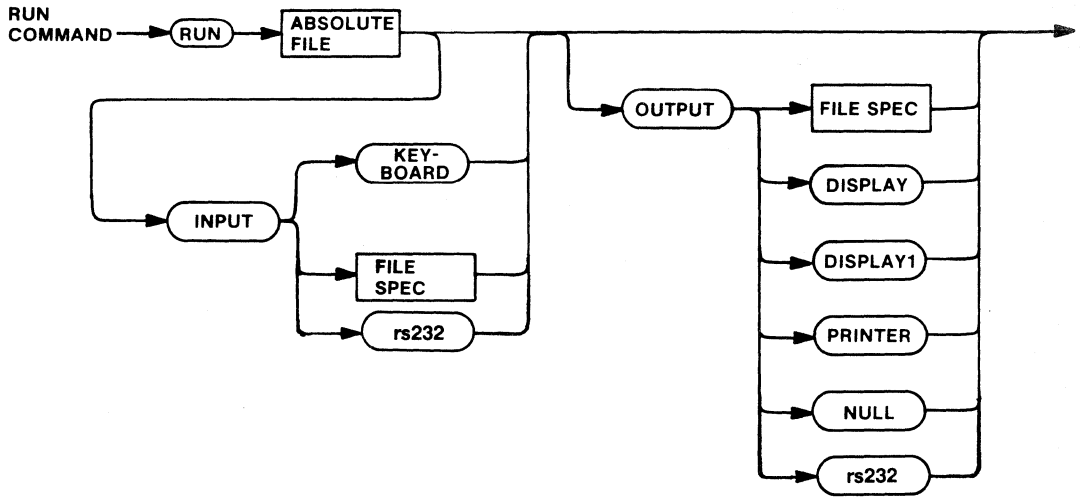

Figure 9-3. Run Command Syntax

Run Command Parameters

<file> — the name of the absolute file containing pseudo-machine code. The syntax for <file> is: <file_name> [:<userid>][:<disc #>].

input — if the input keyword is not used, then the file name associated with INPUT is keyboard.

if the input keyword is used, one of the following must be given:

keyboard
<file spec>

the syntax for <file spec> is:

<file name>[:<userid>][:disc #][:<file type>]

where:

file name is 1 to 9 alphanumeric characters beginning with an upper case alphabetic character.

userid is 1 to 6 alphanumeric characters beginning with an upper case alphabetic character.

disc # is a decimal number.

file type is source, relocatable, absolute, link_com, emul_com, prom, asmb_sym, link_sym, comp_sym, trace, data, asym_db, or comp_db. The default file type is source.

output If the keyword output is not used, then the default file name for OUTPUT is the default listing file. If no default listing file has been specified for the present userid, then the default listing file is null.

If the keyword output is used, then one of the following must be given:

<file spec>
display
display1
printer
rs232
null

The syntax for <file spec> is the same as for input. If the file type is omitted in the file spec for OUTPUT, then the default type is listing.

Additional Program Parameters

If the program about to be executed has program parameters other than INPUT or OUTPUT, then a prompt will be issued for each of the parameters in the order in which they were listed in the program heading. The prompt looks like the following example:

Enter the file name for <identifier>.

<identifier> is the variable identifier from the Pascal program. One may then enter one of the following:

<file spec>
display
display1
keyboard
printer
rs232
null

The syntax for <file spec> is the same as shown with the input keyword. If file type is not specified, then the default file type is source, if the file is a text file, or data, for a non-text file.

If no entry is made following the prompt, the system treats the file as if it were not a program parameter.

Appendix A

Compile-Time and Run-Time Error Messages

Compile-Time Error Messages

```
  1: Error in simple type
  2: Identifier expected
  3: 'PROGRAM' expected
  4: ')' expected
  5: ':' expected
  6: Illegal symbol
  7: Error in parameter list
  8: 'OF' expected
  9: '(' expected
 10: Error in type
 11: '[' expected
 12: ']' expected
 13: END expected
 14: ';' expected
 15: Integer expected
 16: '=' expected
 17: 'BEGIN' expected
 18: Error in declaration part
 19: Error in field list
 20: '.' expected
 21: '*' expected
 22: LABEL expected
 23: CONST, TYPE, VAR, BEGIN, FUNCTION or PROCEDURE expected
 24: EOF expected
 25: Statement begin symbol expected
 26: PROCEDURE or FUNCTION expected
 27: ';' or OTHERWISE expected
 28: '(' or '[' expected
 29: String expected
 30: Type name expected
 31: '..' expected
 32: Error in variant label
 50: Error in constant
 51: ':=' expected
 52: 'THEN' expected
 53: 'UNTIL' expected
 54: 'DO' expected
 55: 'TO/DOWNTO' expected
 58: Error in factor
 59: Error in variable
101: Identifier declared twice
102: Low bound exceeds high bound
103: Identifier is not of appropriate class
104: Identifier not declared
105: Sign not allowed
106: Number expected
```

107: Incompatible subrange types
108: File not allowed here
109: Type must not be REAL
110: Tag field type must be ordinal or subrange
111: Incompatible with tag field type
112: Index type must not be REAL
113: Index type must be ordinal or subrange
114: Base type must not be REAL
115: Base type must be ordinal or subrange
116: Error in type of standard procedure parameter
117: Unsatisfied forward reference
118: Forward reference type identifier in variable declaration
119: Forward declared: repeated parameter list not identical
120: Function result type must not be/contain file
121: File value parameter not allowed
122: Forward declared function; repeated result type not identical
123: Missing result type in function declaration
125: Error in type of standard procedure/function parameter
126: Number of parameters does not agree with declaration
127: Illegal parameter substitution
128: Result type of parameter function does not agree with declaration
129: Type conflict of operands
130: Expression is not of set type
131: Tests on equality allowed only
132: Strict inclusion not allowed
133: File comparison not allowed
134: Illegal type of operand(s)
135: Type of operand must be boolean
136: Set element type must be ordinal or subrange
137: Set element types not compatible
138: Type of variable is not array
139: Index type is not compatible with declaration
140: Type of variable is not record
141: Type of variable must be file or pointer
142: Illegal parameter substitution
143: Illegal type of loop control variable
144: Illegal type of expression
145: Type conflict
146: Assignment of files not allowed
147: Label type incompatible with selecting expression
148: Subrange bounds must be ordinal
149: Index type must not be integer
150: Assignment to standard function is not allowed
151: Assignment to formal function is not allowed
152: No such field in this record
153: Type error in read
154: Actual parameter must be a variable
155: Control variable must be declared locally
156: Multidefined case label
158: Missing corresponding variant declaration

159: Tag field must be ordinal
160: Previous declaration was not forward
161: Again forward declared
162: Parameter size must be constant
163: Missing variant in declaration
164: Substitution of standard proc/func not allowed
165: Multidefined label
166: Multideclared label
167: Undeclared label
168: Undefined label
169: Error in base set
170: Value parameter expected
171: Standard file was redeclared
172: Undeclared external file
174: Pascal procedure or function expected
175: Missing file INPUT in program heading
176: Missing file OUTPUT in program heading
177: Assignment to function identifier not allowed here
178: Multidefined record variant
179: Parameter list of actual proc/func does not match formal declaration
180: Control variable must not be formal
181: Constant part of address out of range
182: Actual VAR parameter must not be packed
183: Assignment to field in non-variable record
184: 'STRING' not allowed in this context
185: Variant may not contain file
186: Expression must be compatible with INTEGER
187: Illegal assignment to FOR control variable
188 Value of preceding function not assigned.
201: Error in real constant: digit expected
202: String constant must not exceed source line
203: Integer constant exceeds range
205: Zero string not allowed
206: Integer part of real constant exceeds range
207: Compiler option must not exceed source line
208: Illegal character in this context
209: Bad data for HEX, OCTAL, or BINARY
230: Structured type identifier expected
250: Too many nested scopes of identifiers
251: Too many nested procedures and/or functions
252: Too many forward references or procedure entries
253: Procedure too long
255: Too many errors on this source line
256: Too many external references
257: Too many externals
260: Too many exit labels
261: Too many nested INCLUDEs
262: Can't open INCLUDE file
263: Too many elements in array
264: Size of structure may not exceed 16383 words
265: Procedure's data area exceeds size limit
266: Case label must be between -32768 and 32767

267: Total segment code exceeds 32767 bytes
268: More than 16 program parameters
269: Sets are restricted to ordinal range 0..4079
270: Number of procedures and structured constants exceeds 255
271: Number of SEGMENTs exceeds 15
300: Division by zero
301: No case provided for this value
302: Index expression out of bounds
303: Value to be assigned is out of bounds
304: Element expression out of range
305: Illegal operator in constant expression
306: Constant expression causes integer overflow
307: Illegal operand type in constant expression
308: Illegal function in constant expression
309: Set element not in range of set base type
310: Structured constant not allowed here
311: Too many constant array elements specified
312: Too few constant array elements specified
313: Constant value inaccessible due to $NOSAVE$
314: Illegal to select elements in constant expression
315: Constant NIL dereferenced
316: Constant record element respecified
317: Variant field specified before tag field
318: Field doesn't belong to previously selected variant
319: All fields of record constant have not been specified
320: Variable which is program parameter must be file
321: String literal is too long
322: a MOD b, b <= 0
397: Error in compiler, contact HP representative
398: Implementation restriction
402: End of source before end of compilation
403: Symbol table overflow; delete symbols; parsing stopped
404: Semantic stack overflow; break up program; parsing stopped
409: Label may not have more than four digits
411: Constant expression expected
413: Assignment to constant
414: More than 255 subroutines
455: Language extensions used in extensions off mode
500: Warning: illegal compiler option; option ignored
502: Warning: Option ignored. Specify before PROGRAM.
503: Warning: Source line exceeds allowed length
504: Warning: non-standard feature used
508: Warning: wide range of CASE labels generates large amounts of code
509: Warning: illegal value for compiler option
510: Warning: $SEGMENT$ not specified before FORWARD. Option ignored.

Run-Time Error Messages

A diagnostic message is displayed on the 64000 Logic Development station screen when a run-time error occurs. The information in the message can be used, together with the program listing, to locate and correct the offending area of the program.

Following is an example program that generates a run-time error. Included, also, is a brief explanation of the error locating procedure.

Example:

```
FILE:TNILREF:PTEST:0      HP 64000 HOST PASCAL

     1  1    1:D      1  "HOST"
     2  1    1:D      1  PROGRAM TNILREF;
     3  1    1:D      1  TYPE
     4  1    1:D      1     PAC10 = PACKED ARRAY [1..10] OF CHAR;
     5  1    1:D      1     PACPTR = ^PAC10;
     6  1    1:D      1  VAR
     7  1    1:D      1     PTR: PACPTR;
     8  1    1:D      2     I: INTEGER;
     9  1    1:C      0  BEGIN
    10  1    1:C      0  PTR := NIL;
    11  1    1:C      3  FOR I := 2 TO 10 DO
    12  1    1:C     24     PTR^[I] := ' ';
    13  1    1:C     68  END.

End of compilation, number of errors=    0
```

Legend

The first column indicates the source file line numbers.

The second column indicates the segment number. In this case segment 1 is listed.

The third column indicates the procedure number, in this case procedure number 1 is shown. D designates the declaration part; C designates the code (or executable) part of block.

The fourth column indicates offset (in words) of the memory location of the variable I in the block local datum area. (Line 8 of the file shows 2 words of offset.)

The fourth column also indicates P-code offset (in bytes) of code for the statement that follows. (Line 12 of the file shows 33 bytes of offset.)

The example program TNILREF contains a reference to a pointer variable that contains the value NIL. When executed, TNILREF will produce the following error message:

```
Error 11 in program TNILREF:PTEST, seg. #1, proc. #1, p-code offset = 37

    "NIL  POINTER  REFERENCE"
```

The error occurred in segment number 1, procedure number 1 with the p-machine's instruction pointer equal to 37. Examining the listing allows us to determine that procedure number 1 is the main program body, and that the error occurred in line number 12.

The code generated by line 12 begins at offset 24, and the code generated by line 13 begins at offset 68. By deduction, then, offset 37 occurs in line 12.

Following is a listing of run-time error messages:

```
VALUE OUT OF RANGE
STANDARD PROCEDURE/FUNCTION PARAMETER OUT OF RANGE
SET ELEMENT OUTSIDE OF RECEIVING SET'S RANGE
OUT OF MEMORY. CAN'T EXPAND STACK OR HEAP
INTEGER OVERFLOW
DIVISION BY 0
A MOD B OPERATION, B < 0
NIL POINTER REFERENCE
NO LABEL SPECIFIED FOR CASE EXPRESSION VALUE
FLOATING POINT OVERFLOW
INVALID OPERAND FOR FLOATING POINT EXPRESSION
FLOATING POINT DIVISION BY 0
STRING TOO LONG FOR RECEIVING STRING VARIABLE
INDEX INTO STRING OUTSIDE OF CURRENT LENGTH
ATTEMPT TO ACCESS PROTECTED MEMORY
ILLEGAL DATA FOR HEX, OCTAL, OR BINARY
UNABLE TO ALLOCATE FILE BUFFER
CORRUPT DATA IN FILE VARIABLE
PROCEDURE HALT EXECUTED
ILLEGAL OPERATION ON "rs232"
<FILE NAME> TIMEOUT INTERVAL ELAPSED
<FILE NAME> RECEIVE BUFFER OVERFLOW
<FILE NAME> BREAK RECEIVED
<FILE NAME> CHARACTER FRAMING ERROR
<FILE NAME> CHARACTER OVERRUN ERROR
<FILE NAME> CHARACTER PARITY ERROR
<FILE NAME> READ/WRITE-ONLY DEVICE OPENED IN WRONG MODE
<FILE NAME> DEVICE CANNOT BE ASSIGNED TO NON-TEXT FILE
<FILE NAME> FILE NAME OPENED BY TWO FILES
<FILE NAME> ILLEGAL SYNTAX FOR READ VARIABLE
<FILE NAME> ILLEGAL VALUE FOR READ OPERATION
FILE NOT OPEN IN PROPER MODE
ILL FORMED FILE NAME
END OF FILE      FILE = <FILE NAME>
ILLEGAL DISC     FILE = <FILE NAME>
FILE NOT FOUND     FILE = <FILE NAME>
DISC FULL     FILE = <FILE NAME>
DIRECTORY FULL    FILE = <FILE NAME>
INVALID OPERATION.  P-CODE IS CORRUPT
```

Appendix B

Sample Pascal Programs

The following is a listing of HOST Pascal statements, and the examples in which the statements are shown.

```
Assignment:  all examples;
Case:  11,14,15,25,27;
Enumerated Type:  25;
For:  5,6,8,10,13,14,17,18,19,20,21,24,26,27,28;
Function:  23,24;
Goto Label:  12,13;
If:  8,9,10,12,13,14,16,20,21,26,27;
Procedure:  19,20,21,22,23,24,27;
Read or Readln:  all examples;
Record:  29;
Repeat:  3,7,9,15,16,26,27;
Reset(f):  27;
Rewrite(f):  27;
While:  4,7,12;
While Not:  27;
With: 29;
Write or Writeln:  all examples;
Write parameters:  18,19,22,23,24;
```

Sample Programs

If any of the sample programs are to be attempted, the program must first have the directive "HOST" or compilation will not be possible.

1. Adding two given integers and printing the results.

```
"HOST"
PROGRAM REG1 (INPUT,OUTPUT);
VAR  A,B,C : INTEGER;

BEGIN
     READ(A,B);
     C := A + B;
     WRITELN (C)
END.
```

2. Calculate automobile operating cost.

```
"HOST"
PROGRAM REG2 (INPUT,OUTPUT);
CONST  ROADTAX = 50.0
VAR  GARAGEBILL, INSURANCE, TOTAL ; REAL;

BEGIN
     READ (GARAGEBILL,INSURANCE);
     TOTAL := GARAGEBILL + INSURANCE + ROADTAX;
     WRITELN ('TOTAL COST IS' TOTAL)
END.
```

3. Repeat two statements until true.

```
"HOST"
PROGRAM REG3 (INPUT,OUTPUT);
VAR  SUM,NUMBER : INTEGER;

BEGIN
     SUM := 0;
     REPEAT
          READ (NUMBER);
          SUM := SUM + NUMBER
     UNTIL NUMBER = 0;
     WRITELN ('THE TOTAL IS', SUM)
END.
```

4. Calculate the largest power of a number that is less than or equal to another number.

```
"HOST"
PROGRAM REG4 (INPUT,OUTPUT);
VAR  A,B,PRODUCT : REAL;
     POWER       : INTEGER;

BEGIN
     READ (A,B);
     POWER := 0;
     PRODUCT := B;
     WHILE PRODUCT <= A DO
     BEGIN
          POWER := POWER + 1;
          PRODUCT := PRODUCT * B;
     END;
     WRITELN ('LARGEST POWER OF',B,'<'.A,'IS',POWER)
END.
```

5. Form the average of N numbers.

```
"HOST"
PROGRAM REG5 (INPUT,OUTPUT);
VAR  I,N                  : INTEGER;
    AVERAGE,SUM,NUMBER : REAL;

BEGIN
     READ (N);
     SUM := 0;
     FOR I BECOMES 1 TO N DO
     BEGIN
          READ (NUMBER);
          SUM := SUM + NUMBER
     END;
     AVERAGE := SUM/N;
     WRITELN ('THE AVERAGE IS",AVERAGE)
END.
```

6. Form the average of each of M groups of numbers.

```
"HOST"
PROGRAM REG6 (INPUT,OUTPUT);
VAR  I,J,M,N               : INTEGER;
     AVERAGE, SUM, NUMBER : REAL;

BEGIN
     READ (M);
     FOR J := 1 TO M DO
     BEGIN
          READ (N);
          SUM := 0;
          FOR I := 1 TO N DO
          BEGIN
               READ (NUMBER);
               SUM := SUM + NUMBER
          END;
          AVERAGE := SUM/N;
          WRITELN ('THE AVERAGE IS', AVERAGE)
     END
END.
```

7. Find HCF of two numbers.

```
"HOST"
PROGRAM REG7 (INPUT,OUTPUT);
VAR  A,B : INTEGER;
BEGIN
     READ (A,B);
     REPEAT
          WHILE A > B DO
          A := A - B;
          WHILE B > A DO
          B := B - A;
     UNTIL A = B;
     WRITELN ('HCF IS', A)
END.
```

8. Find the maximum of N numbers.

```
"HOST"
PROGRAM REG8 (INPUT,OUTPUT);
VAR  N, MAXNO, NUMBER, I : INTEGER;
BEGIN
     READ (N);
     READ (MAXNO);
     FOR I := 1 TO N-1 DO
     BEGIN
          READ (NUMBER);
          IF NUMBER > MAXNO
               THEN MAXNO := NUMBER
     END;
     WRITELN ('THE MAXIMUM NO. IS', MAXNO)
END.
```

9. Count the positive and negative numbers.

```
"HOST"
PROGRAM REG9 (INPUT,OUTPUT);
VAR  POSCOUNT, NEGCOUNT : INTEGER;
     NUMBER             : REAL;
BEGIN
     POSCOUNT := 0;
     NEGCOUNT := 0:
     REPEAT
          READ (NUMBER);
          IF NUMBER > 0
               THEN POSCOUNT := POSCOUNT + 1
               ELSE NEGCOUNT := NEGCOUNT + 1
     UNTIL NUMBER = 0 [LAST NUMBER 0];
     WRITELN ('THERE WERE', POSCOUNT,' POSITIVE NUMBERS.');
     WRITELN ('THERE WERE', NEGCOUNT,' NEGATIVE NUMBERS.');
END.
```

10. Form separate totals of positive and negative numbers, and count the number of zeros.

```
"HOST"
PROGRAM REG10 (INPUT,OUTPUT);
VAR  I, N, NUMBER, POSITIVESUM, NEGATIVESUM, COUNT : INTEGER;

BEGIN
     READ (N);
     POSITIVESUM := 0;
     NEGATIVESUM := 0;
     COUNT := 0;
     FOR I := 1 TO N DO
          BEGIN
          READ (NUMBER);
          IF NUMBER = 0
               THEN COUNT := COUNT + 1
               ELSE IF NUMBER > 0
                    THEN POSITIVESUM := POSITIVESUM + NUMBER
                    ELSE NEGATIVESUM := NEGATIVESUM + NUMBER
          END;
     WRITELN ('TOTAL OF POSITIVE NUMBERS IS', POSITIVESUM);
     WRITELN ('TOTAL OF NEGATIVE NUMBERS IS', NEGATIVESUM);
     WRITELN ('AND THE NUMBER OF ZEROS IS', COUNT);
END.
```

11. Read number, print day of week.

```
"HOST"
PROGRAM REG11 (INPUT,OUTPUT);
VAR  DAYNO : INTEGER;
BEGIN
     READ (DAYNO);
     CASE DAYNO OF
          1:  WRITELN ('MONDAY');
          2:  WRITELN ('TUESDAY');
          3:  WRITELN ('WEDNESDAY');
          4:  WRITELN ('THURSDAY');
          5:  WRITELN ('FRIDAY');
          6:  WRITELN ('SATURDAY');
          7:  WRITELN ('SUNDAY');
     END {CASE}
END.
```

12. Add numbers until number is -1.

```
"HOST"
PROGRAM REG12 (INPUT,OUTPUT);
LABEL 123;
VAR  SUM, NUMBER : INTEGER;

BEGIN
     SUM := 0;
     WHILE TRUE DO
     BEGIN
          READ (NUMBER);
          IF NUMBER = -1
               THEN GOTO 123;
               SUM := SUM + NUMBER
     END;
     123: {JUMP TO THIS POINT WHEN -1 IS READ.}
     WRITELN ('THE SUM IS', SUM)
END.
```

13. Form the average of each of M groups of numbers, stopping if a negative number is encountered.

```
"HOST"
PROGRAM REG13 (INPUT,OUTPUT);
LABEL 33;
VAR  I,J,M,N                : INTEGER;
     AVERAGE, SUM, NUMBER : REAL;

BEGIN
     READ (M);
     FOR J := 1 TO M DO
     BEGIN
          READ (N);
          SUM := 0;
          FOR I := 1 TO N DO
          BEGIN
               READ (NUMBER);
               IF NUMBER < 0
                    THEN GOTO 33;
                    SUM := SUM + NUMBER
          END;
          SUM := SUM/N;
          WRITELN ('AVERAGE IS', AVERAGE)
     END;
     33: {JUMP TO THIS POINT IF NUMBER IS NEGATIVE}
END.
```

14. Tax calculation.

```
"HOST"
PROGRAM REG14 (INPUT,OUTPUT);
CONST  ALLOWANCE = 150;
RATE                = 0.35;
VAR  TAX, INCOME, EXPENSES, ALLOW, CHARITYLEVY : REAL;
     I, N, DEPENDENTS, EMPLTYPE                : INTEGER;

BEGIN
     READ (N);
     FOR I := 1 TO N DO
     BEGIN
          READ (INCOME, EXPENSES, DEPENDENTS, EMPLTYPE);
          CASE EMPLTYPE OF
               1:  CHARITYLEVY := 5;
               2:  CHARITYLEVY := 2;
               3:  CHARITYLEVY := 1
     END;
     EXPENSES := EXPENSES + CHARITYLEVY:
     ALLOW    := ALLOWANCE*DEPENDENTS+EXPENSES;
     IF INCOME > ALLOW
          THEN TAX := (INCOME - ALLOW) * RATE
          ELSE TAX := 0;
     WRITELN ('TAX FOR EMPLOYEE', I,' IS', TAX)
END.
```

15. Program to act as a hand calculator.

```
"HOST"
PROGRAM REG15 (INPUT,OUTPUT);
VAR  OPERATOR      : CHAR;
     ANSWER, NEWNO : REAL;

BEGIN
     ANSWER := 0;
     OPERATOR := +;
     REPEAT
          READ (NEWNO);
          READ (OPERATOR);
          CASE OPERATOR OF
               '+' := ANSWER + NEWNO;
               '-' := ANSWER - NEWNO;
               '*' := ANSWER * NEWNO;
               '/' := ANSWER / NEWNO
          END;
     UNTIL OPERATOR = '=';
     WRITELN ('ANSWER IS', ANSWER)
END.
```

16. Print all numbers with same sign as first number.

```
"HOST"
PROGRAM REG16 (INPUT,OUTPUT);
VAR  NUMBER    : REAL;
     FIRSTSIGN : BOOLEAN;
BEGIN
     READ (NUMBER);
     WRITELN (NUMBER);
     FIRSTSIGN := NUMBER >= 0;
     REPEAT
          READ (NUMBER);
          IF FIRSTSIGN = (NUMBER >= 0)
               THEN WRITELN (NUMBER)
     UNTIL NUMBER = 0
END.
```

17. Read lines of 4 CHAR name and number output each line in other order.

```
"HOST"
PROGRAM REG17 (INPUT,OUTPUT);
VAR  I,N,NUMBER : INTEGER;
     A,B,C,D    : CHAR;
BEGIN
     READLN (N);
     FOR I := 1 TO N DO
     BEGIN
          READ (A,B,C,D);
          READLN (NUMBER);
          WRITELN (NUMBER,A,B,C,D)
     END;
END.
```

18. Tabulate Centigrade integer temperatures from 0 to 99 degrees against the Fahrenheit equivalent to the nearest 0.1 degree.

```
"HOST"
PROGRAM REG18 (INPUT,OUTPUT);
CONST  CONVERSION = 1.8;
       OFFSET     = 32.0;
VAR  CENTEMP   : INTEGER;
     FAHRTEMP  : REAL;
BEGIN
     WRITELN ('CENTIGRADE FAHRENHEIT');
     FOR CENTEMP := 0 TO 99 DO
     BEGIN
          FAHRTEMP := CENTEMP * CONVERSION + OFFSET;
          WRITELN (CENTEMP:2'  ':7,FAHRTEMP:7:1)
     END;
END.
```

19. Add two numbers and print neatly.

```
"HOST"
PROGRAM REG19 (INPUT,OUTPUT);
VAR  NUM1, NUM2,TOTAL : REAL;
PROCEDURE DRAWLINE;
     CONST  LENGTH = 10;
     VAR  I : INTEGER;
     BEGIN
          FOR I := 1 TO LENGTH DO
               WRITE ('-');
               WRITELN
     END;

BEGIN
     READ (NUM1, NUM2);
     WRITELN (NUM1:10:3);
     WRITELN (NUM2:10:3);
     DRAWLINE
     TOTAL := NUM1 + NUM2;
     WRITELN (TOTAL:10;3);
     DRAWLINE
END.
```

20. Draw histogram as lines of appropriate length for the values read.

```
"HOST"
PROGRAM REG20 (INPUT,OUTPUT);
VAR  X,Y,N  : INTEGER;
     NUMBER : REAL;
     PROCEDURE DRAWALINE (LENGTH : INTEGER);
          VAR  I : INTEGER;
          BEGIN
               FOR I := 1 TO LENGTH DO
                    WRITE ('-');
                    WRITELN
          END;

     BEGIN
          READ (N);
          FOR X := 1 TO N DO
          BEGIN
               READ (NUMBER);
               Y := ROUND (NUMBER);
               IF Y < 0
                    THEN DRAWALINE (0)
                    ELSE IF Y > 100
                         THEN DRAWALINE (100)
                         ELSE DRAWALINE (Y)
          END
     END.
```

21. Order each of N data pairs.

```
"HOST"
PROGRAM REG21 (INPUT,OUTPUT);
VAR  I,N : INTEGER;
     X,Y : REAL;
PROCEDURE SWAP (VAR  P,Q : REAL);
     VAR  TEMP : REAL;
     BEGIN
          TEMP := P;
          P := Q;
          Q := TEMP;
     END;

BEGIN
     READ (N);
     FOR I := 1 TO N DO
     BEGIN
          READ (X,Y);
          IF X > Y
               THEN SWAP (X,Y);
     END;
     WRITELN ('ARE THE ORDERED PAIRS')
END.
```

22. Convert a number of inches into miles, yards, feet, and inches.

```
"HOST"
PROGRAM REG22 (INPUT,OUTPUT);
VAR  A,B,C,D,NUMBER : INTEGER;
PROCEDURE CONVERT (VAR  M,Y,F,I,INS : INTEGER);
BEGIN
     M   := INS DIV (1760 * 36);
     INS := INS MOD (1760 * 36);
     Y   := INS DIV 36;
     INS := INS MOD 36;
     F   := INS DIV 12;
     I   := INS MOD 12;
END;

BEGIN
     READ (NUMBER);
     CONVERT (A,B,C,D,NUMBER);
     WRITE(A:4, ' MILES,',B:4, 'YARDS,');
     WRITELN(C:1, 'FEET AND',D:2, ' INCHES')
END.
```

23. Lengths arithmetic.

```
"HOST"
PROGRAM REG23 (INPUT,OUTPUT);
VAR  A,B,C,D,TOTAL : INTEGER;
PROCEDURE CONVERT (VAR  M,Y,F,I,INS : INTEGER);
BEGIN
     M   := INS DIV (1760 * 36);
     INS := INS MOD (1760 * 36);
     Y   := INS DIV 36;
     INS := INS MOD 36;
     F   := INS DIV 12;
     I   := INS MOD 12;
END;
FUNCTION INCHES (M,Y,F,I : INTEGER) : INTEGER;
BEGIN
     INCHES := (((M * 1760) + Y) * 3 + F) * 12 + 1
END;

BEGIN
     READ (A,B,C,D);
     TOTAL := INCHES (A,B,C,D);
     CONVERT (A,B,C,D,TOTAL);
     WRITELN ('SUM IS', TOTAL, INCHES - I.E.);
     WRITE(A:4,' MILES,', B:4,' YARDS,');
     WRITELN(C:1,' FEET AND', D:2' INCHES')
END.
```

24. Print sum of N values. Each value is the average of a set of numbers.

```
"HOST"
PROGRAM REG24 (INPUT,OUTPUT);
CONST  WIDTH = 10;
VAR  I,N,COUNT          : INTEGER;
     LINEAVERAGE, TOTAL : REAL;
PROCEDURE DRAWALINE;
     VAR  I : INTEGER;
BEGIN
     FOR I := 1 TO WIDTH DO
     WRITE ('-');
     WRITELN
END;
FUNCTION AVERAGE (READCOUNT : INTEGER) : REAL;
VAR  I          : INTEGER;
     TOTAL, NUM : REAL;
BEGIN
     TOTAL := 0;
     FOR I := 1 TO READCOUNT DO
     BEGIN
          READ (NUM);
          TOTAL := TOTAL + NUM
     END;
     AVERAGE := TOTAL/ READCOUNT
END;

BEGIN
     TOTAL := 0;
     READ (N);
     FOR I := 1 TO N DO
     BEGIN
          READ (COUNT);
          LINEAVERAGE := AVERAGE (COUNT);
          WRITELN (LINEAVERAGE : WIDTH : 2);
          TOTAL := TOTAL + LINEAVERAGE
     END;
     DRAWALINE;
     WRITELN (TOTAL : WIDTH : 2);
     DRAWALINE
END.
```

25. Scalar enumeration types - defining a new type.

```
"HOST"
PROGRAM REG25 (INPUT,OUTPUT);
TYPE  SUIT     = (CLUB, DIAMOND, HEART, SPADE);
      DAY      = (SAT, SUN, MON, TUE, WED, THUR, FRI);
      PRICOLOR = (RED, YELLOW, BLUE);
      FLOOR    = (GROUND, LOWFIRST, FIRST);
VAR   PAYDAY, DAYOFF, : DAY;
      TRUMP     : SUIT;
      PAINT     : PRICOLOR;
      COATS     : INTEGER;
      UNDERCOAT : BOOLEAN;

BEGIN
      CASE PAINT OF
           RED :
                BEGIN
                     COATS := 1;
                     UNDERCOAT := FALSE;
                END;
           YELLOW :
                BEGIN
                     COATS := 2;
                     UNDERCOAT := TRUE
                END;
           BLUE :
                BEGIN
                     COATS := 2;
                     UNDERCOAT := FALSE
                END
      END;
END.
```

26. Read numbers and calculate their divisors.

```
"HOST"
PROGRAM REG26 (INPUT,OUTPUT);
VAR  NUMBER, DIVISOR : INTEGER;

BEGIN
      REPEAT
      READ (NUMBER);
      IF NUMBER > 0
           THEN
           BEGIN
                WRITELN ('THE DIVISORS OF', NUMBER,' ARE':');
                FOR DIVISOR := 2 TO NUMBER DO
                IF NUMBER MOD DIVISOR = 0
                     THEN WRITELN (DIVISOR)
           END
      UNTIL NUMBER <= 0
END.
```

27. Copying one or more lines of text from one file to another.

```
"HOST"
PROGRAM REG27 (INPUT,OUTPUT);
VAR  OLDFILE, NEWFILE : TEXT;
     CH               : CHAR;
     ERROR            : BOOLEAN;
     N,I, CURRENTLINE : 1..MAXINT;
PROCEDURE COPYLINE (VAR  F1, F2 : TEXT);
VAR  CH : CHAR;
BEGIN
     WHILE NOT EOLN (F1) DO
     BEGIN
          READ (F1, CH);
          WRITE (F2, CH)
     END;
     READLN (F1);
     WRITELN (F2)
END;  {PROCEDURE COPYLINE}

BEGIN {MAIN PROGRAM}
     RESET (OLDFILE);
     REWRITE (NEWFILE);
     CURRENT := 1;
     ERROR   := FALSE;
     REPEAT
          READ (CH);
          IF CH < > 'E'
               THEN READ (N);
               READLN;
               IF (CH = 'E') OR (CH = 'C') OR (CH = 'S')
               OR (CH = 'I')
                    THEN
                         CASE CH OF
                         'C' :
                         BEGIN
                              ERROR := N < CURRENTLINE;
                              FOR I := 1 TO N - CURRENTLINE DO
                                   COPYLINE (OLDFILE, NEWFILE);
                                   CURRENTLINE := N
                         END;
                         'S' :
                         BEGIN
                              ERROR := N < CURRENTLINE;
                              FOR I := 1 TO N-CURRENTLINE DO
                                   READLN (OLDFILE);
                                   CURRENTLINE := N
                         END;
                         'I' :
                              FOR I := 1 TO N DO
                                   COPYLINE (INPUT, NEWFILE);
```

```
                              'E' :
                                   WHILE NOT EOF (OLDFILE) DO
                                        COPYLINE (OLDFILE, NEWFILE)
                              END
                         ELSE ERROR := TRUE
     UNTIL (CH = 'E') OR ERROR;
               IF ERROR
                    THEN WRITELN ('ERROR IN EDIT')
END.
```

28. Print a table of powers of integers.

```
"HOST"
PROGRAM REG28 (INPUT,OUTPUT);
VAR  TABLESIZE, BASE, SQUARE, CUBE, QUAD : INTEGER;

BEGIN
     READ (TABLESIZE);
     FOR BASE := 1 TO TABLESIZE DO
     BEGIN
          CUBE := BASE * SQUARE;
          QUAD : SQR (SQUARE);
          WRITE(BASE, SQUARE, CUBE, QUAD);
          WRITELN(1/BASE, 1/SQUARE, 1/CUBE, 1/QUAD)
     END;
END.
```

29. Calculate area of a polygon.

```
"HOST"
PROGRAM REG29 (INPUT,OUTPUT);
VAR  V : INTEGER;
TYPE R = RECORD
     aa:integer;
     b,c : real;
     END;
FUNCTION A:R;
BEGIN
     A.aa := 3;
     a.b  := 0.6;
     a.c  := 0.2;
END;
BEGIN

     WITH A DO
          V := aa;
     WRITELN (V)
END.
```

Appendix C

Input/Output Characteristics

Introduction

Physical files are either disc files or I/O devices on the 64000 system. The I/O devices keyboard, display, printer, and display1 may only be associated with TEXT files. The I/O device null, and disc files, may be associated with either text or non-text files.

Disc Files

All disc files on the 64000 system consist of a list of variable length records, and can range from 2 to 256 bytes in length. The length of a physical record is always an even number of bytes. The term "logical record" refers to the data in one component of the file. That is, given a FILE OF T, a logical record consists of one data item of type T.

The system performs a mapping of logical records to physical records during an output operation, and a mapping of physical records into logical records during an input operation. The details of this mapping are explained below for both text files and non-text files.

Text Files

Each component, or logical record, of a text file is of type CHAR. Since text files are also structured into lines, the system maps each line into a physical disc record, with the following implications:

a. A line in a text file may never be longer than 256 bytes.

b. A line always consists of an even number of characters. The system generates an extra character, i.e., a blank, if necessary, to pad the line to an even number of characters when a WRITELN is executed.

c. A line can never have zero characters. For example, if the statements to WRITELN occur consecutively,

```
        .
        .
        .

    WRITELN (output);
    WRITELN (output);
        .
        .
        .
```

the record produced by the second WRITELN will consist of two blanks.

Non-text Files

The mapping of logical records to physical records takes place as follows:

a. Each logical record is written to one physical record if the length of the logical record is 256 bytes or less.

b. The system writes as many 256 byte physical records as are needed to satisfy the logical record length. Any remaining portion of the logical record is written into a shorter physical record.

c. The system always writes the same number of bytes for each record. If the file component is a record with variants, then the number of bytes in the longest variant is written.

The system considers the length of the record to be the length of the logical record when reading a non-text file. The record length, then, is determined by the program and not by the data in the file. The system reads physical records until enough data is obtained to fill a logical record.

Problems will occur if data is written using a file variable with one component type of a particular length, and a later attempt is made to read the data using a file variable with a component type that has a different length. The data in the physical records will not map properly into the logical records and results will be unpredictable.

I/O Devices

Null

"Null" is a nothing device. Used for input, null always produces an end-of-file condition. Used for output, null functions as a bit bucket.

Keyboard

"Keyboard" uses the command line area at the bottom of the 64000 screen for keyboard input. An end-of-file condition is produced by entering a zero-length line. In order to clear an end-of-file condition, if it is inadvertently produced, the following may be done:

```
Assume that F is a file assigned to "keyboard";
IF EOF(F) THEN RESET(F);
```

The cursor will always appear in the command area. The RETURN key terminates a line and causes data to be sent to the program. The BACK SPACE, CLR LINE, INSERT CHAR, DELETE CHAR, LEFT ARROW, RIGHT ARROW, and RECALL keys may be used to prepare and edit text in the command line.

ROLL UP, ROLL DOWN, NEXT PAGE, PREV PAGE, and TAB keys have no effect. No "control" character has any special meaning.

Display

The "display device" is 80 characters wide and 18 lines high. Data is always written to the bottom line of the display. The display rolls up and the data on the top line is lost to view. Lines wider than 80 characters are truncated and the additional data is not displayed. Time delays take place after each line is written to ensure that the display does not roll up too rapidly.

The display hardware is sensitive to certain characters that are used for special effects. The following bit patterns are significant:

10UIXXBX

where: X = don't care;
U = 1 turns on underlining;
U = 0 turns off underlining;
I = 1 turns on inverse video;
I = 0 turns off inverse video;
B = 1 turns on blinking video;
B = 0 turns off blinking video.

1111XX0X causes the display to be blanked to the end of the line.

1111XX1X causes the display to be blanked to the end of the display screen. Should this character be used, nothing beyond it, including the message line, command line, and softkey line will be visible.

Printer

The "printer" has a maximum line width of 132 characters. Data in lines wider than 132 characters will be lost. The printer will form feed if the standard procedure PAGE is used.

Display1

"Display1" is similar to the "display" device in that data is shown in the area at the top of the screen.

a. display may only be associated with a text file.

b. text files have a maximum line length.

c. The maximum line length defaults to 240 characters, but is setable from 1 to 256 characters with the "$LINESIZE$" option.

d. If more than the maximum number of characters are written without a WRITELN being issued, a WRITELN will be automatically generated.

e. Cursor positioning, using control characters to the display1 device is not a WRITELN. A common error occurs when control characters are sent to display1 to reset the cursor without a WRITELN. In the following example, some data will appear on line 2 of the display because of the automatic WRITELN generated by HOST Pascal.

```
FOR I := 1 TO 100 DO
     BEGIN
     WRITE (CHR (194)); {HOME CURSOR}
     WRITE ('I WANT THIS DATA TO APPEAR ON LINE 1');
     END;
```

The above FOR statement will operate correctly if the second WRITE is replaced by a WRITELN.

The "display_cursor", although invisible, determines where the next character received by display1 will be shown.

a. If the display_cursor is located in the 80th column of a line and another data character is to be displayed, then the cursor wraps to the first column of the next line.

b. If the display_cursor is on the 18th line and the display_cursor is forced to the next line, then the displayed data is rolled up, the top line is lost, the 18th line is blanked, and the display_cursor remains on the 18th line.

c. After performing a REWRITE or APPEND to a file associated with "display1", the display_cursor is positioned in the upper left hand corner, i.e., the home position.

d. The following characters are not displayed by "display1" but are treated as control characters:

chr(192) - SET X,Y - The next two characters received by "display1" are not displayed but are used to set the column and row, respectively, of the display_cursor. The columns are numbered 0 thru 79. If the column character is greater than 79, the column will be set equal to 79. The rows are numbered 0 thru 17. If row character is greater than 17, the row will be set equal to 17.

chr(193) - CARRIAGE RETURN - The display_cursor is set to the beginning of the current line, i.e., with column equal to zero.

chr(194) - HOME - The display_cursor is set to the "home" position, i.e., the upper left corner of the screen, with both row and column equal to zero.

chr(195) - CLEAR TO END OF LINE - The screen is blanked from the current display_cursor position to the end of the present line. The position of the display_cursor is unchanged.

chr(196) - CLEAR TO END OF SCREEN - The screen is blanked from the current display_cursor position to the end of the 19th line of the screen. The position of the display_cursor is unchanged.

chr(197) - CLEAR SCREEN - The entire data area of the screen is blanked. The position of the display_cursor is unchanged.

chr(198) - BINK - The audible alarm is sounded.

chr(199) - CURSOR RIGHT - The display_cursor is moved right one position. If the cursor is at the end of the line, a wrap to the next line occurs. If the cursor wraps and is on the bottom line, the display is rolled up one line.

chr(200) - CURSOR LEFT - The display_cursor is moved left one position. If the cursor is in the first column of a line, it is moved to the last column of the previous line. If the cursor is in the home position nothing happens.

chr(201) - CURSOR UP - The display_cursor is moved to the previous line with the column unchanged. If the cursor is on the top line of the display nothing happens.

chr(202) - CURSOR DOWN - The display_cursor is moved to the next line with the column unchanged. If the cursor was on the bottom line of the screen the screen is rolled up one line.

e. Characters having the bit pattern 1111XX1X are changed to 7FH. This bit pattern would cause the display to blank-to-end-of-display and is filtered out.

f. All other characters are displayed unchanged. For easy reference the display enhancement bit pattern is as follows:

10UIXXBX

where: X = don't care;
U = 1 turns on underlining;
U = 0 turns off underlining;
I = 1 turns on inverse video;
I = 0 turns off inverse video;
B = 1 turns on blinking video;
B = 0 turns off blinking video.

In addition, the bit pattern 1111XXXX causes the display to be blanked to the end of line.

RS-232

The RS-232 I/O device performs serial input and output in the asynchronous mode. The hardware interface is either the RS-232 port or the current loop connector. A thorough discussion of the serial hardware interface may be found in the HP 64000 System Configuration and Installation Reference Manual.

Input and output using the RS-232 port presents more problems to the programmer than I/O using disc files and other devices. A list of the extra or exaggerated problems follows.

a. Both input and output operations must be done on the same device. In the case of full duplex protocols, input and output operations occur simultaneously under the control of a Pascal program which is an inherently sequential language.

b. Since the behavior of the device at the other end of the communications link is beyond the control of the 64000 system, the program must be able to deal with exception conditions. The problem of no response or timeout must be handled.

c. Input and output operations must take place within real time limits. A more thorough discussion of timing considerations is presented later in this section.

d. The communications link has a much higher error rate than other peripheral devices.

e. Modem signals must sometimes be manipulated by the program.

HOST Pascal has features to solve these problems in most cases. A detailed description of the RS-232 implementation and an example program are shown at the end of this Appendix.

Hardware Options

The following hardware options are selected by switches on the 64000 I/O board. See the 64000 Configuration and Installation Reference Manual for details.

RS-232C or current loop interface.
20 ma. or 60 ma. current loop.
Internal or external clock source.
Baud rate using internal clock.
Full duplex or half duplex modem control.
Number of data bits in character.
Character parity bits (none, odd, or even).
Number of stop bits in character.

Receiver and Transmitter

It is useful to think of the RS-232 I/O device as two separate devices, a receiver and a transmitter that share the same hardware resource. The HOST Pascal program controls the RS-232 transmitter using one TEXT variable and the RS-232 receiver using a different TEXT variable. The following program fragment illustrates this arrangement.

```
VAR RECEIVER: TEXT; TRANSMITTER: TEXT; BEGIN RESET(RECEIVER,'rs232');
  REWRITE(TRANSMITTER,'rs232');
```

Since HOST Pascal allows a file variable to open for either reading or writing but not both, the use of two text files is required. The RESET procedure associates the text file RECEIVER with the RS-232 receiver device while the REWRITE procedure associates the text file TRANSMITTER with the RS-232 transmitter device. Normally, HOST Pascal would produce a run time error if two file variables were associated with the same device. An exception to this rule is allowed with RS-232 if one file is open for reading and another for writing.

Character and Protocol Transparency

The RS-232 device is designed to be transparent to any character code or communications protocol. The RS-232 receiver does not recognize any character as a control character and the RS-232 transmitter does not automatically send any control characters. The recognition or sending of control characters must be done in the HOST Pascal program. This arrangement has the following implications:

The end-of-file condition for the RS-232 device is not defined. If F is a text file associated with the RS-232 receiver, then EOF(F) always returns FALSE. If F is associated with the RS-232 transmitter, then EOF(F) will always return TRUE. Recognizing or producing end-of-file conditions must be accomplished by the HOST Pascal program.

Line markers are not defined for the RS-232 device. If F is a text file associated with RS-232 receiver, then EOLN(F) always returns FALSE. If F is associated with the RS-232 transmitter, then WRITELN(F) has no effect. Recognizing or producing line markers or record delimiters must be accomplished by the HOST Pascal program.

Circular Buffers

There is a circular data buffer associated with the TEXT variable that is open to the RS-232 receiver. There is another circular data buffer associated with the TEXT variable that is open to the RS-232 transmitter. These data buffers are located in the Device Control Block (DCB) that is associated with all file variables. (See Chapter 8, Table 8-2, for more information.)

The size of the DCB, and therefore the size of the circular buffer, is controllable through the use of the $BUFFERS n$ compiler option, the same as for other files. The value of n may be 1, 2, 4, 8, or 16 which results in a circular buffer size of 128, 256, 512, 1024, or 2048 words, respectively. In general, one word in the buffer can hold one character of data and any error information associated with it.

These circular buffers accomplish several things. When receiving, it allows higher speed operation. Typically, some received characters require only brief processing while others initiate lengthy processes. The HOST Pascal program does not have to process each character as soon as it is received but need only keep up with the average rate of data input. It also allows simultaneous transmitting and receiving in full duplex protocols. When one does a WRITE operation, the output data is stored in the circular buffer, transmission is initiated, and the WRITE procedure returns immediately. The HOST program can then do other processing, including receiving, while transmission is taking place. The programmer can use the standard function LINEPOS to determine the number of data characters in either circular buffer.

TIMEOUT(F,T) Procedure

There is a new predefined procedure that allows the HOST Pascal program to deal with the case of no input data to the RS-232 receiver. The procedure TIMEOUT(F,T) specifies the time interval for waiting for a character to be received on the RS-232 device. If no data, is received in the specified interval, a "timeout" condition occurs and the READ procedure returns control to the HOST Pascal program after setting the I/O result code to specified value. The HOST Pascal program calls the predefined function IORESULT to determine whether data was received or a timeout occurred.

The procedure TIMEOUT(F,T) has no effect unless the text file F is open for reading to the RS-232 device. The integer expression T specifies the time interval as a number of clock "ticks". A "tick" is either 1/60th or 1/50th of a second depending on the AC line frequency. After TIMEOUT is

called, the specified time interval remains in effect until either another call to TIMEOUT is made or the file F is closed. If the value of T is negative, then timing is disabled and the timeout interval is effectively infinite.

Example:

```
$EXTENSIONS ON$ VAR
  F: TEXT;
BEGIN RESET(F,'rs232'); TIMEOUT(F,120);
```

In the example, the timeout interval is set to 120 "ticks" which is 2.0 seconds if the AC line frequency is 60 Hz. or 2.4 seconds if the AC line frequency is 50 Hertz. TIMEOUT is an extension to HP Standard Pascal and requires the $EXTENSIONS ON$ compiler option to be in effect.

RS-232 Receiver Operation

The basic operation of the RS-232 receiver is as follows:

1. When a file is opened for input to the RS-232 (e.g., RESET(F,'rs232');), the circular buffer is reset to indicate empty and the receiver hardware is enabled. The timeout interval is initialized to -1 which specifies an infinite time. Subsequent calls to TIMEOUT can be made to change this value.

2. As characters are received, the character is combined with any error indications associated with the character and put into a word in the circular buffer. Possible error indications are BREAK, overrun error, framing error, and parity error.

3. If the circular buffer contains only one free word, then the character/error data is not stored. Rather an error code indicating buffer overflow is stored and the buffer becomes full. If characters are received after the buffer is full, they are discarded but there will always be a buffer overflow indication when any data is lost.

4. The READ(F,CH) procedure where CH is a CHAR variable is used to remove character/error data from the circular buffer. If no data is in the buffer, this procedure will wait until a character is received or until the interval specified by TIMEOUT has elapsed. The character data is put into CH and the error information is used to set the value for a subsequent call to IORESULT. IORESULT returns the following values:

   0 -No error. CH contains the received data.

   15 -Timeout. CH contains CHR(0).

   16 -Buffer overflow. CH contains CHR(0).

   17 -Break received. CH contains CHR(0).

   18 -Framing error. CH contains the received

   19 -Overrun error. CH contains the received data.

   20 -Parity error. CH contains the received data.

5. The predefined procedure LINEPOS(F) will return the number of data characters in the circular buffer at any time.

6. When a file open for reading to the RS-232 device is closed, the receiver hardware is disabled.

RS-232 Transmitter Operation

The general operation of the RS-232 transmitter is as follows:

1. When a file is opened for output to the RS-232 device, the circular buffer is reset to indicate empty and the transmit hardware is disabled.

2. When an output operation (PUT or WRITE) is done, the output data characters are put into words in the circular buffer and the transmitter hardware is enabled. If there is space in the circular buffer for all the data, the PUT or WRITE procedures return immediately. Otherwise, they will wait until space becomes available before returning. Space will become available as the transmitter hardware removes the data from the buffer and sends it.

NOTE

The transmitter hardware will not actually send data until the CTS (Clear To Send) modem signal is high. See the HP 64000 System Overview Manual for details.

3. After the transmitter hardware sends a character, it looks in the circular buffer for another. If more data is available, it is transmitted. Otherwise, the transmitter hardware is disabled.

4. The predefined function LINEPOS(F) may be used to determine the number of characters remaining in the buffer at any time. The value returned by LINEPOS is the number of characters in the circular buffer plus the number of characters contained in the transmitter hardware registers.

5. Whenever a file open for writing to the RS-232 is closed, the close procedure will wait until all characters in the circular buffer have actually been sent before finishing.

Modem Control

The modem signals DTR (Data Terminal Ready) and RTS (Request To Send) can be controlled by the HOST Pascal. Note that RTS is only controllable if the full/half duplex switch on the I/O board is in the half duplex position. Otherwise, RTS is always high. The normal state for the DTR and RTS signals is high. When a HOST Pascal program is run, the RS-232 hardware is initialized and both DTR and RTS are set low. The DTR signal is set high when a file is opened for reading to the RS-232 device. The DTR signal remains high until the file is closed. Thus the program can control DTR by alternately calling RESET and CLOSE. The RTS signal is set high when a file is opened for writing to the RS-232 device. The RTS signal remains high until the file is closed and all characters in the transmitter buffers have been sent. Thus the program can control RTS by alternately calling REWRITE and CLOSE. Note that the RS-232 transmitter hardware will not send data until the CTS (Clear To Send) input signal is high.

Restricted Use of READ and READLN

When a text file F is open for reading to the RS-232 device, many forms of READ(F,V) and READLN(F) are not allowed. A run-time error will result if the type of variable V is integer, real, longreal, string, or PAC. READLN is always illegal with the RS-232 device. READ is only allowed when the variable V is compatible with type CHAR. The are two reasons for these restrictions. READLN and READ of string and PAC variables depend on the EOLN condition in their definitions. However, line markers are not defined for the RS-232 device. READ with integer and real variables is prohibited because the interaction of multi-character input and timeout timing is difficult to define in a straight forward way.

Timing Considerations

In data communications programs written in HOST Pascal, there are three areas where real time constraints must be considered.

1. When receiving, the system software must remove a character from the receiver hardware register before the next character is received. If this does not happen fast enough, the result is called an overrun error. The critical time interval here is the character transmission time which is a function of baud rate. At 110 bps, the character time is about 100 ms. At 9600 bps, the character time is about 1 ms.

2. When receiving, the HOST Pascal program must remove data from the circular buffer faster, on the average, than it is received. If this does not happen, eventually the circular buffer will become full and the result is called a buffer overflow error. The average rate that characters are received is a function of baud rate and, also, the communications protocol of the sending device. That is, if the sender sends blocks of data and pauses, the average transmission rate is less than if the sending was continuous. The average rate that characters are removed is a function of the HOST Pascal program; how much time it spends processing each character.

3. In some communications protocols, one station sends a message to another station and expects a response within a certain time interval. If the response is not received within the time limit, the result is a timeout error. The critical time interval is a function of the particular computer system being used. Typically, this time is on the order of several seconds.

Given the above problems, there follows a list of recommendations to avoid or overcome them. Since data communications systems are so varied, meaningful limits on performance or speed cannot be formulated. Therefore, the following discussion is general.

Overrun errors should only be a problem at speeds of 2400 bps and above. They can be nearly eliminated by avoiding disc I/O operations while data is begin received. That is, at high speed, pick a communications protocol and design your HOST Pascal program so that disc operations and RS-232 receiving can happen at alternate times.

The problem of receive buffer overflow is complicated with many factors involved. One important factor is communications protocol. At speeds of 2400 bps and above, it is probably impossible to do any meaningful processing of received data if transmission is continuous. The HOST Pascal program simply cannot process the received characters fast enough. One must use a communications protocol that sends data in bursts and then pauses so that the HOST program can catch up with its data processing.

Using a communications protocol that divides the transmissions into blocks has another advantage. One can then design a program where is main receiving loop is very brief. Typically, the main receiving loop need only call the function IORESULT to check for errors, store the received character in memory, and check if the character is a transmission terminator. Time consuming functions can be performed in the time interval between blocks. Examples of time consuming functions are I/O

operations to the disc, printer or display, computations of CRC sums, data reformatting, and data interpretation. In extreme cases, avoid procedure and function calls in favor of in-line code. Also, avoid doing READ or WRITE operations with integer or real data to text files. The conversion to and from internal format to character format requires considerable computation.

Example Program of RS-232 Implementation

```
"HOST"
{*****************************************************************}
{                                                                 }
{ THIS PROGRAM SIMULATES A "DUMB" TERMINAL.  IT ACCEPTS DATA      }
{ FROM THE KEYBOARD AND SENDS IT ON THE rs232 DEVICE.  THE        }
{ RECEIVING STATION ECHOES THIS DATA.  THE PROGRAM RECEIVES       }
{ DATA FROM THE rs232 DEVICE AND DISPLAYS THE PRINTABLE           }
{ CHARACTERS ON THE display1 DEVICE.                              }
{                                                                 }
{ THE REMOTE STATION USES THE ENQ/ACK PROTOCOL.  THE REMOTE       }
{ STATION SENDS ENQ AFTER EVERY 80 CHARACTERS OF OUTPUT DATA.     }
{ THE PROGRAM RESPONDS WITH ACK WHEN IT IS READY TO ACCEPT 80     }
{ MORE CHARACTERS.                                                }
{                                                                 }
{ GENERAL FLOW                                                    }
{  1.  INITIALIZATION (OPEN FILES)                                }
{  2.  READ THE KEYBOARD.  IF KEYBOARD DATA IS '\\', END PROGRAM. }
{  3.  TRANSMIT THE KEYBOARD DATA OVER THE RS232 FOLLOWED BY CR.  }
{  4.  RECEIVE AND DISPLAY DATA FROM RS232 UNTIL DC1 CHAR IS      }
{      RECEIVED OR UNTIL THE TIMEOUT INTERVAL HAS ELAPSED.        }
{  5.  GO TO 2.                                                   }
{                                                                 }
{*****************************************************************}
$EXTENSIONS ON$
PROGRAM TTERM;
CONST
   {CONTROL CHARACTERS FOR THE display1 DEVICE}
   INVERSE = CHR(144);
   ENH_OFF = CHR(128);
   CARRET = CHR(193);
   CLRSCREEN = CHR(197);
   BINK = CHR(198);
   CURLEFT = CHR(200);
   CURDOWN = CHR(202);

   {CODES RETURNED BY THE FUNCTION IORESULT}
   INOERROR = 0;
   IENDOFFILE = 1;
   ITIMEOUT = 15;
   IBUFOFLO = 16;
   IBREAK = 17;
   IFRAMERR = 18;
   IOVERRUN = 19;
   IPARITY = 20;
```

```
   {ASCII CONTROL CHARACTERS}
   ENQ = CHR(5);
   ACK = CHR(6);
   BELL = CHR(7);
   BS = CHR(8);
   LF = CHR(10);
   CR = CHR(13);
   DC1 = CHR(17);
   DEL = CHR(127);

VAR
  DONE1,DONE2: BOOLEAN;
  CH: CHAR;
  STR: STRING[240];
  KBD,DISP: TEXT;
$BUFFERS 2$
  INRS,OUTRS: TEXT;

BEGIN
{*********************************************************************}
{               1.  INITIALIZATION                                    }
{*********************************************************************}
RESET(KBD,'keyboard');
REWRITE(DISP,'display1');
RESET(INRS,'rs232');
REWRITE(OUTRS,'rs232');
WRITE(DISP,CLRSCREEN);
TIMEOUT(INRS,600);        {10 SECOND TIMEOUT INTERVAL}
DONE2 := FALSE;
{*********************************************************************}
{ 2.  READ KEYBOARD.  IF DATA IS '\\' THEN END THE PROGRAM.            }
{*********************************************************************}
REPEAT
  IF EOF(KBD) THEN
    BEGIN
    RESET(KBD);
    STR := '';
    END
  ELSE
    READLN(KBD,STR);
  IF STRPOS('\\',STR) = 1 THEN
    BEGIN
    DONE2 := TRUE;
    DONE1 := TRUE;
    END
  ELSE
```

```
{*****************************************************************}
{ 3.  TRANSMIT THE KEYBOARD DATA FOLLOWED BY A CARRIAGE RETURN    }
{*****************************************************************}
    BEGIN
    WRITE(OUTRS,STR,CR);
    DONE1 := FALSE;
    END;
{*****************************************************************}
{ 4.  RECEIVE AND DISPLAY DATA FROM RS232 UNTIL A DC1 IS RECEIVED}
{     OR UNTIL TIMEOUT OCCURS.                                    }
{*****************************************************************}
  WHILE NOT DONE1 DO
    BEGIN {WHILE}
$IOCHECK OFF$
    READ(INRS,CH);
$IOCHECK ON$
    CASE IORESULT OF
      INOERROR:
         BEGIN {NO ERROR}
         IF ORD(CH) >= 128 THEN
            CH := CHR(ORD(CH) - 128);
         CASE CH OF
           ' '..DEL:
             WRITE(DISP,CH);
           CR:
             BEGIN
             WRITE(DISP,CARRET);
             END;
           LF:
             WRITELN(DISP);
           BS:
             IF LINEPOS(DISP) > 0 THEN
               WRITE(DISP,CURLEFT);
           BELL:
             WRITE(DISP,BINK);
           ENQ:
             WRITE(OUTRS,ACK);    {RESPONSE TO ENQ WITH ACK}
           DC1:
             DONE1 := TRUE;
           OTHERWISE
           END;  {CASE}
         END;  {NO ERROR}
```

```
      ITIMEOUT:
         DONE1 := TRUE;
      IBUFOFLO:
         WRITE(DISP,INVERSE,'BUF',ENH_OFF);
      IBREAK:
         WRITE(DISP,INVERSE,'BRK',ENH_OFF);
      IFRAMERR:
         WRITE(DISP,INVERSE,'FRM',ENH_OFF);
      IOVERRUN:
         WRITE(DISP,INVERSE,'OVR',ENH_OFF);
      IPARITY:
         WRITE(DISP,INVERSE,'PAR',ENH_OFF);
      OTHERWISE
         HALT;
      END;  {CASE}
    END;  {WHILE}
{*************************************************************************}
{ 5.  GO TO 2.                                                            }
{*************************************************************************}
UNTIL DONE2;
END.
```

INDEX

a

Abs.....7-15
Absolute file.....9-1
Allocation for elements of packed structures.....8-4
Allocation for scalar variables.....8-1
Allocation for structured variables.....8-3
Allocation, memory.....8-11
Alphabetic characters.....2-1
AND.....5-18
APPEND.....1-2, 6-5, 6-6, 7-1
Arctan.....7-16
Arithmetic functions.....7-15
Arithmetic operators.....5-17
Array.....1-1, 4-15
Array constant.....4-7
Array subscripts.....5-16
Arrays, multi-dimensioned 4-16
Assignment compatible types.....5-26
Assignment statement.....5-2
Associating files through the string parameter.....6-6
Associating logical and physical files.....6-5

b

BINARY.....1-2, 7-19
BINK, chr(198).....C-5
Boolean operators.....5-18
Buffer variables.....4-24

c

CARRIAGE RETURN, chr(193).....C-4
CASE.....1-1
CASE Statement.....5-6
Character and protocol transparency, RS-232.....C-7
Character set.....2-1
Characteristics, logical file.....6-2
Characters, alphabetic.....2-1
Characters, numeric.....2-1
Characters, special.....2-1
Chr.....7-18
chr(192), SET X,Y.....C-4
chr(193), CARRIAGE RETURN.....C-4
chr(194), HOME.....C-5
chr(195), CLEAR TO END OF LINE.....C-5
chr(196), CLEAR TO END OF SCREEN.....C-5

Index (Cont'd)

chr(197), CLEAR SCREEN.....C-5
chr(198), BINK.....C-5
chr(199), CURSOR RIGHT.....C-5
chr(200), CURSOR LEFT.....C-5
chr(201), CURSOR UP.....C-5
chr(202), CURSOR DOWN.....C-5
Circular buffers, RS-232.....C-8
CLEAR SCREEN, chr(197).....C-5
CLEAR TO END OF LINE, chr(195).....C-5
CLEAR TO END OF SCREEN, chr(196).....C-5
CLOSE.....1-2, 7-2
Closing files.....6-13
Command parameters.....9-2
Comments.....2-8
Compatible types.....5-25
Compiler options.....2-8
Compile-Time error messages.....A-1
Compiling the source file.....9-1
Component variables.....4-24
Compound statements.....4-31, 5-5
CONST.....1-1
CONSTant declaration.....4-3
Constant expressions.....4-6, 5-24
Constant, array.....4-7
Constant, RECORD.....4-8
Constant, SET.....4-9
Constant, simple.....4-6
Constant, structured.....4-6
Constants, predefined.....2-7
Cos.....7-16
Current position pointer.....6-3
CURSOR DOWN, chr(202).....C-5
CURSOR LEFT, chr(200).....C-5
CURSOR RIGHT, chr(199).....C-5
CURSOR UP, chr(201).....C-5

d

Data allocation.....8-1
Data types, string.....4-17
DCB.....8-11
Declaration section.....4-2
 CONSTant.....4-3
 LABEL.....4-2
 Literal, string.....4-4
 Simple constants.....4-6
Declaration, FUNCTION.....4-26
Declaration, PROCEDURE.....4-25

Index (Cont'd)

Declaration, VARiable....4-23
Declarations, routine....4-25
Declarations, within routines....4-29
Delimiter, compiler directive....1-1
Device control buffer, (DCB)....8-11
Directives....2-7, 4-30
Disc files....C-1
Display, I/O Device....C-3
Display1, I/O Device....C-3
Display_cursor....C-4
DISPOSE....1-1, 7-11, 7-12
Double precision data type....1-2
DTR (Data Terminal Ready)....C-11
Dynamic variables....4-15
Dynamic variable base type....4-15

e

Empty statement....5-11
Entire variables....4-24
Eof....7-16
EOLN....6-13, 7-16
Error messages, compile-time....A-1
Error messages, run-time....A-5
Example Program of RS-232 Implementation....C-13
Exp....7-15
Expression operands....4-6
Expressions....5-11
Extensions, Pascal....1-1
APPEND....1-2
Array....1-1
BINARY....1-2
CASE....1-1
CLOSE....1-2
Compiler directive delimiter....1-1
CONST....1-1
DISPOSE....1-1
Double precision data type....1-2
HEX....1-2
LINEPOS....1-2
LONGREAL....1-2
MARK....1-1
NEW....1-1
OCTAL....1-2
OTHERWISE....1-1
POSITION....1-2
Predefined functions....1-2
Predefined procedures....1-2
Range symbol....1-1

Index (Cont'd)

RECORD.....1-1
RELEASE.....1-1
Set.....1-1
STRING.....1-2
TYPE.....1-1
VAR.....1-1

f

Field designators.....4-24
Field selection.....5-16
Field-width parameter.....6-11
FILE.....4-21
File buffer selection.....5-16
File buffer variable.....6-3
File handling procedures.....7-1
File handling functions.....7-19
File opening, procedure.....6-1
File states.....6-3
File variables.....6-1
Files, logical.....6-1
Files, physical.....6-2
Files, predefined.....2-6
Files, sequential.....6-2
Files, textfiles.....6-2
FOR Statement.....5-8
FUNCTION Declaration.....4-26
Function parameter.....4-29
Function references.....5-23
Functions, arithmetic.....7-15
 Abs.....7-15
 Arctan.....7-16
 Cos.....7-16
 Exp.....7-15
 Ln.....7-15
 Sin.....7-16
 Sqr.....7-15
 Sqrt.....7-15
Functions, file handling.....7-19
 IORESULT.....7-19
 LINEPOS.....7-19
 POSITION.....7-19
Functions, numeric conversion.....7-19
 BINARY.....7-19
 HEX.....7-19
 OCTAL.....7-19

Index (Cont'd)

Functions, ordinal.....7-17
Chr.....7-18
Ord.....7-17
Pred.....7-18
Succ.....7-18
Functions, predefined.....1-2, 2-6, 5-24
Functions, string handling.....7-6
STR.....7-8
STRLEN.....7-8
STRLTRIM.....7-9
STRMAX.....7-8
STRPOS.....7-10
STRRPT.....7-10
STRRTRIM.....7-9
Functions, transfer.....7-16
Round.....7-17
Trunc.....7-16

g

GET.....6-6, 7-2
GOTO Statement.....5-10

h

Hardware Options, RS-232.....C-7
Heap.....5-16, 7-10, 8-11
HEX.....1-2, 7-19
HOME, chr(194).....C-5
HOST Compiler options.....2-8

i

Identical types.....5-25
Identifiers.....2-5
Identifiers, predefined.....2-6
IF Statement.....5-5
Indexed variables.....4-24
Input file.....6-1
I/O Devices.....C-2
Display.....C-3
Display1.....C-3
Keyboard.....C-2
Null.....C-2
physical files.....6-2
Printer.....C-3
RS-232.....C-6

Index (Cont'd)

I/O Error handling....8-12
Integer literals....5-14
IOCHECK....8-12
IORESULT....7-19

k

Keyboard, I/O Device....C-2

l

LABEL....4-1
LABEL Declaration....4-2
Legend, error message....A-5
LINEPOS....1-2, 6-13, 7-19
LINESIZE, option....6-12
Listfile....9-3
Literal value....4-3
Literals....5-13
 integer....5-14
 real....5-14
 string....2-7, 4-10, 5-14
Ln....7-15
Logical file characteristics....6-2
Logical files....6-1
LONGREAL....1-2

m

Main program module....3-1
MARK....1-1, 7-11, 7-12
Memory allocation....8-11
Modem control, RS-232....C-11
Multi-dimensioned arrays....4-16

n

NEW....1-1, 7-11
NEW(p)....4-15
NIL, pointer....4-15
Non-text files....C-2
NOT....5-18
Null, I/O Device....C-2
Numeric characters....2-1
Numeric conversion functions....7-19

Index (Cont'd)

o

OCTAL....1-2, 7-19
Odd....7-16
Opening files....6-4
Operands....5-13, 5-24
Operators....5-17, 5-24
 arithmetic....5-17
 boolean....5-18
 set....5-19
Option, IOCHECK....8-12
Options....9-3
Options, HOST compiler....2-8
OR....5-18
Ord....7-17
Ordinal functions....7-17
Ordinal relationals....5-21
Ordinal types, predefined....4-11, 4-12
 BOOLEAN....4-12
 CHAR....4-12
 INTEGER....4-13
Ordinal types, user defined....4-11, 4-13
 Enumerated type....4-13
 Subrange type....4-13
OTHERWISE....1-1
Output file....6-1

p

PAC....4-17
PAC Relationals....5-22
PACK....4-23, 7-12
PACKED....4-15
PACKED type modifier....4-22
PAGE....6-13, 7-3
Parameter lists....4-27
 Actual parameter list....4-28
 Formal parameter list....4-27
 Procedure parameter....4-29
 Value parameter....4-28
 Variable parameter....4-28
Parameter List Compatibility....4-29
Parameter, function....4-29
Parameter, listfile....9-3
Parameter, options....9-3

Index (Cont'd)

Parameter, passed.....7-15
Parameter, procedure.....4-29
Parameter, source file.....9-2
Parameters, command.....9-2
Parameters, run command.....9-4
Pascal statement listing, sample programs.....B-1
Physical files.....6-2
Pointer dereferencing.....5-16
Pointer relationals.....5-22
Pointer types.....4-15
Pointer, current position.....6-3
POSITION.....1-2, 7-19
Pred.....7-18
Predefined functions.....5-24
Predefined identifiers.....2-6
 constants.....2-7
 files.....2-6
 functions.....2-6
 procedures.....2-6
 types.....2-7
Predefined ordinal types.....4-12
Printer, I/O Device.....C-3
Procedures, predefined.....1-2, 2-6
PROCEDURE Declaration.....4-25
Procedure statement.....5-3
Procedures and Functions, summary.....6-14
Procedures, Allocation and De-allocation.....7-10
 DISPOSE.....7-12
 MARK.....7-12
 NEW.....7-11
 RELEASE.....7-12
Procedures, file handling.....7-1
 APPEND.....6-5, 7-1
 CLOSE.....7-2
 GET.....6-6, 7-2
 PAGE.....6-13, 7-3
 PUT.....6-6, 7-3
 READ.....6-6, 7-3
 READLN.....6-6, 7-4
 RESET.....6-4, 7-2
 REWRITE.....6-4, 7-2
 TIMEOUT.....7-4
 WRITE.....6-6, 7-5
 WRITELN.....6-6, 7-5

Index (Cont'd)

Procedures, string handling.....7-6
SETSTRLEN.....7-6
STRAPPEND.....7-6
STRDELETE.....7-7
STRINSERT.....7-6
STRMOVE.....7-7
STRREAD.....7-9
STRWRITE.....7-10
Procedures, transfer.....7-12
PACK.....7-12
UNPACK.....7-14
Predicates.....7-16
Eof.....7-16
Eoln.....7-16
Odd.....7-16
Program block.....3-2
Program heading.....3-1
Program parameters, additional.....9-6
PUT.....6-6, 7-3

r

Range symbol.....1-1
READ.....6-6, 7-3
READLN.....6-6, 7-4
Real types.....4-14
REAL.....4-14
LONGREAL.....4-14
Real literals.....5-14
Receiver Operation, RS-232.....C-9
RECORD.....1-1, 4-19
RECORD constant.....4-8
RECORD field.....4-19
Recursive routines.....4-30
Referenced variables.....4-25
Relational operators.....5-21
Relational operators, sets.....4-21
Set difference (-).....4-21
Set intersection (*).....4-21
Set union (+).....4-21
Relationals, ordinal.....5-21
Relationals, PAC.....5-22
Relationals, pointer.....5-22
Relationals, set.....5-23
RELEASE.....1-1, 7-11, 7-12
REPEAT Statement.....5-8
Reserved words.....2-4
RESET.....6-4, 6-6, 7-2
Restricted Use of READ and READLN.....C-11

Index (Cont'd)

REWRITE.....6-4, 6-6, 7-2
Round.....7-17
Routine body.....4-29
Routine declarations.....4-25
Routines, recursive.....4-30
RS-232 Hardware Options.....C-7
RS-232 Implementation, example program.....C-13
RS-232, I/O Device.....C-6
RS-232 Receiver and Transmitter.....C-7
 Character and protocol transparency.....C-7
 Transmitter operation.....C-10
Run command parameters.....9-4
Running HOST Pascal programs.....9-4
Run-Time error messages.....A-5
RTS (Request To Send).....C-11

S

Sample Pascal programs.....B-1
Scope.....4-30
SEGMENT, directive.....4-1
Segment, program.....3-4
Selectors.....5-15
Sequential file operations.....6-6
Sequential Files.....6-2
SET.....1-1, 4-21
SET, base type.....4-21
SET constant.....4-9
Set constructor.....5-20
Set difference.....5-19
Set intersection.....5-19
Set operators.....5-19
Set relationals.....5-23
Set union.....5-19
SET X,Y, chr(192).....C-4
SETSTRLEN.....7-6
Simple statements.....4-31
Sin.....7-16
Source file.....9-1, 9-2
Source file, compiling.....9-1
Special cases.....5-27
Special characters.....2-1
Sqr.....7-15
Sqrt.....7-15
Static variables.....4-15
STACK.....8-11

Index (Cont'd)

Statement label.....5-2
Statements.....4-31
Statements and expressions.....5-1
Statements, compound.....4-31
Statements, simple.....4-31
STRING.....1-2
String comparison.....5-22
String data types.....4-17
String handling procedures and functions.....7-6
 STR.....7-8
 STRAPPEND.....7-6
 STRDELETE.....7-7
 STRINSERT.....7-6
 STRLEN.....7-6, 7-8
 STRLTRIM.....7-9
 STRMAX.....7-6, 7-8
 STRMOVE.....7-7
 STRPOS.....7-10
 STRREAD.....7-9
 STRRPT.....7-10
 STRRTRIM.....7-9
 STRWRITE.....7-10
String literals.....2-7, 4-4, 4-10, 5-14
String operators.....5-21
Structured constants.....4-6
Structured types.....4-15
Succ.....7-18
Summary of procedures and functions.....6-14
Symbol, nonprinting character.....1-2
Symbolic constants.....5-14

t

Tag field.....4-8, 4-19
Tagless variant.....4-8
Textfile operation.....6-6
Textfiles.....6-2, C-1
TIMEOUT.....7-4
TIMEOUT Procedure, RS-232.....C-8
 RS-232 Receiver operation.....C-9
Timing considerations, RS-232.....C-11
Transfer functions.....7-16
Transfer procedures.....7-12
Transmitter Operation, RS-232.....C-10
 Modem control.....C-11
Trunc.....7-16
TYPE.....1-1, 4-10
Type compatibility.....5-25
Type declaration.....4-11
 Simple types.....4-11

Index (Cont'd)

TYPE definitions....4-10
enumerated....4-13
real....4-14
ordinal....4-11
pointer....4-15
predefined....2-7
simple....4-11
structured....4-15
subrange....4-13

U

UNPACK....4-23, 7-14
User defined ordinal types....4-13
Enumerated type....4-13
Subrange type....4-13

V

Value, literal....4-3
VAR....1-1
VARiable....4-23
VARiable, declaration....4-23
Variable, file buffer....6-3
Variables....5-15
buffer....4-24
component....4-24
dynamic....4-15
entire....4-24
indexed....4-24
referenced....4-25
static....4-15
Variant field....4-8
Variant, CASE....4-19
OTHERWISE....4-19
Variant, RECORD....4-19

W

WHILE Statement....5-7
WITH Statement....5-9
WRITE....6-6, 7-5
WRITELN....6-6, 7-5

64817-90904, DECEMBER 1983
Replaces: 64817-90903, May 1982

PRINTED IN U.S.A.