

PASCAL REFERENCE
MANUAL
for the Lisa™

*Beta Draft
April 1983*

Customer Satisfaction

If you discover physical defects in the manuals distributed with a Lisa product or in the media on which a software product is distributed, Apple will replace the documentation or media at no charge to you during the 90-day period after you purchased the product.

In addition, if Apple releases a corrective update to a software product during the 90-day period after you purchased the software, Apple will replace the applicable diskettes and documentation with the revised version at no charge to you during the six months after the date of purchase.

In some countries the replacement period may be different; check with your authorized Lisa dealer. Return any item to be replaced with proof of purchase to Apple or to an authorized Lisa dealer.

Limitation on Warranties and Liability

All implied warranties concerning this manual and media, including implied warranties of merchantability and fitness for a particular purpose, are limited in duration to ninety (90) days from the date of original retail purchase of this product.

Even though Apple has tested the software described in this manual and reviewed its contents, neither Apple nor its software suppliers make any warranty or representation, either express or implied, with respect to this manual or to the software described in this manual, their quality, performance, merchantability, or fitness for any particular purpose. As a result, this software and manual are sold "as is," and you the purchaser are assuming the entire risk as to their quality and performance.

In no event will Apple or its software suppliers be liable for direct, indirect, special, incidental, or consequential damages resulting from any defect in the software or manual, even if they have been advised of the possibility of such damages. In particular, they shall have no liability for any programs or data stored in or used with Apple products, including the costs of recovering or reproducing these programs or data.

The warranty and remedies set forth above are exclusive and in lieu of all others, oral or written, express or implied. No Apple dealer, agent or employee is authorized to make any modification, extension or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights that vary from state to state.

License and Copyright

This manual and the software (computer programs) described in it are copyrighted by Apple or by Apple's software suppliers, with all rights reserved, and they are covered by the Lisa Software License Agreement signed by each Lisa owner. Under the copyright laws and the License Agreement, this manual or the programs may not be copied, in whole or in part, without the written consent of Apple, except in the normal use of the software or to make a backup copy. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold, given, or loaned to other persons if they agree to be bound by the provisions of the License Agreement. Copying includes translating into another language or format.

You may use the software on any computer owned by you, but extra copies cannot be made for this purpose. For some products, a multiuse license may be purchased to allow the software to be used on more than one computer owned by the purchaser, including a shared-disk system. (Contact your authorized Lisa dealer for more information on multiuse licenses.)

Product Revisions

Unless you have purchased the product update service available through your authorized Lisa dealer, Apple cannot guarantee that you will receive notice of a revision to the software described in this manual, even if you have returned a registration card received with the product. You should check periodically with your authorized Lisa dealer.

©1983 by Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010

Apple, Lisa, and the Apple logo are trademarks of Apple Computer, Inc.

Simultaneously published in the USA and Canada.

Reorder Apple Product #A6D0101 (Complete Pascal package)
#A6L0111 (Manuals only)

CONTENTS

Chapter 1

TOKENS AND CONSTANTS

1.1	Character Set and Special Symbols	1-3
1.2	Identifiers	1-4
1.3	Directives	1-4
1.4	Numbers.....	1-4
1.5	Labels	1-6
1.6	Quoted String Constants.....	1-6
1.7	Constant Declarations	1-7
1.8	Comments and Compiler Commands.....	1-7

Chapter 2

BLOCKS, LOCALITY, AND SCOPE

2.1	Definition of a Block	2-3
2.2	Rules of Scope	2-5

Chapter 3

DATA TYPES

3.1	Simple-Types (and Ordinal-Types)	3-4
3.2	Structured-Types	3-9
3.3	Pointer-Types	3-16
3.4	Identical and Compatible Types	3-16
3.5	The Type-Declaration-Part	3-19

Chapter 4

VARIABLES

4.1	Variable-Declarations	4-3
4.2	Variable-References.....	4-3
4.3	Qualifiers	4-4

Chapter 5

EXPRESSIONS

5.1	Operators	5-6
5.2	Function-Calls	5-13
5.3	Set-Constructors	5-14

Chapter 6

STATEMENTS

6.1	Simple Statements	6-3
6.2	Structured-Statements	6-6

Chapter 7

PROCEDURES AND FUNCTIONS

7.1	Procedure-Declarations	7-3
7.2	Function-Declarations	7-6

7.3	Parameters	7-7
Chapter 8		
PROGRAMS		
8.1	Syntax.....	8-3
8.2	Program-Parameters	8-3
8.3	Segmentation	8-3
Chapter 9		
UNITS		
9.1	Regular-Units	9-3
9.2	Intrinsic-Units.....	9-6
9.3	Units that Use Other Units	9-6
Chapter 10		
INPUT/OUTPUT		
10.1	Introduction to I/O	10-3
10.2	Record-Oriented I/O.....	10-10
10.3	Text-Oriented I/O	10-12
10.4	Untyped File I/O	10-21
Chapter 11		
STANDARD PROCEDURES AND FUNCTIONS		
11.1	Exit and Halt Procedures.....	11-3
11.2	Dynamic Allocation Procedures	11-4
11.3	Transfer Functions	11-6
11.4	Arithmetic Functions	11-7
11.5	Ordinal Functions	11-10
11.6	String Procedures and Functions	11-11
11.7	Byte-Oriented Procedures and Functions.....	11-13
11.8	Packed Array of Char Procedures and Functions	11-14
Chapter 12		
THE COMPILER		
12.1	Compiler Commands	12-3
12.2	Conditional Compilation	12-4
12.3	Optimization of If-Statements	12-7
12.4	Optimization of While-Statements and Repeat-Statements	12-8
12.5	Efficiency of Case-Statements	12-8

Appendix A
COMPARISON WITH APPLE II AND APPLE III PASCAL

Appendix B
KNOWN ANOMALIES IN THE COMPILER

Appendix C
SYNTAX OF THE LANGUAGE

Appendix D
FLOATING-POINT ARITHMETIC

Appendix E
QUICKDRAW

Appendix F
HARDWARE INTERFACE

INDEX

TABLES

5-1	Precedence of Operations	5-3
5-2	Binary Arithmetic Operations	5-7
5-3	Unary Arithmetic Operations (Signs).....	5-7
5-4	Boolean Operations	5-9
5-5	Set Operations.....	5-9
5-6	Relational Operations	5-10
5-7	Pointer Operation	5-11
10-1	Combinations of File Variable Types with External File Species and Categories	10-5
D-1	Results of Addition and Subtraction on Infinities	D-4
D-2	Results of Multiplication and Division on Infinities	D-5



SYNTAX DIAGRAMS

A, B

actual-parameter	5-13
actual-parameter-list	5-13
array-type	3-11
assignment-statement	6-3
base-type	3-16
block	2-3

C

case	6-8
case-statement	6-8
compound-statement	6-6
conditional-statement	6-7
constant	1-7
constant-declaration	1-7
constant-declaration-part	2-4
control-variable	6-11

D, E, F

digit-sequence	1-4
enumerated-type	3-8
expression	5-6
factor	5-4
field-declaration	3-12
field-designator	4-6
field-list	3-12
file-buffer-symbol	4-6
file-type	3-15
final-value	6-11
fixed-part	3-12
for-statement	6-11
formal-parameter-list	7-8
function-body	7-6
function-call	5-13
function-declaration	7-6
function-heading	7-6

G, H, I

goto-statement	6-5
----------------------	-----

hex-digit-sequence	1-5
identifier	1-4
identifier-list	3-8
if-statement	6-7
implementation-part	9-4
index	4-5
index-type	3-11
initial-value	6-11
interface-part	9-4

L, M, O

label	2-3, 6-3
label-declaration-part	2-3
member-group	5-14
ordinal-type	3-4
otherwise-clause	6-8

P

parameter-declaration	7-8
pointer-object-symbol	4-7
pointer-type	3-16
procedure-and-function-declaration-part	2-4
procedure-body	7-3
procedure-declaration	7-3
procedure-heading	7-3
procedure-statement	6-4
program	8-3
program-heading	8-3
program-parameters	8-3

Q, R

qualifier	4-4
quoted-character-constant	1-7
quoted-string-constant	1-6
real-type	3-4
record-type	3-12
regular-unit	9-3
repeat-statement	6-9
repetitive-statement	6-9
result-type	7-6

S

scale-factor	1-5
set-constructor	5-14

set-type	3-14
sign	1-5
signed-number	1-5
simple-expression	5-5
simple-statement	6-3
simple-type	3-4
size-attribute	3-7
statement	6-3
statement-part	2-5
string-character	1-6
string-type	3-7
structured-statement	6-6
structured-type	3-10
subrange-type	3-9

T

tag-field-type	3-13
term	5-5
type	3-3
type-declaration	3-3
type-declaration-part	2-4

U

unit-heading	9-3
unsigned-constant	5-4
unsigned-integer	1-5
unsigned-number	1-5
unsigned-real	1-5
uses-clause	8-3

V, W

variable-declaration	4-3
variable-declaration-part	2-4
variable-identifier	4-4
variable-reference	4-3
variant	3-13
variant-part	3-13
while-statement	6-10
with-statement	6-13

PREFACE

This manual is intended for Pascal programmers. It describes an implementation of Pascal for the Lisa computer. The compiler and code generator translate Pascal source text to MC68000 object code.

The language is reasonably compatible with Apple II and Apple III Pascal. See Appendix A for a discussion of the differences between these forms of Pascal.

In addition to providing nearly all the features of standard Pascal, as described in the *Pascal User Manual and Report* (Jensen and Wirth), this Pascal provides a variety of extensions. These are summarized in Appendix A. They include 32-bit integers, an *otherwise* clause in case statements, procedural and functional parameters with type-checked parameter lists, and the @ operator for obtaining a pointer to an object. The real arithmetic conforms to many aspects of the proposed IEEE standard for single-precision arithmetic.

Operating Environment

The compiler will operate in any standard Lisa hardware configuration; this manual assumes the Workshop software environment.

Related Documents

Pascal User Manual and Report, Jensen and Wirth, Springer-Verlag 1975.

Workshop Reference Manual for the Lisa, Apple Computer, Inc. 1983.

Other Lisa documentation.

Definitions

For the purposes of this manual the following definitions are used:

- **Error**: Either a run-time error or a compiler error.
- **Scope**: The body of text for which the declaration of an identifier or label is valid.
- **Undefined**: The value of a variable or function when the variable does not necessarily have a meaningful value of its type assigned to it.
- **Unspecified**: A value or action or effect that, although possibly well-defined, is not specified and may not be the same in all cases or for all versions or configurations of the system. Any programming construct that leads to an unspecified result or effect is not supported.

Notation and Syntax Diagrams

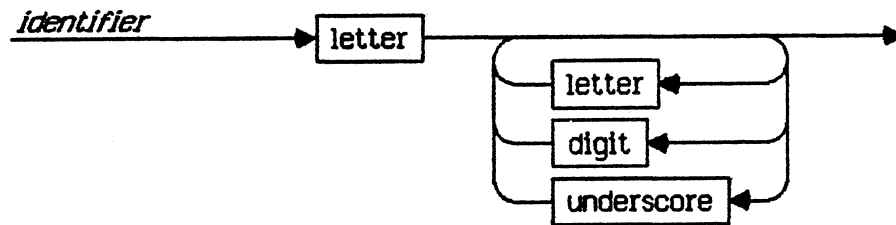
All numbers in this manual are in decimal notation, except where hexadecimal notation is specifically indicated.

Throughout this manual, bold-face type is used to distinguish Pascal text from English text. For example, **sqr(n div 16)** represents a fragment of a Pascal

program. Sometimes the same word appears both in plain text and in bold-face; for example, "The declaration of a Pascal procedure begins with the word **procedure**."

Underlining is used when technical terms are introduced.

Pascal syntax is specified by diagrams. For example, the following diagram gives the syntax for an identifier:

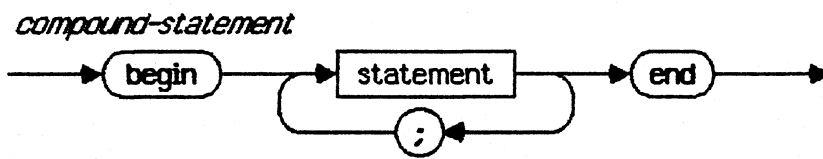


Start at the left and follow the arrows through the diagram. Numerous paths are possible. Every path that begins at the left and ends at the arrow-head on the right is valid, and represents a valid way to construct an identifier. The boxes traversed by a path through the diagram represent the elements that can be used to construct an identifier. Thus the diagram embodies the following rules:

- An identifier must begin with a letter, since the first arrow goes directly to a box containing the name "letter."
- An identifier might consist of nothing but a single letter, since there is a path from this box to the arrow-head on the right, without going through any more boxes.
- The initial letter may be followed by another letter, a digit, or an underscore, since there are branches of the path that lead to these boxes.
- The initial letter may be followed by any number of letters, digits, or underscores, since there is a loop in the path.

A word contained in a rectangular box may be a name for an atomic element like "letter" or "digit," or it may be a name for some other syntactic construction that is specified by another diagram. The name in a rectangular box is to be replaced by an actual instance of the atom or construction that it represents, e.g. "3" for "digit" or "counter" for "variable-reference".

Pascal symbols, such as reserved words, operators, and punctuation, are bold-face and are enclosed in circles or ovals, as in the following diagram for the construction of a compound-statement:



Text in a circle or oval represents itself, and is to be written as shown (except that capitalization of letters is not significant). In the diagram above, the semicolon and the words **begin** and **end** are symbols. The word "statement" refers to a construction that has its own syntax diagram.

A compound-statement consists of the reserved word **begin**, followed by any number of statements separated by semicolons, followed by the reserved word **end**. (As will be seen in Chapter 6, a statement may be null; thus **begin end** is a valid compound-statement.)

Chapter 1

TOKENS AND CONSTANTS

1.1	Character Set and Special Symbols	1-3
1.2	Identifiers	1-4
1.3	Directives	1-4
1.4	Numbers	1-4
1.5	Labels	1-6
1.6	Quoted String Constants	1-6
1.6.1	Quoted Character Constants	1-6
1.7	Constant Declarations	1-7
1.8	Comments and Compiler Commands	1-7

TOKENS AND CONSTANTS

Tokens are the smallest meaningful units of text in a Pascal program; structurally, they correspond to the words in an English sentence. The tokens of Pascal are classified into special symbols, identifiers, numbers, labels, and quoted string constants.

The text of a Pascal program consists of tokens and separators; a separator is either a blank or a comment. Two adjacent tokens must be separated by one or more separators, if both tokens are identifiers, numbers, or reserved words.

No separators can be embedded within tokens, except in quoted string constants.

1.1 Character Set and Special Symbols

The character set used by Pascal on the Lisa is 8-bit extended ASCII, with characters represented by numeric codes in the range from 0 to 255.

Letters, digits, hex-digits, and blanks are subsets of the character set:

- The letters are those of the English alphabet, A through Z and a through z.
- The digits are the Arabic numerals 0 through 9; the hex-digits are the Arabic numerals 0 through 9, the letters A through F, and the letters a through f.
- The blanks are the space character (ASCII 32), the horizontal tab character (ASCII 9), and the CR character (ASCII 13).

Special symbols and reserved words are tokens having one or more fixed meanings. The following single characters are special symbols:

+ - * / = < > [] . , () : ; ^ @ { } \$

The following character pairs are special symbols:

<> <= >= := .. (* *)

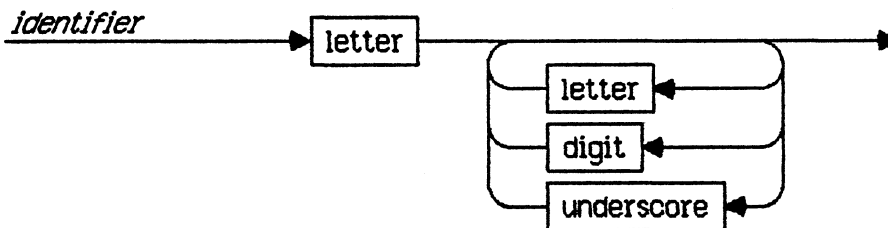
The following are the reserved words:

and	end	label	program	until
array	file	methods*	record	uses
begin	for	mod	repeat	var
case	function	nil	set	while
const	goto	not	string	with
creation*	if	of	subclass*	
div	implementation	or	then	
downto	in	otherwise	to	
do	interface	packed	type	
else	intrinsic*	procedure	unit	

The reserved words marked with asterisks are reserved for future use. Corresponding upper and lower case letters are equivalent in reserved words. Only the first 8 characters of a reserved word are significant.

1.2 Identifiers

Identifiers serve to denote constants, types, variables, procedures, functions, units and programs, and fields in records. Identifiers can be of any length, but only the first 8 characters are significant. Corresponding upper and lower case letters are equivalent in identifiers.



NOTE

The first 8 characters of an identifier must not match the first 8 characters of a reserved word.

Examples of identifiers:

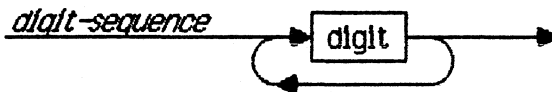
X Rome gcd SUM get_byte

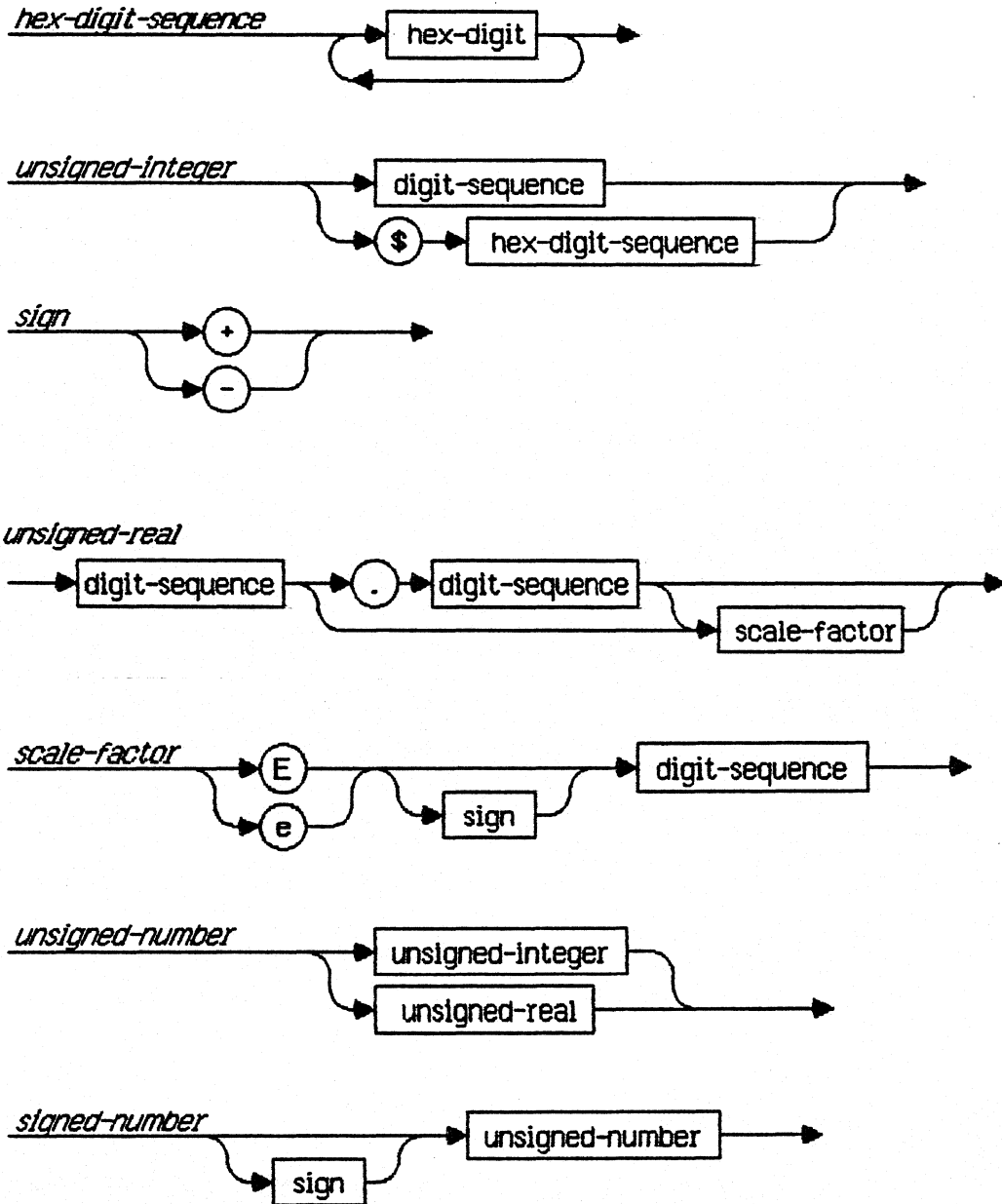
1.3 Directives

Directives are words that have special meanings in particular contexts. They are not reserved and can be used as identifiers in other contexts. For example, the word **forward** is interpreted as a directive if it occurs immediately after a procedure-heading or function-heading, but in any other position it is interpreted as an identifier.

1.4 Numbers

The usual decimal notation is used for numbers that are constants of the data types **integer**, **longint**, and **real** (see Section 3.1.1). Also, a hexadecimal integer constant can be written by using the **\$** character as a prefix.





The letter E or e preceding the scale in an unsigned-real means "times ten to the power of".

Examples of numbers:

1 +100 -0.1 5E-3 87.35e+8 \$A05D

Note that 5E-3 means 5×10^{-3} , and 87.35e+8 means 87.35×10^8 .

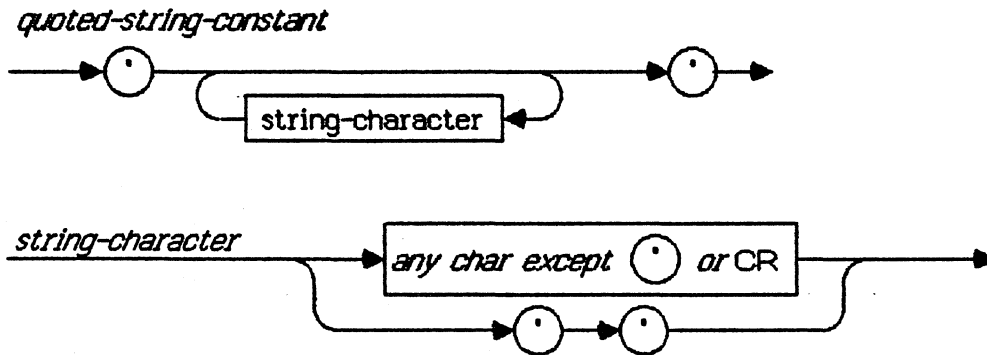
1.5 Labels

A label is a digit-sequence in the range from 0 through 9999.

1.6 Quoted String Constants

A quoted-string-constant is a sequence of zero or more characters, all on one line of the program source text and enclosed by apostrophes. Currently, the maximum number of characters is 255. A quoted-string-constant with nothing between the apostrophes denotes the null string.

If the quoted-string-constant is to contain an apostrophe, this apostrophe must be written twice.



Examples of quoted-string-constants:

'Pascal' 'THIS IS A STRING' 'Don't worry!'
'A' ';'

All string values have a *length* attribute (see Section 3.1.1.6). In the case of a string constant value the length is fixed; it is equal to the actual number of characters in the string value.

1.6.1 Quoted Character Constants

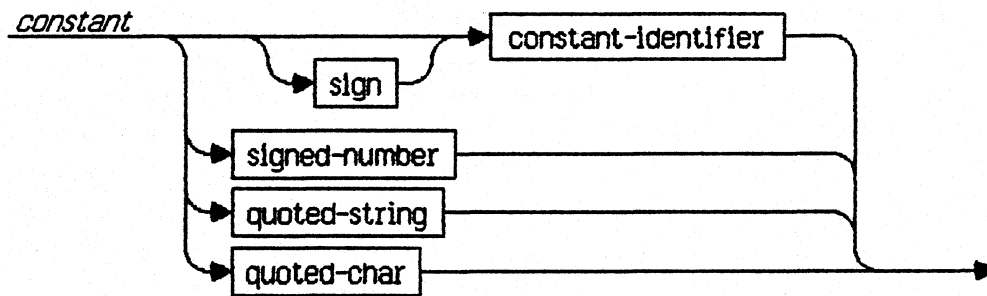
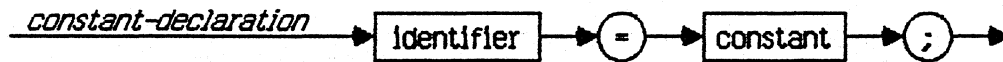
Syntactically, a quoted-character-constant is simply a quoted-string-constant whose length is exactly 1.



A quoted-character-constant is compatible with any char-type or string-type; that is, it can be used either as a character value or as a string value.

1.7 Constant Declarations

A constant-declaration defines an identifier to denote a constant, within the block that contains the declaration. The scope of a constant-identifier (see Chapter 2) does not include its own declaration.



Note: A constant-identifier is an identifier that has already been declared to denote a constant.

A constant-identifier following a sign must denote a value of type *Integer*, *longint*, or *real*.

1.8 Comments and Compiler Commands

The constructs:

```
{ any-text-not-containing-right-brace }
(* any-text-not-containing star-right-paren *)
```

are called comments.

A compiler command is a comment that contains a \$ character immediately after the { or (* that begins the comment. The \$ character is followed by the mnemonic of the compiler command (see Section 12).

Apart from the effects of compiler commands, the substitution of a blank for a comment does not alter the meaning of a program.

A comment cannot be nested within another comment formed with the same kind of delimiters. However, a comment formed with {...} delimiters can be nested within a comment formed with (*...*) delimiters, and vice versa.

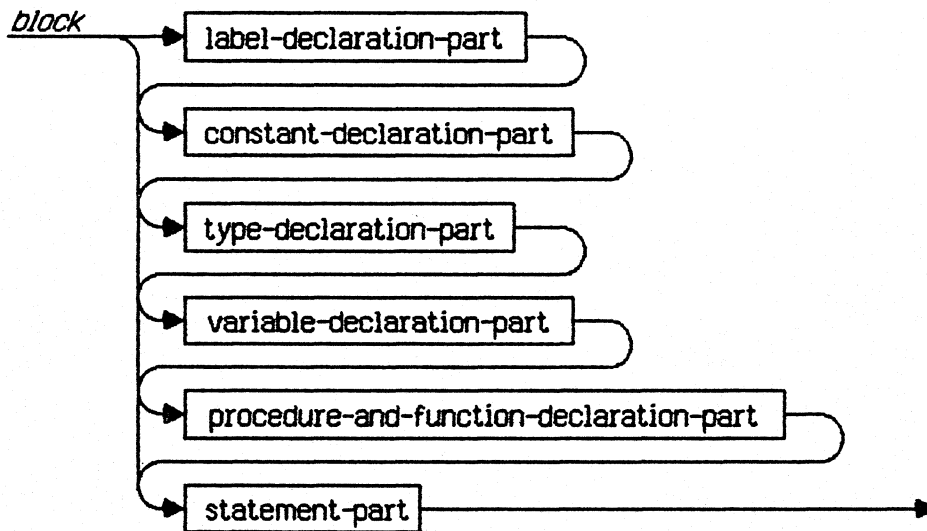
Chapter 2 BLOCKS, LOCALITY, AND SCOPE

2.1	Definition of a Block	2-3
2.2	Rules of Scope	2-5
2.2.1	Scope of a Declaration	2-5
2.2.2	Redeclaration in an Enclosed Block	2-5
2.2.3	Position of Declaration within its Block	2-5
2.2.4	Redeclaration within a Block	2-5
2.2.5	Identifiers of Standard Objects	2-6

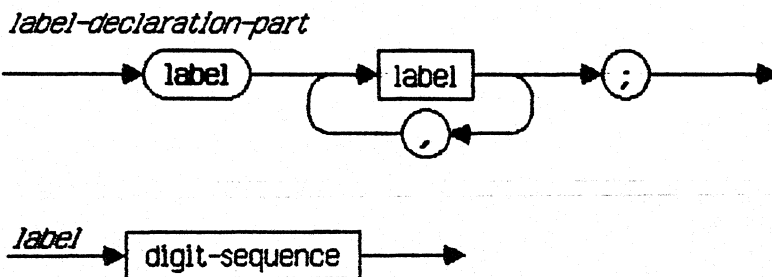
BLOCKS, LOCALITY, AND SCOPE

2.1 Definition of a Block

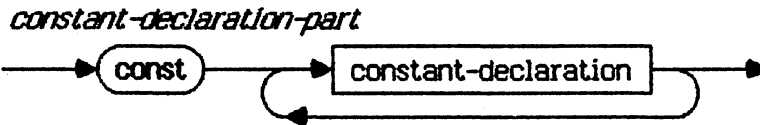
A **block** consists of declarations and a statement-part. Every block is part of a procedure-declaration, a function-declaration, a program, or a unit. All identifiers and labels that are declared in a particular block are **local** to that block.



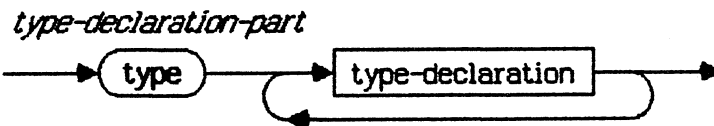
The label-declaration-part declares all labels that mark statements in the corresponding statement-part. Each label must mark exactly one statement in the statement-part.



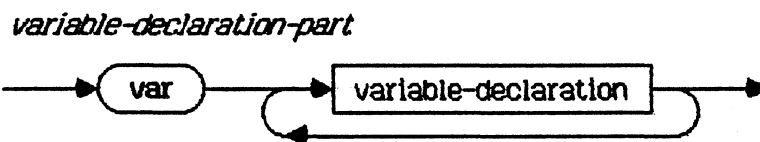
The constant-declaration-part contains all constant-declarations local to the block.



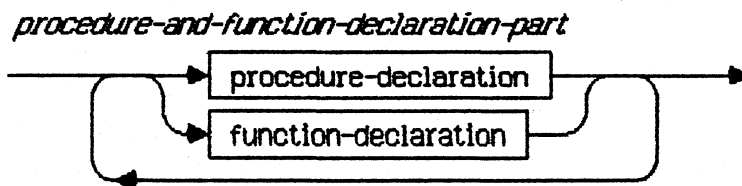
The type-declaration-part contains all type-declarations local to the block.



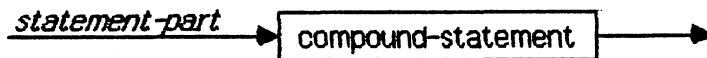
The variable-declaration-part contains all variable-declarations local to the block.



The procedure-and-function-declaration-part contains all procedure and function declarations local to the block.



The statement-part specifies the algorithmic actions to be executed upon an activation of the block.

**NOTE**

At run time, all variables declared within a particular block have unspecified values each time the statement-part of the block is entered.

2.2 Rules of Scope

This chapter discusses the scope of objects *within the program or unit in which they are defined*. See Chapter 9 for the scope of objects defined in the interface-part of a unit and referenced in a host program or unit.

2.2.1 Scope of a Declaration

The appearance of an identifier or label in a declaration defines the identifier or label. All corresponding occurrences of the identifier or label must be within the scope of this declaration.

This scope is the block that contains the declaration, and all blocks enclosed by that block except as explained in Section 2.2.2 below.

2.2.2 Redeclaration in an Enclosed Block

Suppose that **outer** is a block, and **inner** is another block that is enclosed within **outer**. If an identifier declared in block **outer** has a further declaration in block **inner**, then block **inner** and all blocks enclosed by **inner** are excluded from the scope of the declaration in block **outer**. (See Appendix B for some odd cases.)

2.2.3 Position of Declaration within its Block

The declaration of an identifier or label must precede all corresponding occurrences of that identifier or label in the program text -- i.e., identifiers and labels cannot be used until after they are declared.

There is one exception to this rule: The base-type of a pointer-type (see Section 3.3) can be an identifier that has not yet been declared. In this case, the identifier must be declared somewhere in the same type-declaration-part in which the pointer-type occurs. (See Appendix B for some odd cases.)

2.2.4 Redeclaration within a Block

An identifier or label cannot be declared more than once in the outer level of a particular block, except for record field identifiers.

A record field identifier (see Sections 3.2.2, 4.3, and 4.3.2) is declared within a record-type. It is meaningful only in combination with a reference to a variable of that record-type. Therefore a field identifier can be declared again within the same block, as long as it is not declared again at the same level within the same record-type. Also, an identifier that has been declared to denote a

constant, a type, or a variable can be declared again as a record field identifier in the same block.

2.2.5 Identifiers of Standard Objects

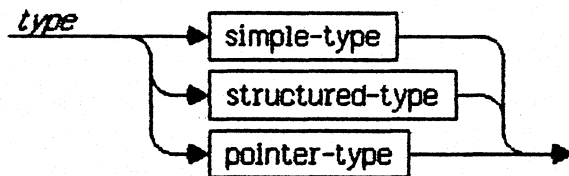
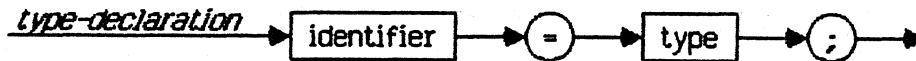
Pascal on the Lisa provides a set of *standard* (predeclared) constants, types, procedures, and functions. The identifiers of these objects behave as if they were declared in an outermost block enclosing the entire program; thus their scope includes the entire program.

Chapter 3 DATA TYPES

3.1	Simple-Types (and Ordinal-Types)	3-4
3.1.1	Standard Simple-Types and String-Types	3-5
3.1.1.1	The Integer Type	3-5
3.1.1.2	The LongInt Type	3-5
3.1.1.3	The Real Type	3-6
3.1.1.4	The Boolean Type	3-7
3.1.1.5	The Char Type	3-7
3.1.1.6	String-Types	3-7
3.1.2	Enumerated-Types	3-8
3.1.3	Subrange-Types	3-9
3.2	Structured-Types	3-9
3.2.1	Array-Types	3-10
3.2.2	Record-Types	3-12
3.2.3	Set-Types	3-14
3.2.4	File-Types	3-15
3.3	Pointer-Types	3-16
3.4	Identical and Compatible Types	3-16
3.4.1	Type Identity	3-16
3.4.2	Compatibility of Types	3-18
3.4.3	Assignment-Compatibility	3-18
3.5	The Type-Declaration-Part	3-19

DATA TYPES

A **type** is used in declaring variables; it determines the set of values which those variables can assume, and the operations that can be performed upon them. A **type-declaration** associates an identifier with a type.



The occurrence of an identifier on the left-hand side of a type-declaration declares it as a **type-identifier** for the block in which the type-declaration occurs. The scope of a type-identifier does not include its own declaration, except for pointer-types (see Sections 2.2.3 and 3.3).

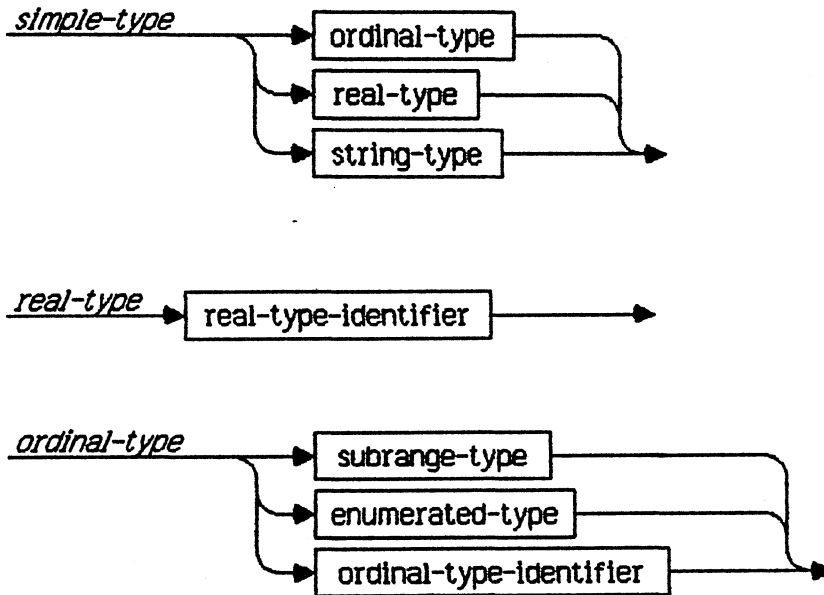
To help clarify the syntax description with some semantic hints, the following terms are used to distinguish identifiers according to what they denote. Syntactically, all of them mean simply an identifier:

simple-type-identifier
 structured-type-identifier
 pointer-type-identifier
 ordinal-type-identifier
 real-type-identifier
 string-type-identifier

In other words, a **simple-type-identifier** is any identifier that is declared to denote a simple type, a **structured-type-identifier** is any identifier that is declared to denote a structured type, and so forth. A **simple-type-identifier** can be the predeclared identifier of a standard type such as **integer**, **boolean**, etc.

3.1 Simple-Types (and Ordinal-Types)

All the simple-types define ordered sets of values.



The standard real-type-identifier is `real`.

String-types are discussed in Section 3.1.1.6 below.

Ordinal-types are a subset of the simple-types, with the following special characteristics:

- Within a given ordinal-type, the possible values are an ordered set and each possible value is associated with an ordinality, which is an integer value. The first value of the ordinal-type has ordinality 0, the next has ordinality 1, etc. Each possible value except the first has a predecessor based on this ordering, and each possible value except the last has a successor based on this ordering.
- The standard function `ord` (see Section 11.5.1) can be applied to any value of ordinal-type, and returns the ordinality of the value.
- The standard function `pred` (see Section 11.5.4) can be applied to any value of ordinal-type, and returns the predecessor of the value. (For the first value in the ordinal-type, the result is unspecified.)

- The standard function `succ` (see Section 11.5.3) can be applied to any value of ordinal-type, and returns the successor of the value. (For the first value in the ordinal-type, the result is unspecified.)

All simple-types except `real` and the string-types are ordinal-types. The standard ordinal-type-identifiers are

```
integer
longint
char
boolean
```

Note that in addition to these standard types, the enumerated-types and subrange-types are ordinal-types.

3.1.1 Standard Simple-Types and String-Types

A standard type is denoted by a predefined type-identifier. The simple-types `integer`, `longint`, `real`, `char`, and `boolean` are standard. The string-types are user-defined simple-types.

3.1.1.1 The Integer Type

The values are a subset of the whole numbers. (As constants, these values can be denoted as specified in Section 1.4.) The predefined `integer` constant `maxint` is defined to be 32767. `Maxint` defines the range of the type `integer` as the set of values:

-maxint-1, -maxint, ... -1, 0, 1, ... maxint-1, maxint

These are 16-bit, 2's-complement integers.

3.1.1.2 The Longint Type

The values are a subset of the whole numbers. (As constants, these values can be denoted as specified in Section 1.4.) The range is the set of values from $-(2^{31}-1)$ to $2^{31}-1$, i.e., -2147483648 to 2147483647.

These are 32-bit integers.

Arithmetic on `integer` and `longint` operands is done in both 16-bit and 32-bit precision. An expression with mixed operand sizes is evaluated in a manner similar to the FORTRAN single/double precision floating-point arithmetic rules:

- All "integer" constants in the range of type `integer` are considered to be of type `integer`. All "integer" constants in the range of type `longint`, but not in the range of type `integer`, are considered to be of type `longint`.
- When both operands of an operator (or the single operand of a unary operator) are of type `integer`, 16-bit operations are always performed and the result is of type `integer` (truncated to 16 bits if necessary).

- When one or both operands are of type `longint`, all operands are first converted to type `longint`, 32-bit operations are performed, and the result is of type `longint`. However, if this value is assigned to a variable of type `integer`, it is truncated (see next rule).
- The expression on the right of an assignment statement is evaluated independently of the size of the variable on the left. If necessary, the result of the expression is truncated or extended to match the size of the variable on the left.

The `ord4` function (see Section 11.3.3) can be used to convert an `integer` value to a `longint` value.

IMPLEMENTATION NOTE

There is a performance penalty for the use of `longint` values. The penalty is essentially a factor of 2 for operations other than division and multiplication; for division and multiplication, the penalty is much worse than a factor of 2.

3.1.1.3 The Real Type

For details of IEEE standard floating-point arithmetic, see Appendix D. The possible real values are

- Finite values (a subset of the mathematical real numbers). As constants, these values can be denoted as specified in Section 1.4.

The largest absolute numeric real value is approximately 3.402823466E38 in Pascal notation.

The smallest absolute numeric non-zero real value is approximately 1.401298464E-45 in Pascal notation.

The real zero value has a sign, like other numbers. However, the sign of a zero value is disregarded except in division of a finite number by zero and in textual output.

- Infinite values, $+\infty$ and $-\infty$. These arise either as the result of an operation that overflows the maximum absolute finite value, or as the result of dividing a finite value by zero. Appendix D gives the rules for arithmetic operations using these values.
- NaNs (the word "NaN" stands for "not a number"). These are values of type `real` that convey diagnostic information. For example, the result of multiplying ∞ by 0 is a NaN.

3.1.1.4 The Boolean Type

The values are truth values denoted by the predefined constant identifiers `false` and `true`. These values are ordered so that `false` is "less than" `true`. The function-call `ord(false)` returns 0, and `ord(true)` returns 1 (see Section 11.5.1).

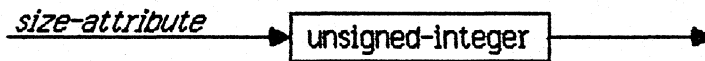
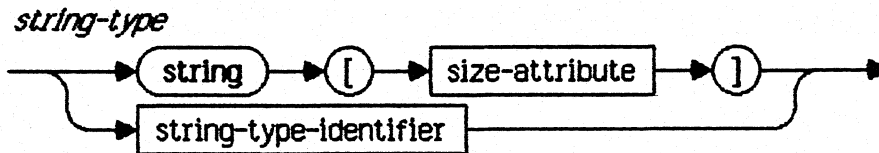
3.1.1.5 The Char Type

The values are extended 8-bit ASCII, represented by numeric codes in the range 0..255. The ordering of the `char` values is defined by the ordering of these numeric codes. The function-call `ord(c)`, where `c` is a `char` value, returns the numeric code of `c` (see Section 11.5.1).

3.1.1.6 String-Types

A string value is a sequence of characters that has a dynamic `length` attribute. The length is the actual number of characters in the sequence at any time during program execution.

A string type has a static `size` attribute. The size is the maximum limit on the length of any value of this type. The current value of the length attribute is returned by the standard function `length` (see Section 11.6); the size attribute of a string type is determined when the string type is defined.



where the size attribute is an unsigned-integer.

IMPLEMENTATION NOTE

In the current implementation, the size-attribute must be in the range from 1 to 255.

The ordering relationship between any two string values is determined by lexical comparison based on the ordering relationship between character values in corresponding positions in the two strings. (When the two strings are of unequal lengths, each character in the longer string that does not correspond

to a character in the shorter one compares "higher"; thus the string 'attribute' is ordered higher than 'at'.)

Do not confuse the size with the length.

NOTES

The size attribute of a string *constant* is equal to the length of the string constant value, namely the number of characters actually in the string.

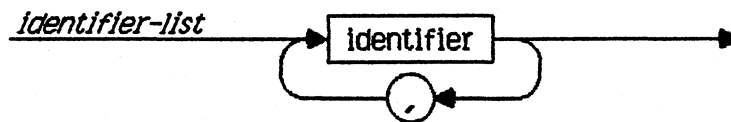
Although string-types are simple-types by definition, they have some characteristics of structured-types. As explained in Section 4.3.1, individual characters in a string can be accessed as if they were components of an array. Also, all string-types are implicitly packed types and all restrictions on packed types apply to strings (see Sections 7.3.2, 5.1.6.1, and 11.7).

Do not make any assumptions about the internal storage format of strings, as this format may not be the same in all implementations.

Operators applicable to strings are specified in Section 5.1.5. Standard procedures and functions for manipulating strings are described in Section 11.6.

3.1.2 Enumerated-Types

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



The occurrence of an identifier within the identifier-list of an enumerated-type declares it as a constant for the block in which the enumerated-type is declared. The type of this constant is the enumerated-type being declared.

Examples of enumerated-types:

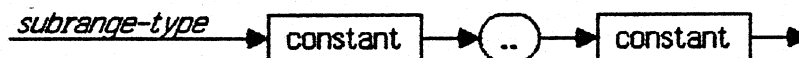
```
color = (red, yellow, green, blue)
suit  = (club, diamond, heart, spade)
maritalStatus = (married, divorced, widowed, single)
```

Given these declarations, `yellow` is a constant of type `color`, `diamond` is a constant of type `suit`, and so forth.

When the `ord` function (see Section 11.5.1) is applied to a value of an enumerated-type, it returns an integer representing the ordering of the value with respect to the other values of the enumerated-type. For example, given the declarations above, `ord(red)` returns 0, `ord(yellow)` returns 1, and `ord(blue)` returns 3.

3.1.3 Subrange-Types

A subrange-type provides for range-checking of values within some ordinal-type. The syntax for a subrange-type is



Both constants must be of ordinal-type. Both constants must either be of the same ordinal-type, or one must be of type `integer` and the other of type `longint`. If both are of the same ordinal-type, this type is called the host-type. If one is of type `integer` and the other of type `longint`, the host-type is `longint`.

Examples of subrange-types:

```
1..100
-10..+10
red..green
```

A variable of subrange-type possesses all the properties of variables of the host type, with the restriction that its run-time value must be in the specified closed interval.

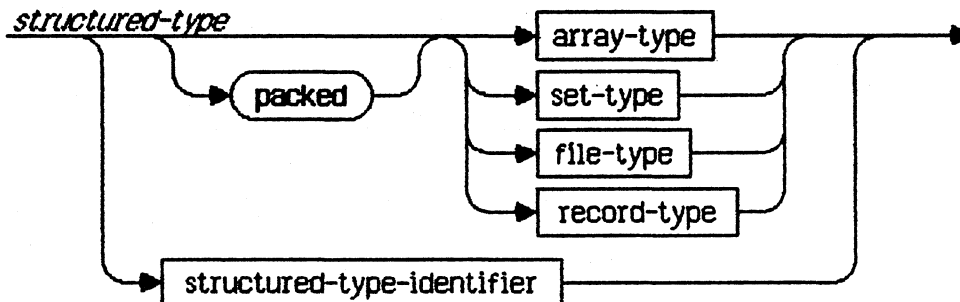
IMPLEMENTATION NOTE

Range-checking is enabled and disabled by the compiler commands `$R+` and `$R-` (see Chapter 12). The default is `$R+` (range-checking enabled).

3.2 Structured-Types

A structured-type is characterized by its structuring method and by the type(s) of its components. If the component type is itself structured, the resulting

structured-type exhibits more than one level of structuring. There is no specified limit on the number of levels to which data-types can be structured.



The use of the word **packed** in the declaration of a structured-type indicates to the compiler that data storage should be economized, even if this causes an access to a component of a variable of this type to be less efficient.

The word **packed** only affects the representation of one level of the structured-type in which it occurs. If a component is itself structured, the component's representation is packed only if the word **packed** also occurs in the declaration of its type.

For restrictions on the use of components of packed variables, see Sections 7.3.2, 5.1.6.1, and 11.7.

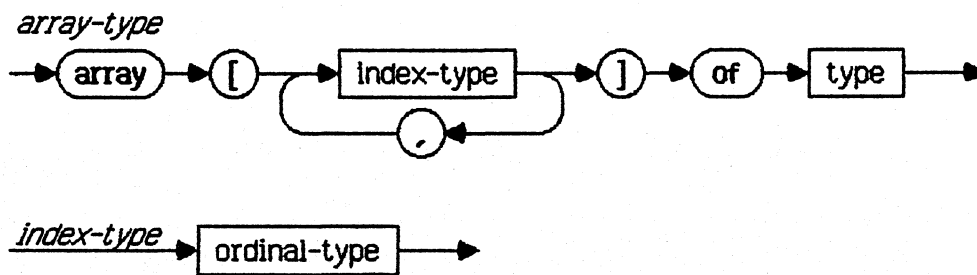
The implementation of packing is complex, and details of the allocation of storage to components of a packed variable are *unspecified*.

IMPLEMENTATION NOTE

In the current implementation, the word **packed** has no effect on types other than array and record.

3.2.1 Array-Types

An array-type consists of a fixed number of components that are all of one type, called the component-type. The number of elements is determined by one or more index-types, one for each dimension of the array. There is no specified limit on the number of dimensions. In each dimension, the array can be indexed by every possible value of the corresponding index-type, so the number of elements is the product of the cardinalities of all the index-types.



The type following the word *of* is the component-type of the array.

IMPLEMENTATION NOTE

In the current implementation, the index-type should not be `longint` or a subrange of `longint`, and arrays should not contain more than 32767 bytes.

Examples of array-types:

```

array[1..100] of real
array[boolean] of color
  
```

If the component-type of an array-type is also an array-type, the result can be regarded as a single multi-dimensional array. The declaration of such an array is equivalent to the declaration of a multi-dimensional array, as illustrated by the following examples:

```

array[boolean] of array[1..10] of array[size] of real
  
```

is equivalent to:

```

array[boolean, 1..10, size] of real
  
```

Likewise,

```

packed array[1..10] of packed array[1..8] of boolean
  
```

is equivalent to:

```

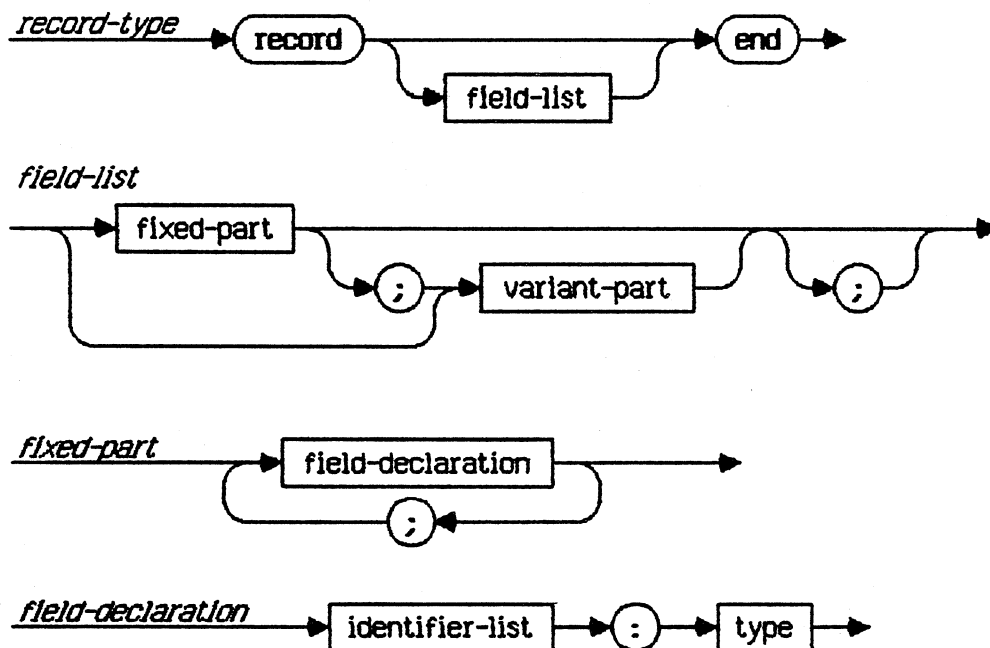
packed array[1..10, 1..8] of boolean
  
```

"Equivalent" means that the compiler does the same thing with the two constructions.

A component of an array can be accessed by referencing the array and applying one or more indexes (see Section 4.3.1).

3.2.2 Record-Types

A record-type consists of a fixed number of components called *fields*, possibly of different types. For each component, the record-type declaration specifies the type of the field and an identifier that denotes it.



The fixed-part of a record-type specifies a list of "fixed" fields, giving an identifier and a type for each field. Each of these fields contains data that is always accessed in the same way (see Section 4.3.2).

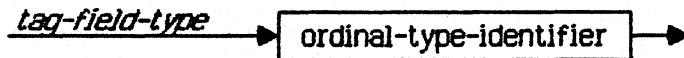
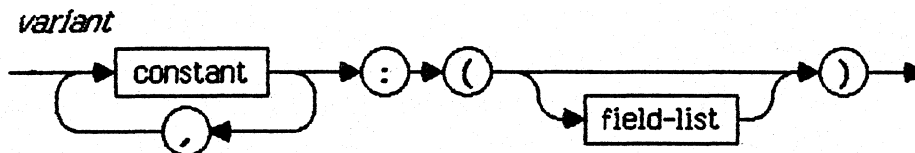
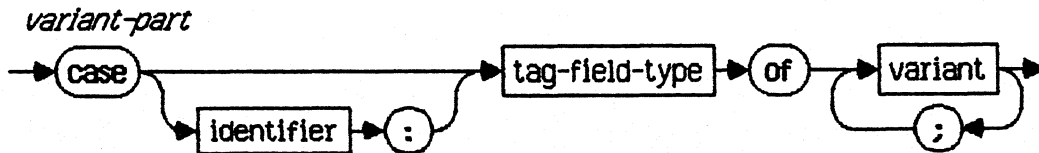
Example of a record-type:

```

record
  year: integer;
  month: 1..12;
  day: 1..31
end
  
```

A variant-part allocates memory space with more than one list of fields, thus permitting the data in this space to be accessed in more than one way. Each list

of fields is called a variant. The variants "overlay" each other in memory, and all fields of all variants are accessible at all times.



IMPLEMENTATION NOTE

In the current implementation, the type `longint` should not be used as a tag-type as it will not work correctly.

Each variant is introduced by one or more constants. All the constants must be distinct and must be of an ordinal-type that is compatible with the tag-type (see Section 3.4).

The variant-part allows for an optional identifier, called the tag-field identifier. If a tag-field identifier is present, it is automatically declared as the identifier of an additional fixed field of the record, called the tag-field.

The value of the tag-field may be used by the program to indicate which variant should be used at a given time. If there is no tag-field, then the program must select a variant on some other criterion.

Examples of record-types with variants:

```

record
  name, firstName: string[80];
  age: 0..99;
  case married: boolean of
    true: (spousesName: string[80]);
    false: ()
end

record
  x, y: real;
  area: real;
  case s: shape of
    triangle: (side: real; inclination, angle1, angle2:
              angle);
    rectangle: (side1, side2 : real; skew, angle3: angle);
    circle: (diameter: real);
end

```

NOTE

The constants that introduce a variant are not used for referring to fields of the variant; however, they can be used as optional arguments of the new procedure (see Section 11.2). Variant fields are accessed in exactly the same way as fixed fields (see Section 4.3.2).

3.2.3 Set-Types

A set-type defines a range of values that is the powerset of some ordinal-type, called the base-type. In other words, each possible value of a set-type is some subset of the possible values of the base-type.



IMPLEMENTATION NOTE

In the present implementation the base-type must not be `longint`. The base-type must not have more than 4088 possible values. If the base-type is a subrange of `integer`, it must be within the limits 0..4087.

Operators applicable to sets are specified in Section 5.1.4. Section 5.3 shows how set values are denoted in Pascal.

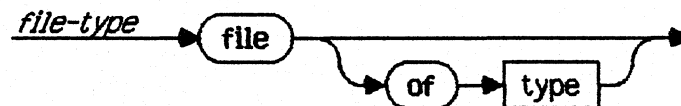
Sets with less than 32 possible values in the base-type can be held in a register and offer the best performance. For sets larger than this, there is a performance penalty that is essentially a linear function of the size of the base-type.

The empty set (see Section 5.1.4) is a possible value of every set-type.

3.2.4 File-Types

A file-type is a structured-type consisting of a sequence of components that are all of one type, the component-type. The component-type may be any type.

The component data is not in program-addressable memory but is accessed via a peripheral device. The number of components (i.e. the length of the file) is not fixed by the file-type declaration.



The type `file` (without the "of type" construct) represents a so-called "untyped file" type for use with the `blockread` and `blockwrite` functions (see Section 10.4).

NOTE

Although the symbol `file` can be used as if it were a type-identifier, it cannot be redeclared since it is a reserved word.

The standard file-type `text` denotes a file of text organized into lines. The file may be stored on a file-structured device, or it may be a stream of characters from a character device such as the Lisa keyboard. Files of type `text` are supported by the specialized I/O procedures discussed in Section 10.3.

In Pascal on the Lisa, the type `text` is distinct from the type `file of char` (unlike standard Pascal). The type `file of char` is a file whose records are of type `char`, containing `char` values that are not interpreted or converted in any way during I/O operations.

In a stored file of type `text` or `file of -128..127`, the component values are packed into bytes on the storage medium. However, this does not apply to the type `file of char`; the component values of this type are stored in 16-bit words.

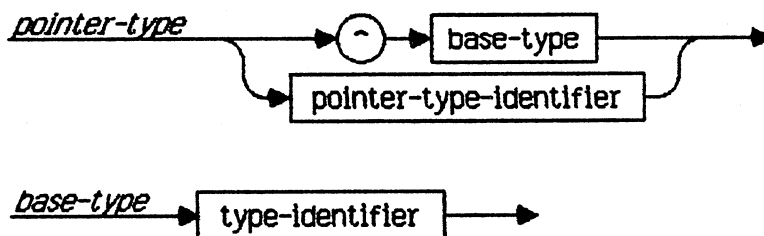
In Pascal on the Lisa, files can be passed to procedures and functions as variable parameters, as explained in Section 7.3.2.

Sections 4.3.3, 10.2, 10.3, and 10.4 discuss methods of accessing file components and data.

3.3 Pointer-Types

A pointer-type defines an unbounded set of values that point to variables of a specified type called the base-type.

Pointer values are created by the standard procedure `new` (see Section 11.2.1), by the `@` operator (see Section 5.1.6), and by the standard procedure `pointer` (see Section 11.3.4).



NOTE

The base-type may be an identifier that has not yet been declared. In this case, it must be declared somewhere in the same block as the pointer-type.

The special symbol `nil` represents a standard pointer-valued constant that is a possible value of every pointer type. Conceptually, `nil` is a pointer that does not point to anything.

Section 4.3.4 discusses the syntax for referencing the object pointed to by a pointer variable.

3.4 Identical and Compatible Types

As explained below, this Pascal has stronger typing than standard Pascal. In Pascal on the Lisa, two types may or may not be identical, and identity is required in some contexts but not in others.

Even if not identical, two types may still be compatible, and this is sufficient in contexts where identity is not required — except for assignment, where assignment-compatibility is required.

3.4.1 Type Identity

Identical types are required *only* in the following contexts:

- Variable parameters (see Section 7.3.2).

- Result types of functional parameters (see Section 7.3.4).
- Value and variable parameters within parameter-lists of procedural or functional parameters (see Section 7.3.5).
- One-dimensional **packed** arrays of **char** being compared via a relational operator (see Section 5.1.5).

Two types, **t1** and **t2**, are identical if either of the following is true:

- The same *type identifier* is used to declare both **t1** and **t2**, as in

```
foo = ^integer;
t1 = foo;
t2 = foo;
```

- **t1** is declared to be equivalent to **t2** as in

```
t1 = t2;
```

Note that the declarations

```
t1 = t2;
t3 = t1;
```

do *not* make **t3** and **t2** identical, even though they make **t1** identical to **t2** and **t3** identical to **t1**!

Also note that the declarations

```
t4 = integer;
t5 = integer;
```

do make **t4** and **t5** identical, since both are defined by the same type identifier. In general, the declarations

```
t6 = t7;
t8 = t7;
```

do make **t6** and **t8** identical if **t7** is a type-identifier.

However, the declarations

```
t9 = ^integer;
t10 = ^integer;
```

do *not* make **t9** and **t10** identical since **^integer** is not a type identifier but a user-defined type consisting of the special symbol **^** and a type identifier.

Finally, note that two variables declared in the same declaration, as in

```
var1, var2: ^integer;
```

are of identical type. However, if the declarations are separate then the definitions above apply.

The declarations

```
var1: ^integer;
var2: ^integer;
var3: integer;
var4: integer;
```

make `var3` and `var4` identical in type, but not `var1` and `var2`.

3.4.2 Compatibility of Types

Compatibility is required in the majority of contexts where two or more entities are used together, e.g. in expressions. Specific instances where type compatibility is required are noted elsewhere in this manual.

Two types are compatible if any of the following are true:

- They are identical.
- One is a subrange of the other.
- Both are subranges of the same type.
- Both are string-types (the lengths and sizes may differ).
- Both are set-types, and their base-types are compatible.

3.4.3 Assignment-Compatibility

Assignment-compatibility is required whenever a value is assigned to something, either explicitly (as in an assignment-statement) or implicitly (as in passing value parameters).

The value of an expression `expval` of type `exptyp` is assignment-compatible with a variable, parameter, or function-identifier of type `vtyp` if any of the following is true.

- `vtyp` and `exptyp` are identical and neither is a file-type, or a structured-type with a file component.
- `vtyp` is real and `exptyp` is integer or longint (`expval` is coerced to type real).
- `vtyp` and `exptyp` are compatible ordinal-types, and `expval` is within the range of possible values of `vtyp`.
- `vtyp` and `exptyp` are compatible set-types, and all the members of `expval` are within the range of possible values of the base-type of `vtyp`.
- `vtyp` and `exptyp` are string types, and the current length of `expval` is equal to or less than the size-attribute of `vtyp`.
- `vtyp` is a string type or a char type and `expval` is a quoted-character-constant.

- `vtyp` is a packed array[1..*n*] of `char` and `expval` is a string constant containing exactly *n* characters.

If the index-type of the packed array of `char` is not 1..*n*, but the array does have exactly *n* elements, no error will occur. However, the results are unspecified.

Whenever assignment-compatibility is required and none of the above is true, either a compiler error or a run-time error occurs.

3.5 The Type-Declaration-Part

Any program, procedure, or function that declares types contains a type-declaration-part, as shown in Chapter 2.

Example of a type-declaration-part:

```

type count = integer;
range = integer;
color = (red, yellow, green, blue);
sex = (male, female);
year = 1900..1999;
shape = (triangle, rectangle, circle);
card = array[1..80] of char;
str = string[80];
polar = record r: real; theta: angle end;
person = ~personDetails;
personDetails = record
  name, firstName: str;
  age: integer;
  married: boolean;
  father, child, sibling: person;
  case s: sex of
    male: (enlisted, bearded: boolean);
    female: (pregnant: boolean)
  end;
people = file of personDetails;
intfile = file of integer;

```

In the above example `count`, `range`, and `integer` denote identical types. The type `year` is compatible with, but not identical to, the types `range`, `count`, and `integer`.

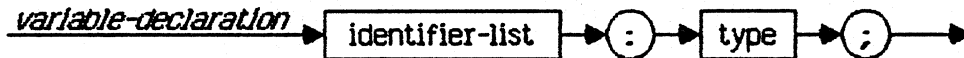
Chapter 4 VARIABLES

4.1	Variable-Declarations	4-3
4.2	Variable-References	4-3
4.3	Qualifiers	4-4
4.3.1	Arrays, Strings, and Indexes	4-5
4.3.2	Records and Field-Designators	4-6
4.3.3	File-Buffers	4-6
4.3.4	Pointers and Their Objects	4-7

VARIABLES

4.1 Variable-Declarations

A variable-declaration consists of a list of identifiers denoting new variables, followed by their type.



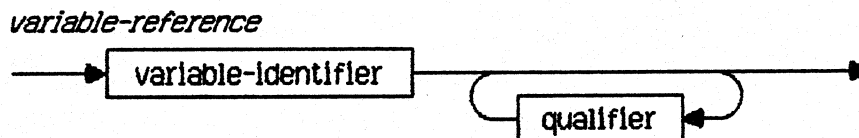
The occurrence of an identifier within the identifier-list of a variable-declaration declares it as a variable-identifier for the block in which the declaration occurs. The variable can then be referenced throughout the remaining lexical extent of that block, except as specified in Section 2.2.2.

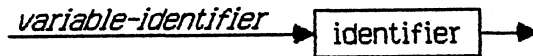
Examples of variable-declarations:

```
x, y, z: real;
i, j: integer;
k: 0..9;
p, q, r: boolean;
operator: (plus, minus, times);
a: array[0..63] of real;
c: color;
f: file of char;
hue1, hue2: set of color;
p1, p2: person;
r, r1, r2: array[1..10, 1..10] of real;
coord: polar;
pooltape: array[1..4] of tape;
```

4.2 Variable-References

A variable-reference denotes the value of a variable of simple-type or pointer-type, or the collection of values represented by a variable of structured-type.

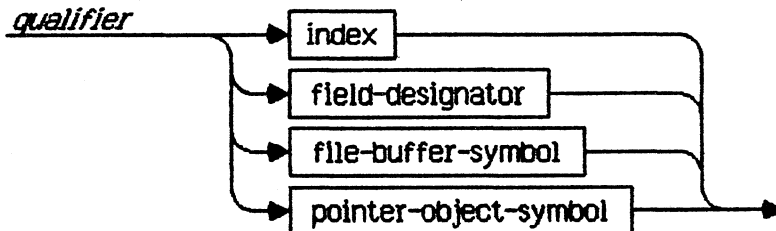




Syntax for the various kinds of qualifiers is given below.

4.3 Qualifiers

As shown above, a variable-reference is a variable-identifier followed by zero or more qualifiers. Each qualifier modifies the meaning of the variable-reference.



An array identifier with no qualifier is a reference to the entire array:

xResults

If the array identifier is followed by an index, this denotes a specific component of the array:

xResults[current+1]

If the array component is a record, the index may be followed by a field-designator; in this case the variable-reference denotes a specific field within a specific array component.

xResults[current+1].link

If the field is a pointer, the field-designator may be followed by the pointer-object-symbol, to denote the object pointed to by the pointer:

xResults[current+1].link^

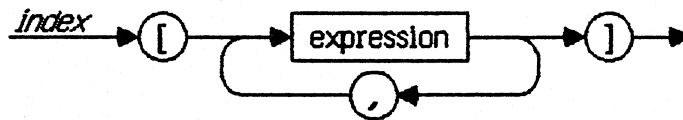
If the object of the pointer is an array, another index can be added to denote a component of this array (and so forth):

xResults[current+1].link^[1]

4.3.1 Arrays, Strings, and Indexes

A specific component of an array variable is denoted by a variable-reference that refers to the array variable, followed by an index that specifies the component.

A specific character within a string variable is denoted by a variable-reference that refers to the string variable, followed by an index that specifies the character position.



Examples of Indexed arrays:

```
m[i, j]
a[i+j]
```

Each expression in the index selects a component in the corresponding dimension of the array. The number of expressions must not exceed the number of index-types in the array declaration, and the type of each expression must be assignment-compatible with the corresponding index-type.

In indexing a multi-dimensional array, you can use either multiple indexes or multiple expressions within an index. The two forms are completely equivalent. For example,

```
m[1][j]
```

is equivalent to

```
m[i, j]
```

For array variables, each index expression must be *assignment-compatible* with the corresponding index-type specified in the declaration of the array-type.

A string value can be indexed by only one index expression, whose value must be in the range $1..n$, where n is the current length of the string value. The effect is to access one character of the string value.

Warning: When a string value is manipulated by assigning values to individual character positions, the dynamic length of the string is not maintained. For example, suppose that `strval` is declared as follows:

```
strval: string[10];
```

The memory space allocated for `strval` includes space for 10 `char` values and a number that will represent the current length of the string — i.e., the number

of char values currently in the string. Initially, all of this space contains unspecified values. The assignment

```
strval[1]:= 'F'
```

may or may not work, depending on what the unspecified length happens to be. If this assignment works, it stores the char value 'F' in character position 1, but the length of `strval` remains unspecified. In other words, the value of `strval[1]` is now 'F', but the value of `strval` is unspecified. Therefore, the effect of a statement such as `writeln(strval)` is unspecified.

Therefore, this kind of string manipulation is not recommended. Instead, use the standard procedures described in Section 11.6. These procedures properly maintain the lengths of the string values they modify.

4.3.2 Records and Field-Designators

A specific field of a record variable is denoted by a variable-reference that refers to the record variable, followed by a field-designator that specifies the field.



Examples of field-designators:

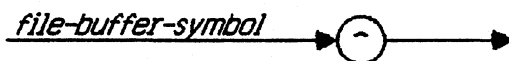
```
p2^.pregnant
coord.theta
```

4.3.3 File-Buffers

Although a file variable may have any number of components, only one component is accessible at any time. The position of the current component in the file is called the current file position. See Sections 10.2 and 10.3 for standard procedures that move the current file position. Program access to the current component is via a special variable associated with the file, called a file-buffer.

The file-buffer is implicitly declared when the file variable is declared. If `F` is a file variable with components of type `T`, the associated file-buffer is a variable of type `T`.

The file-buffer associated with a file variable is denoted by a variable-reference that refers to the file variable, followed by a qualifier called the file-buffer-symbol.



Thus the file-buffer of file `F` is referenced by `F^`.

Sections 10.2 and 10.3 describe standard procedures that are used to move the current file position within the file and to transfer data between the file-buffer and the current file component.

4.3.4 Pointers and Their Objects

The value of a pointer variable is either `nil`, or a value that identifies some other variable, called the object of the pointer.

The object pointed to by a pointer variable is denoted by a variable-reference that refers to the pointer variable, followed by a qualifier called the pointer-object-symbol.



NOTE

Pointer values are created by the standard procedure `new` (see Section 11.2.1), by the `@` operator (see Section 5.1.6), and by the standard procedure `pointer` (see Section 11.3.4).

The constant `nil` (see Section 3.3) does not point to a variable. If you access memory via a `nil` pointer reference, the results are unspecified; there may not be any error indication.

Examples of references to objects of pointers:

```
p1~
p1~ .sibling~
```


Chapter 5

EXPRESSIONS

5.1	Operators	5-6
5.1.1	Binary Operators: Order of Evaluation of Operands	5-6
5.1.2	Arithmetic Operators	5-7
5.1.3	Boolean Operators	5-9
5.1.4	Set Operators	5-9
5.1.4.1	Result Type in Set Operations	5-10
5.1.5	Relational Operators	5-10
5.1.5.1	Comparing Numbers	5-10
5.1.5.2	Comparing Booleans	5-11
5.1.5.3	Comparing Strings	5-11
5.1.5.4	Comparing Sets	5-11
5.1.5.5	Testing Set Membership	5-11
5.1.5.6	Comparing Packed Arrays of Char	5-11
5.1.6	@-Operator	5-11
5.1.6.1	@-Operator with a Variable	5-12
5.1.6.2	@-Operator with a Value Parameter	5-12
5.1.6.3	@-Operator with a Variable Parameter	5-12
5.1.6.4	@-Operator with a Procedure or Function	5-12
5.2	Function-Calls	5-13
5.3	Set-Constructors	5-14

EXPRESSIONS

Expressions consist of operators and operands, i.e. variables, constants, set-constructors, and function calls. Table 5-1 shows the operator precedence:

Table 5-1
Precedence of Operators

<i>Operators</i>	<i>Precedence</i>	<i>Categories</i>
@, not	highest	unary operators
*, /, div, mod, and	second	"multiplying" operators
+, -, or	third	"adding" operators & signs
=, <>, <, >, <=, >=, in	lowest	relational operators

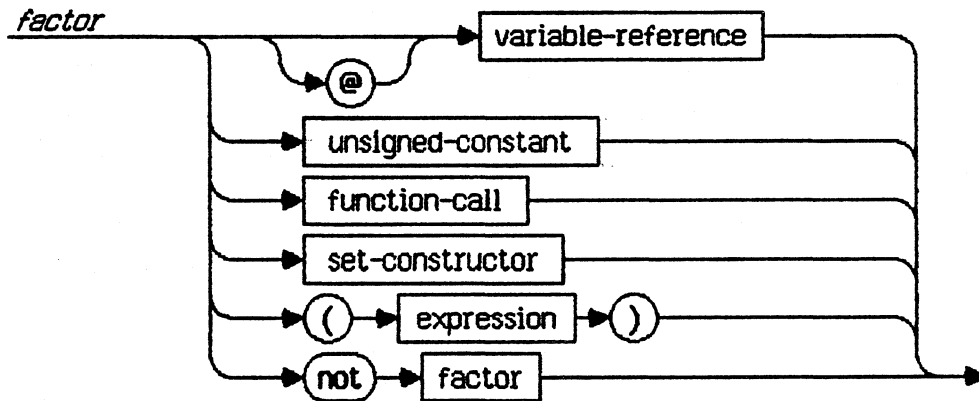
The following rules specify the way in which operands are bound to operators:

- When an operand is written between two operators of different precedence, it is bound to the operator with the higher precedence.
- When an operand is written between two operators of the same precedence, it is bound to the operator on the left.

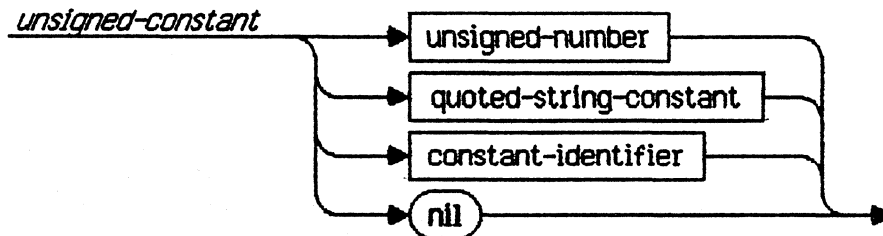
Note that the order in which operations are performed is not specified.

These rules are implicit in the syntax for expressions, which are built up from factors, terms, and simple-expressions.

The syntax for a factor allows the unary operators @ and not to be applied to a value:



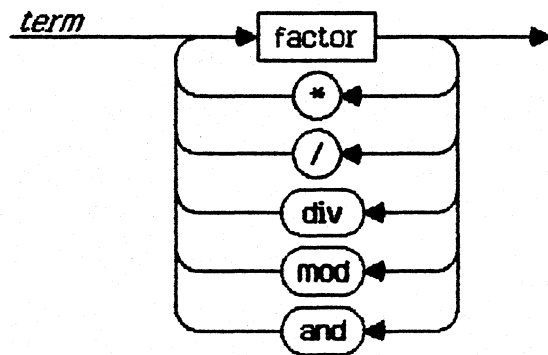
A function-call activates a function, and denotes the value returned by the function (see Section 5.2). A set-constructor denotes a value of a set-type (see Section 5.3). An unsigned-constant has the following syntax:



Examples of factors:

x	{variable-reference}
@x	{pointer to a variable}
15	{unsigned-constant}
(x+y+z)	{sub-expression}
sin(x/2)	{function-call}
['A'..'F', 'a'..'f']	{set-constructor}
not p	{negation of a boolean}

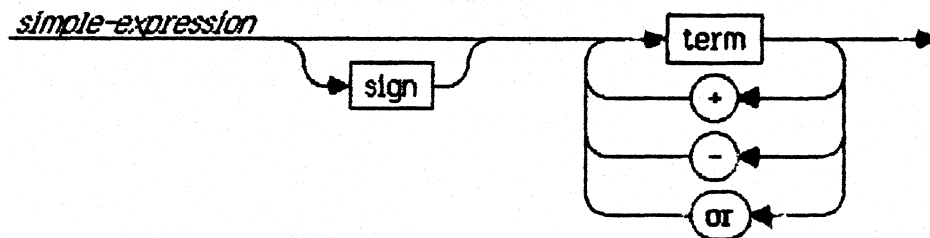
The syntax for a term allows the "multiplying" operators to be applied to factors:



Examples of terms:

x*y
 i/(1-i)
 p and q
 (x <= y) and (y < z)

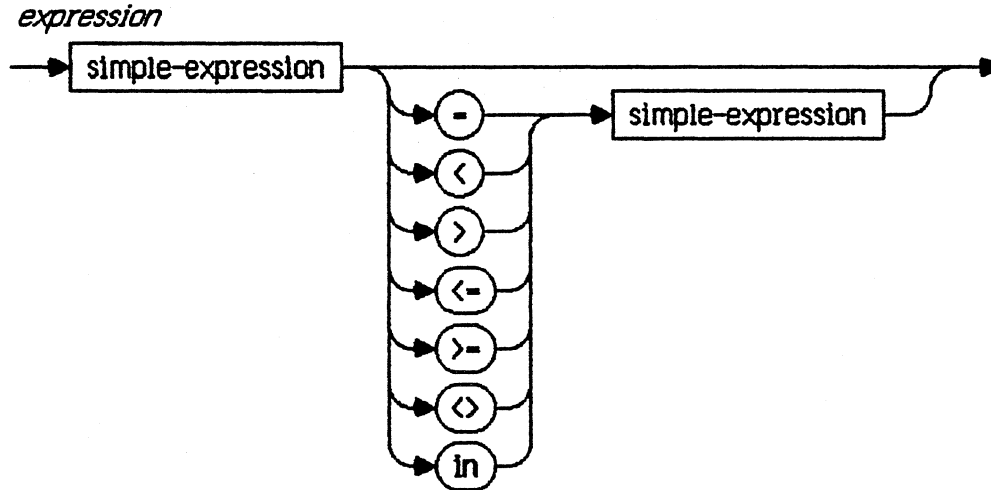
The syntax for a simple-expression allows the "adding" operators and signs to be applied to terms:



Examples of simple-expressions:

x+y
 -x
 hue1 + hue2
 i*j + 1

The syntax for an expression allows the relational operators to be applied to simple-expressions:



Examples of expressions:

```
x = 1.5
p <= q
p = q and r
(i < j) = (j < k)
c in hue1
```

5.1 Operators

5.1.1 Binary Operators: Order of Evaluation of Operands

The order of evaluation of the operands of a binary operator is unspecified.

5.1.2 Arithmetic Operators

The types of operands and results for arithmetic binary and unary operations are shown in Tables 5-2 and 5-3 respectively.

Table 5-2
Binary Arithmetic Operations

<i>Operator</i>	<i>Operation</i>	<i>Operand Types</i>	<i>Type of Result</i>
+	addition	integer, real, or longint	integer, real, or longint
-	subtraction		
*	multiplication		
/	division	integer, real, or longint	real
div	division with integer result	integer or longint	integer or longint
mod	modulo	integer or longint	integer
<i>Note:</i> The symbols +, -, and * are also used as set operators (see Section 5.1.4).			

Table 5-3
Unary Arithmetic Operations (Signs)

<i>Operator</i>	<i>Operation</i>	<i>Operand Types</i>	<i>Type of Result</i>
+	identity	integer, real, or longint	same as operand
-	sign-negation		

Any operand whose type is *subr*, where *subr* is a subrange of some ordinal-type *ordtyp*, is treated as if it were of type *ordtyp*. Consequently an expression that consists of a single operand of type *subr* is itself of type *ordtyp*.

If both the operands of the addition, subtraction, or multiplication operators are of type *integer* or *longint*, the result is of type *integer* or *longint* as described in Section 3.1.1.2; otherwise, the result is of type *real*.

NOTE

See Appendix D for more information on all arithmetic operations with operands or results of type `real`.

The result of the identity or sign-negation operator is of the same type as the operand.

The value of `i div j` is the mathematical quotient of i/j , rounded toward zero to an `integer` or `longint` value. An error occurs if $j=0$.

The value of `i mod j` is equal to the value of

$$i - (i \text{ div } j) * j$$

The sign of the result of `mod` is always the same as the sign of `i`. An error occurs if $j=0$.

The predefined constant `maxint` is of type `integer`. Its value is 32767. This value satisfies the following conditions:

- All whole numbers in the closed interval from $-\text{maxint}-1$ to $+\text{maxint}$ are representable in the type `integer`.
- Any unary operation performed on a whole number in this interval will be correctly performed according to the mathematical rules for whole-number arithmetic.
- Any binary integer operation on two whole numbers in this same interval will be correctly performed according to the mathematical rules for whole-number arithmetic, provided that the result is also in this interval. If the mathematical result is not in this interval, then the actual result is the low-order 16 bits of the mathematical result.
- Any relational operation on two whole numbers in this same interval will be correctly performed according to the mathematical rules for whole-number arithmetic.

5.1.3 Boolean Operators

The types of operands and results for Boolean operations are shown in Table 5-4.

Table 5-4
Boolean Operations

<i>Operator</i>	<i>Operation</i>	<i>Operand Types</i>	<i>Type of Result</i>
or	disjunction	boolean	boolean
and	conjunction		
not	negation		

Whether a Boolean expression is completely or partially evaluated if its value can be determined by partial evaluation is unspecified. For example, consider the expression

true or boolTst(x)

where **boolTst** is a function that returns a **boolean** value. This expression will always have the value **true**, regardless of the result of **boolTst(x)**. The language definition does not specify whether the **boolTst** function is called when this expression is evaluated. This could be important if **boolTst** has side-effects.

5.1.4 Set Operators

The types of operands and results for set operations are shown in Table 5-5.

Table 5-5
Set Operations

<i>Operator</i>	<i>Operation</i>	<i>Operand Types</i>	<i>Type of Result</i>
+	union	compatible set-types	(see 5.1.4.1)
-	difference		
*	intersection		

5.1.4.1 Result Type in Set Operations

The following rules govern the type of the result of a set operation where one (or both) of the operands is a set of *subr*, where *ordtyp* represents any ordinal-type and *subr* represents a subrange of *ordtyp*:

- If *ordtyp* is not the type *integer*, then the type of the result is *set of ordtyp*.
- If *ordtyp* is the type *integer*, then the type of the result is *set of 0..4087* in the current implementation (0..32767 in a future implementation). This rule results from the limitations on set-types (see Section 3.2.3).

5.1.5 Relational Operators

The types of operands and results for relational operations are shown in Table 5-6, and discussed further below.

Table 5-6
Relational Operations

<i>Operator</i>	<i>Operation</i>	<i>Operand Types</i>	<i>Type of Result</i>
-	equal	compatible set-, simple-, or pointer-types (& see below)	boolean
<>	not equal		
<	less	compatible simple-types (& see below)	
>	greater		
<=	less/equal		
>=	greater/equal		
<=	subset of	compatible	
>=	superset of	set-types	
in	member of	<i>left operand:</i> any ordinal-type T <i>right operand:</i> set of T	

5.1.5.1 Comparing Numbers

When the operands of <, >, >=, or <= are numeric, they need not be of compatible type *if* one operand is real and the other is *integer* or *longint*.

NOTE

See Appendix D for more information on relational operations with operands of type *real*.

5.1.5.2 Comparing Booleans

If *p* and *q* are boolean operands, then *p*=*q* denotes their equivalence and *p*<=*q* denotes the implication of *q* by *p* (because *false*<*true*). Similarly, *p*<>*q* denotes logical "exclusive-or."

5.1.5.3 Comparing Strings

When the relational operators =, <>, <, >, <=, and > are used to compare strings (see Section 3.1.1.6), they denote lexicographic ordering according to the ordering of the ASCII character set. Note that any two string values can be compared since all string values are compatible.

5.1.5.4 Comparing Sets

If *u* and *v* are set operands, then *u*<=*v* denotes the inclusion of *u* in *v*, and *u*>=*v* denotes the inclusion of *v* in *u*.

5.1.5.5 Testing Set Membership

The *in* operator yields the value *true* if the value of the ordinal-type operand is a member of the set-type operand; otherwise it yields the value *false*.

5.1.5.6 Comparing Packed Arrays of Char

In addition to the operand types shown in the table, the = and <> operators can also be used to compare a packed array[1..N] of *char* with a string *constant* containing exactly N characters, or to compare two one-dimensional packed arrays of *char* of *identical* type.

5.1.6 @-Operator

A pointer to a variable can be computed with the @-operator. The operand and result types are shown in Table 5-7.

Table 5-7
Pointer Operation

<i>Operator</i>	<i>Operation</i>	<i>Operand</i>	<i>Type of Result</i>
@	pointer formation	variable, parameter, procedure, or function	same as nil

@ is a unary operator taking a single variable, parameter, procedure, or function as its operand and computing the value of its pointer. The type of the

value is equivalent to the type of nil, and consequently can be assigned to any pointer variable.

5.1.6.1 @-Operator With a Variable

For an ordinary variable (not a parameter), the use of @ is straightforward. For example, if we have the declarations

```
type twochar = packed array[0..1] of char;
var int: integer;
    twocharptr: ^twochar;
```

then the statement

```
twocharptr := @int
```

causes twocharptr to point to int. Now twocharptr[^] is a reinterpretation of the bit value of int as though it were a packed array[0..1] of char.

The operand of @ cannot be a component of a packed variable.

5.1.6.2 @-Operator With a Value Parameter

When @ is applied to a formal value parameter, the result is a pointer to the stack location containing the actual value. Suppose that foo is a formal value parameter in a procedure and fooPtr is a pointer variable. If the procedure executes the statement

```
fooPtr := @foo
```

then fooPtr[^] is a reference to the value of foo. Note that if the actual-parameter is a variable-reference, fooPtr[^] is not a reference to the variable itself; it is a reference to the value taken from the variable and stored on the stack.

5.1.6.3 @-Operator With a Variable Parameter

When @ is applied to a formal variable parameter, the result is a pointer to the actual-parameter (the pointer is taken from the stack). Suppose that fum is a formal variable parameter of a procedure, fie is a variable passed to the procedure as the actual-parameter for fum, and fumPtr is a pointer variable.

If the procedure executes the statement

```
fumPtr := @fum
```

then fumPtr is a pointer to fie. fumPtr[^] is a reference to fie itself.

5.1.6.4 @-Operator With a Procedure or Function

It is possible to apply @ to a procedure or a function, yielding a pointer to the entry-point. Note that Pascal provides no mechanism for using such a pointer. Currently the only use for a procedure pointer is to pass it to an assembly-language routine, which can then JSR to that address.

If the procedure pointed to is in the local segment, @ returns the current address of the procedure's entry point. If the procedure is in some other

segment, however, @ returns the address of the jump table entry for the procedure.

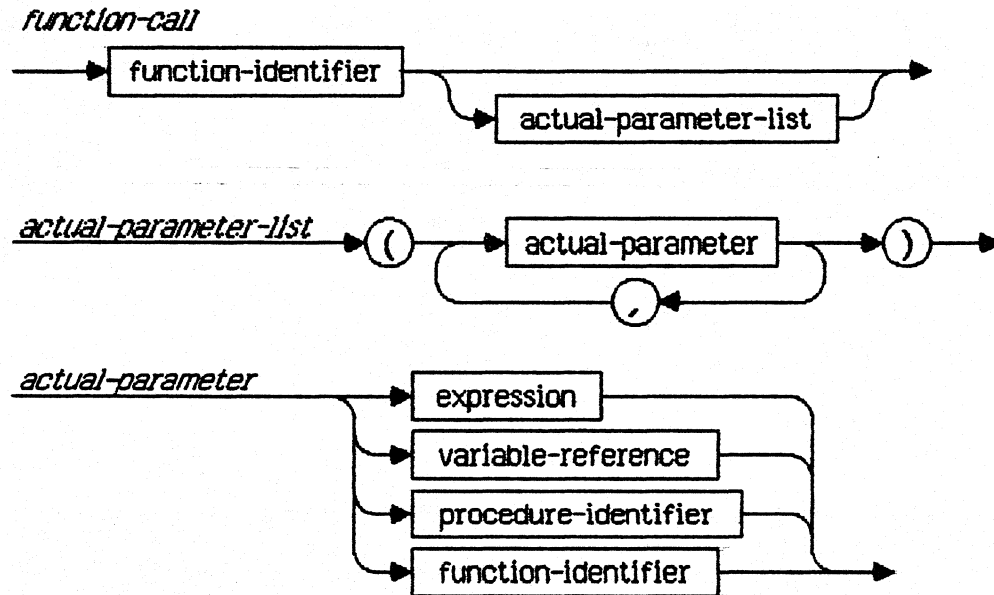
In logical memory mapping (see *Workshop Reference Manual for the Lisa*), the procedure pointer is always valid.

In physical memory mapping, code swapping may change a local-segment procedure address without warning, and the procedure pointer can become invalid. If the procedure is not in the local segment, the jump-table entry address will remain valid despite swapping because the jump table is not moved.

5.2 Function-Calls

A function-call specifies the activation of the function denoted by the function-identifier. If the corresponding function-declaration contains a list of formal-parameters, then the function-call must contain a corresponding list of actual-parameters. Each actual-parameter is substituted for the corresponding formal-parameter. The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively. The number of actual-parameters must be equal to the number of formal parameters.

The order of evaluation and binding of the actual-parameters is unspecified.



A function-identifier is any identifier that has been declared to denote a function.

Examples of function-calls:

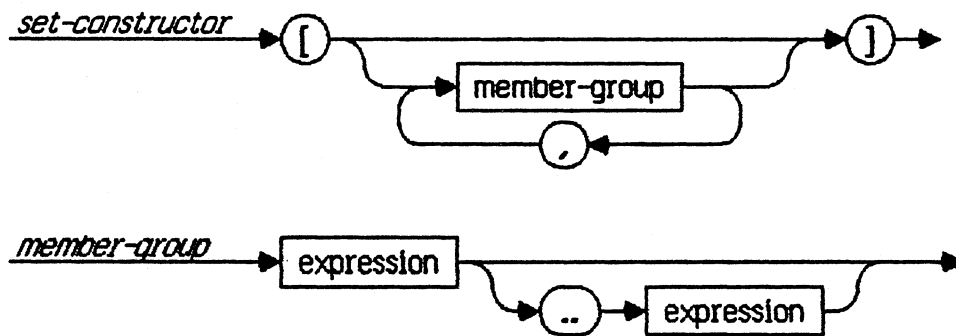
```

sum(a, 63)
gcd(147, k)
sin(x+y)
eof(f)
ord(f^~)

```

5.3 Set-Constructors

A set-constructor denotes a value of a set-type, and is formed by writing expressions within [brackets]. Each expression denotes a value of the set.



The notation [] denotes the empty set, which belongs to every set-type. Any member-group $x..y$ denotes as set members the range of all values of the base-type in the closed interval x to y .

If x is greater than y , then $x..y$ denotes no members and $[x..y]$ denotes the empty set.

All values designated in member-groups in a particular set-constructor must be of the same ordinal-type. This ordinal-type is the base-type of the resulting set. If an integer value designated as a set member is outside the limits given in Section 3.2.3 (0..4087 in the current implementation), the results are unspecified.

Examples of set-constructors:

```

[red, c, green]
[1, 5, 10..k mod 12, 23]
['A'..'Z', 'a'..'z', chr(xcode)]

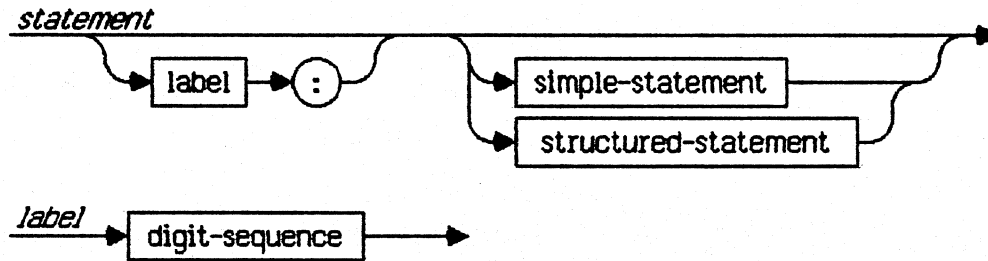
```

Chapter 6 STATEMENTS

6.1	Simple Statements	6-3
6.1.1	Assignment-Statements	6-3
6.1.2	Procedure-Statements	6-4
6.1.3	Goto-Statements	6-5
6.2	Structured-Statements	6-6
6.2.1	Compound-Statements	6-6
6.2.2	Conditional-Statements	6-7
6.2.2.1	If-Statements	6-7
6.2.2.2	Case-Statements	6-7
6.2.3	Repetitive-Statements	6-9
6.2.3.1	Repeat-Statements	6-9
6.2.3.2	While-Statements	6-10
6.2.3.3	For-Statements	6-11
6.2.4	With-Statements	6-13

STATEMENTS

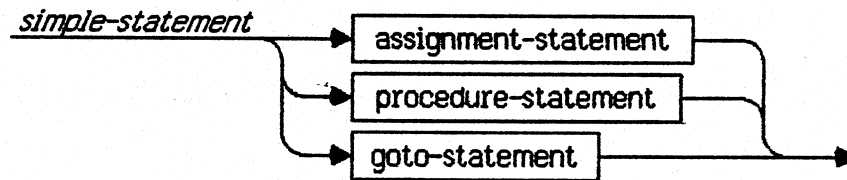
Statements denote algorithmic actions, and are executable. They can be prefixed by labels; a labeled statement can be referenced by a goto-statement.



A digit-sequence used as a label must be in the range 0..9999, and must first be declared as described in Section 2.1.

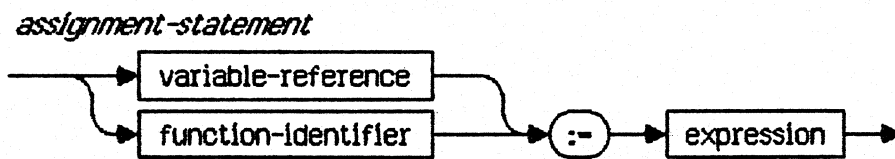
6.1 Simple Statements

A simple-statement is a statement that does not contain any other statement.



6.1.1 Assignment-Statements

The syntax for an assignment-statement is as follows:



The assignment-statement can be used in two ways:

- To replace the current value of a variable by a new value specified as an expression

- To specify an expression whose value is to be returned by a function.

The expression must be assignment-compatible with the type of the variable or the result-type of the function.

NOTE

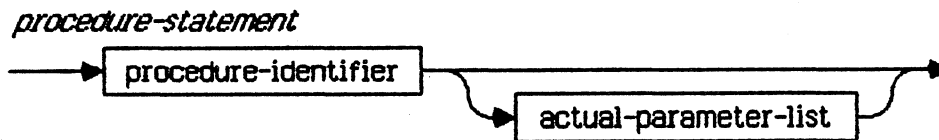
If the selection of the variable involves indexing an array or taking the object of a pointer, it is not specified whether these actions precede or follow the evaluation of the expression.

Examples of assignment-statements:

```
x := y+z;
p := (i<=1) and (i<100);
i := sqr(k) - (i*j);
hue1 := [blue, succ(c)];
```

6.1.2 Procedure-Statements

A procedure-statement serves to execute the procedure denoted by the procedure-identifier.



(A procedure-identifier is simply an identifier that has been used to declare a procedure.)

If the procedure has formal-parameters (see Section 7.3), the procedure-statement must contain a list of actual-parameters that are bound to the corresponding formal-parameters. The number of actual-parameters must be equal to the number of formal parameters. The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively.

The rules for an actual-parameter *AP* depend on the corresponding formal-parameter *FP*:

- If *FP* is a value parameter, *AP* must be an expression. The type of the value of *AP* must be assignment-compatible with the type of *FP*.
- If *FP* is a variable parameter, *AP* must be a variable-reference. The type of *AP* must be identical to the type of *FP*.

- If FP is a procedural parameter, AP must be a procedure-identifier. The type of each formal-parameter of AP must be identical to the type of the corresponding formal-parameter of FP.
- If FP is a functional parameter, AP must be a function-identifier. The type of each formal-parameter of AP must be identical to the type of the corresponding formal-parameter of FP, and the result-type of AP must be identical to the result-type of FP.

NOTE

The order of evaluation and binding of the actual parameters is unspecified.

Examples of procedure-statements:

```
printhead;
transpose(a, n, m);
bisect(fct, -1.0, +1.0, x);
```

6.1.3 Goto-Statements

A goto-statement causes a jump to another statement in the program, namely the statement prefixed by the label that is referenced in the goto-statement.



NOTE

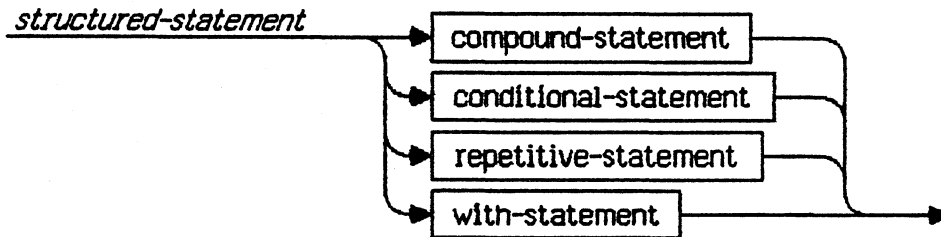
The constants that introduce cases within a case-statement (see Section 6.2.2.2) are not labels, and cannot be referenced in goto-statements.

The following restrictions apply to goto-statements:

- The effect of a jump into a structured statement from outside of the structured statement is unspecified.
- The effect of a jump between the then part and the else part of an if-statement is unspecified.
- The effect of a jump between two different cases within a case-statement is unspecified.

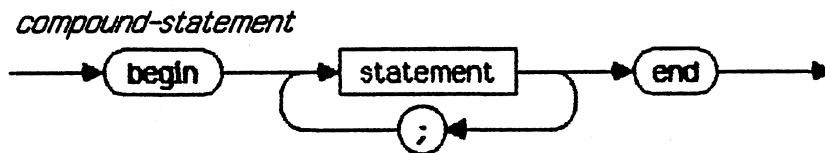
6.2 Structured-Statements

Structured-statements are constructs composed of other statements that must be executed either conditionally (conditional-statements), repeatedly (repetitive-statements), or in sequence (compound-statement or with-statement).



6.2.1 Compound-Statements

The compound-statement specifies that its component statements are to be executed in the same sequence as they are written.



Example of compound-statement:

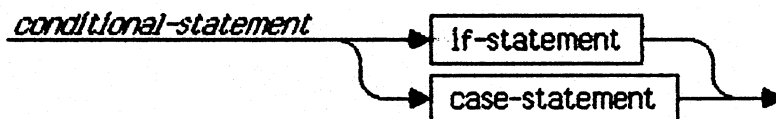
```

begin
  z := x;
  x := y;
  y := z
end
  
```

An important use of the compound-statement is to group more than one statement into a single statement, in contexts where Pascal syntax only allows one statement. The symbols **begin** and **end** act as "statement brackets." Examples of this will be seen in Section 6.2.3.2.

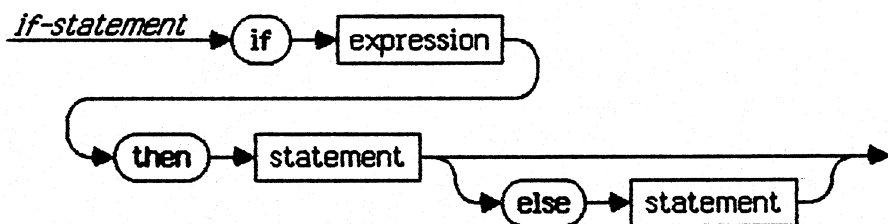
6.2.2 Conditional-Statements

A conditional-statement selects for execution a single one (or none) of its component statements.



6.2.2.1 If-Statements

The syntax for if-statements is as follows:



The expression must yield a result of type **boolean**. If the expression yields the value **true**, the statement following the **then** is executed.

If the expression yields **false** and the **else** part is present, the statement following the **else** is executed; if the **else** part is not present, nothing is executed.

The syntactic ambiguity arising from the construct:

```

if e1 then
  if e2 then s1
  else s2
  
```

is resolved by interpreting the construct as being equivalent to:

```

if e1 then begin
  if e2 then s1
  else s2
end
  
```

Examples of if-statements:

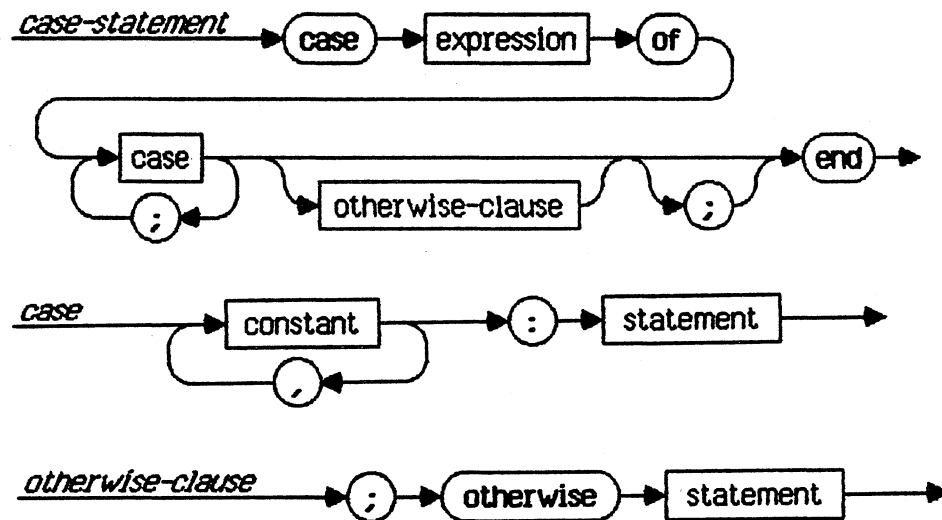
```

if x < 1.5 then z := x*y else z := 1.5;
if p1 <> nil then p1 := p1^.father;
  
```

6.2.2.2 Case-Statements

The case-statement contains an expression (the selector) and a list of statements. Each statement must be prefixed with one or more constants

(called case-constants), or with the reserved word **otherwise**. All the case-constants must be distinct and must be of an ordinal-type that is compatible with the type of the selector.



The case-statement specifies execution of the statement prefixed by a case-constant equal to the current value of the selector. If no such case-constant exists and an **otherwise** part is present, the statement following the word **otherwise** is executed; if no **otherwise** part is present, nothing is executed.

Examples of case-statements:

```

case operator of
  plus:  x := x+y;
  minus: x := x-y;
  times: x := x*y
end
case 1 of
  1:      x := sin(x);
  2:      x := cos(x);
  3,4,5:  x := exp(x);
  otherwise x := ln(x)
end

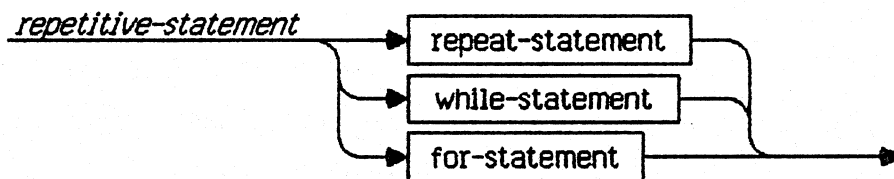
```

IMPLEMENTATION NOTE

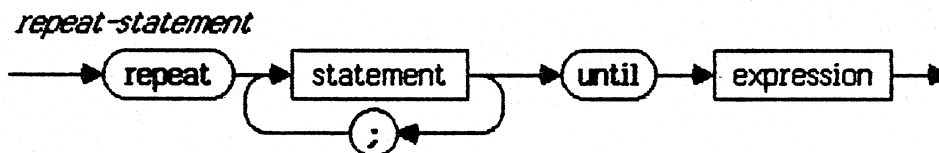
In the current implementation, the case-statement will not work correctly if any case-constant is of type `longint` or the value of the selector is of type `longint`.

6.2.3 Repetitive-Statements

Repetitive-statements specify that certain statements are to be executed repeatedly.

**6.2.3.1 Repeat-Statements**

A repeat-statement contains an expression which controls the repeated execution of a sequence of statements contained within the repeat-statement.



The expression must yield a result of type `boolean`. The statements between the symbols `repeat` and `until` are repeatedly executed until the expression yields the value `true` on completion of the sequence of statements. The sequence of statements is executed at least once, because the expression is evaluated *after* execution of the sequence.

Examples of repeat-statements:

```

repeat
  k := i mod j;
  i := j;
  j := k
until j = 0
  
```

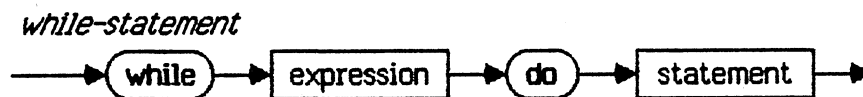
```

repeat
  process(f^);
  get(f)
until eof(f)

```

6.2.3.2 while-Statements

A while-statement contains an expression which controls the repeated execution of one statement (possibly a compound-statement) contained within the while-statement.



The expression must yield a result of type **boolean**. It is evaluated *before* the contained statement is executed. The contained statement is repeatedly executed as long as the expression yields the value **true**. If the expression yields **false** at the beginning, the statement is not executed.

The while-statement:

```

while b do body

```

is equivalent to:

```

if b then repeat
  body
until not b

```

Examples of while-statements:

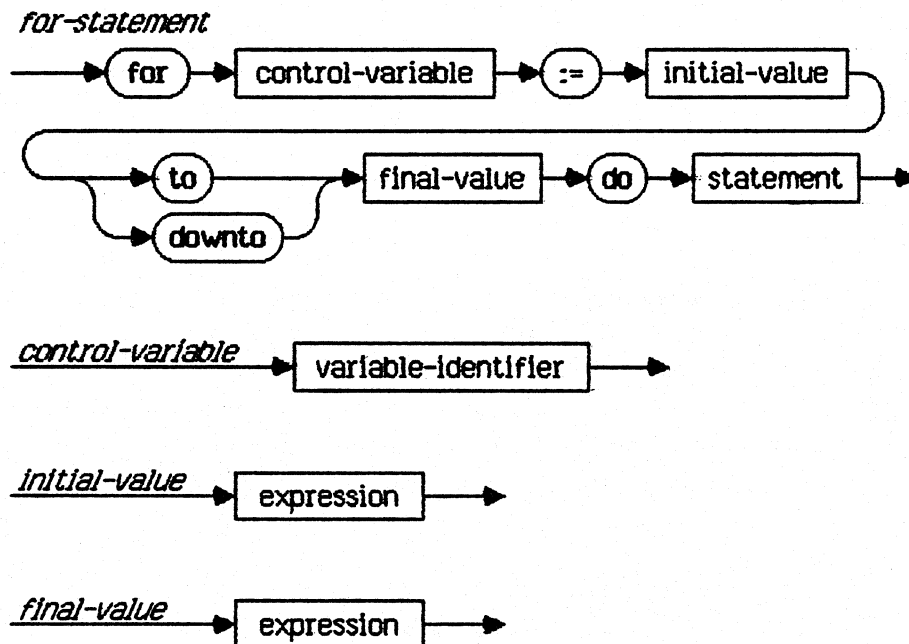
```

while a[i] <> x do i := i+1
while i>0 do begin
  if odd(i) then z := z*x;
  i := i div 2;
  x := sqr(x)
end
while not eof(f) do begin
  process(f^);
  get(f)
end

```

6.2.3.3 For-Statements

The for-statement causes one contained statement (possibly a compound-statement) to be repeatedly executed while a progression of values is assigned to a variable called the control-variable.



The control-variable must be a variable-identifier (without any qualifier). It must be local to the innermost block containing the for-statement, and must not be a variable parameter of that block. The control-variable must be of ordinal-type, and the initial and final values must be of a type compatible with this type.

The first value assigned to the control-variable is the initial-value.

If the for-statement is constructed with the reserved word **to**, each successive value of the control-variable is the successor (see Section 3.1) of the previous value, using the inherent ordering of values according to the type of the control-variable. When each value is assigned to the control-variable, it is compared to the final-value; if it is less than or equal to the final value, the contained statement is then executed.

If the for-statement is constructed with the reserved word **downto**, each successive value of the control-variable is the predecessor (see Section 3.1) of the previous value. When each value is assigned to the control-variable, it is

compared to the final-value; if it is greater than or equal to the final value, the contained statement is then executed.

If the value of the control-variable is altered by execution of the repeated statement, the effect is unspecified. After a for-statement is executed, the value of the control-variable is unspecified, unless the for-statement was exited by a goto. Apart from these restrictions, the for-statement:

```
for v := e1 to e2 do body
```

is equivalent to:

```
begin
  temp1 := e1;
  temp2 := e2;
  if temp1 <= temp2 then begin
    v := temp1;
    body;
    while v <> temp2 do begin
      v := succ(v);
      body
    end
  end
end
```

and the for-statement:

```
for v := e1 downto e2 do body
```

is equivalent to:

```
begin
  temp1 := e1;
  temp2 := e2;
  if temp1 >= temp2 then begin
    v := temp1;
    body;
    while v <> temp2 do begin
      v := pred(v);
      body
    end
  end
end
```

where `temp1` and `temp2` are auxiliary variables of the host type of the variable `v` that do not occur elsewhere in the program.

Examples of for-statements:

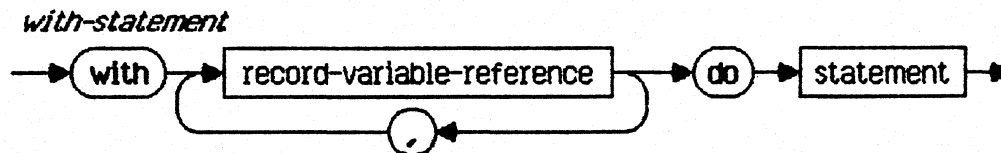
```

for i := 2 to 63 do if a[i] > max then max := a[i]
for i := 1 to n do for j := 1 to n do
begin
  x := 0;
  for k := 1 to n do x := x + m1[i,k]*m2[k,j];
  m[i,j] := x
end
for c := red to blue do q(c)

```

6.2.4 With-Statements

The syntax for a with-statement is



(A record-variable reference is simply a reference to some record variable.) The occurrence of a record-variable-reference in a with-statement affects the way the compiler processes variable-references within the statement following the word **do**. Fields of the record-variable can be referenced by their field-identifiers, without explicit reference to the record-variable.

Example of with-statement:

```

with date do if month = 12 then begin
  month := 1;
  year := year + 1
end
else month := month + 1

```

This is equivalent to:

```

if date.month = 12 then begin
  date.month := 1;
  date.year := date.year + 1
end
else date.month := date.month + 1

```

Within a with-statement, each variable-reference is checked to see if it can be interpreted as a field of the record. Suppose that we have the following declarations:

```

type recTyp = record
    foo: integer;
    bar: real
end;

```

```

var baz: recTyp;
    foo: integer;

```

The identifier `foo` can refer both to a field of the record variable `baz` and to a variable of type `integer`. Now consider the statement

```

with baz do begin
    ...
    foo := 36; {which foo is this?}
    ...
end

```

The `foo` in this with-statement is a reference to the field `baz.foo`, not the variable `foo`.

The statement:

```

with v1, v2, ... vn do s

```

is equivalent to the following "nested" with-statements:

```

with v1 do
  with v2 do
    ...
    with vn do s

```

If `vn` in the above statements is a field of both `v1` and `v2`, it is interpreted to mean `v2.vn`, not `v1.vn`. The list of record-variable-references in the with-statement is checked from right to left.

If the selection of a variable in the record-variable-list involves the indexing of an array or the de-referencing of a pointer, these actions are executed before the component statement is executed.

Warning: If a variable in the record-variable-list is a pointer-reference, the value of the pointer must not be altered within the with-statement. If the value of the pointer is altered, the results are unspecified.

Example of unsafe with-statement using pointer-reference:

```
with ppp^ do begin
  ...
  new(ppp); {Don't do this ...}
  ...
  ppp:=xxx; {... or this}
  ...
end
```

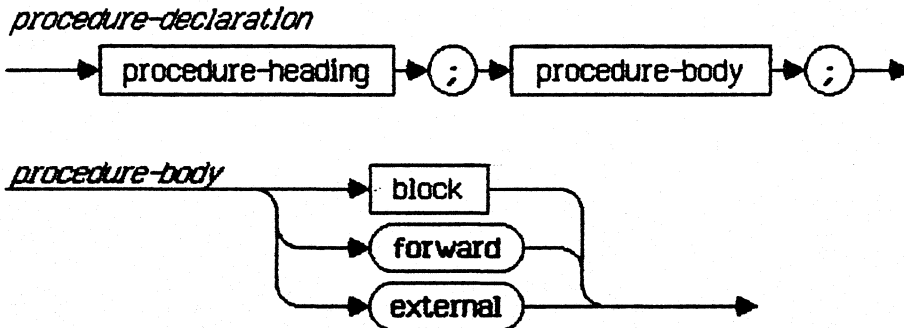

Chapter 7 PROCEDURES AND FUNCTIONS

7.1	Procedure-Declarations	7-3
7.2	Function-Declarations	7-6
7.3	Parameters	7-7
7.3.1	Value Parameters	7-9
7.3.2	Variable Parameters	7-9
7.3.3	Procedural Parameters	7-9
7.3.4	Functional Parameters.....	7-11
7.3.5	Parameter List Compatibility	7-11

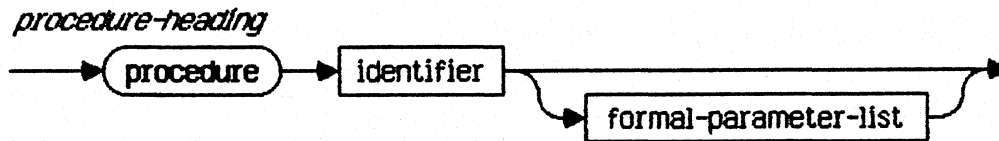
PROCEDURES AND FUNCTIONS

7.1 Procedure-Declarations

A procedure-declaration associates an identifier with part of a program so that it can be activated by a procedure-statement.



The procedure-heading specifies the identifier for the procedure, and the formal parameters (if any).



The syntax for a formal-parameter-list is given in Section 7.3.

A procedure is activated by a procedure-statement (see Section 6.1.2), which gives the procedure's identifier and any actual-parameters required by the procedure. The statements to be executed upon activation of the procedure are specified by the statement-part of the procedure's block. If the procedure's identifier is used in a procedure-statement within the procedure's block, the procedure is executed recursively.

Example of a procedure-declaration:

```

procedure readInteger (var f: text; var x: integer);
  var value, digitValue: integer;
  begin
    while (f^ = ' ') and not eof(f) do get(f);
    value := 0;
    while (f^ in ['0'..'9']) and not eof(f) do begin
      digitValue := ord(f^) - ord('0');
      value := 10*value + digitValue;
      get(f)
    end;
    x := value
  end;

```

A procedure-declaration that has **forward** instead of a block is called a **forward declaration**. Somewhere after the forward declaration (and in the same block), the procedure is actually defined by a **defining declaration** — a procedure-declaration that uses the same procedure-identifier, omits the formal-parameter-list, and includes a block. The forward declaration and the defining declaration must be local to the same block, but need not be contiguous; that is, other procedures or functions can be declared between them and can call the procedure that has been declared forward. This permits mutual recursion.

The forward declaration and the defining declaration constitute a complete declaration of the procedure. The procedure is considered to be declared at the place of the forward declaration.

Example of forward declaration:

```

procedure walter(m, n: integer); {forward declaration}
  forward;

procedure clara(x, y: real);
  begin
    ...
    walter(4, 5); {OK because walter is forward declared}
    ...
  end;

procedure walter; {defining declaration}
  begin
    ...
    clara(8.3, 2.4);
    ...
  end;

```

A procedure-declaration that has **external** instead of a block defines the Pascal interface to a separately assembled or compiled routine (a **.PROC** in the case

of assembly language). The external code must be linked with the compiled Pascal host program before execution; see the *Workshop Reference Manual for the Lisa* for details.

Example of an external procedure-declaration:

```
procedure makescreen(index: integer);
  external;
```

This means that `makescreen` is an external procedure that will be linked to the host program before execution.

IMPLEMENTATION NOTE

It is the programmer's responsibility to ensure that the external procedure is compatible with the `external` declaration in the Pascal program; the current Linker does no checking.

NOTE

This Pascal (unlike Apple II and Apple III Pascal) does not allow a variable parameter of an external procedure or function to be declared without a type. To obtain a similar effect, use a formal-parameter of pointer-type, as in the following example:

```
type bigpaoc = packed array[0..32767] of char;
  bigpaoptr = ^bigpaoc;
...
procedure whatever (bytearray: bigpaoptr);
  external;
```

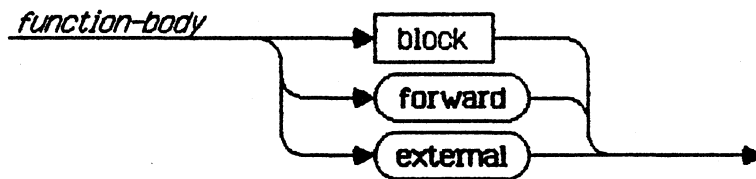
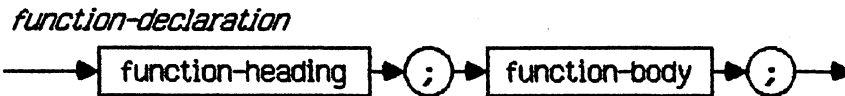
The actual-parameter can be any pointer value obtained via the `@`-operator (see Section 5.1.6). For example, if `dots` is a packed array of `boolean`, it can be passed to `whatever` by writing

```
whatever(@dots)
```

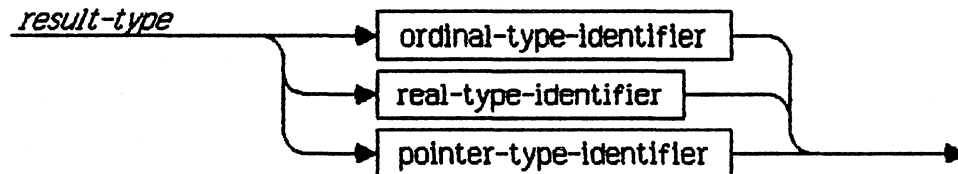
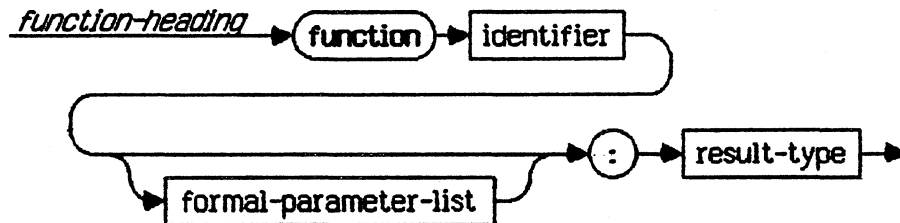
This description of external procedures also applies to external functions.

7.2 Function-Declarations

A function-declaration serves to define a part of the program that computes and returns a value of simple-type or pointer-type.



The function-heading specifies the identifier for the function, the formal parameters (if any), and the type of the function result.



The syntax for a formal-parameter-list is given in Section 7.3.

A function is activated by the evaluation of a function-call (see Section 5.2), which gives the function's identifier and any actual-parameters required by the function. The function-call appears as an operand in an expression. The expression is evaluated by executing the function, and replacing the function-call with the value returned by the function.

The statements to be executed upon activation of the function are specified by the statement-part of the function's block. This block should normally contain at least one assignment-statement (see Section 6.1.1) that assigns a value to the function-identifier. The result of the function is the last value assigned. If

no such assignment-statement exists, or if it exists but is not executed, the value returned by the function is unspecified.

If the function's identifier is used in a function-call within the function's block, the function is executed recursively.

Examples of function-declarations:

```
function max(a: vector; n: integer): real;
  var x: real; i: integer;
  begin
    x := a[1];
    for i := 2 to n do if x < a[i] then x := a[i]
    max := x
  end;

function power(x: real; y: integer): real; { y >= 0 }
  var w, z: real; i: integer;
  begin
    w := x; z := 1; i := y;
    while i > 0 do begin
      {z*(w**1) = x ** y }
      if odd(i) then z := z*w;
      i := i div 2;
      w := sqr(w)
    end;
    {z = x**y }
    power := z
  end;
```

A function can be declared forward in the same manner as a procedure (see Section 7.1 above). This permits mutual recursion.

A function-declaration that has **external** instead of a block defines the Pascal interface to a separately compiled or assembled external routine (a **.FUNC** in the case of assembly language). See the explanation in Section 7.1 above.

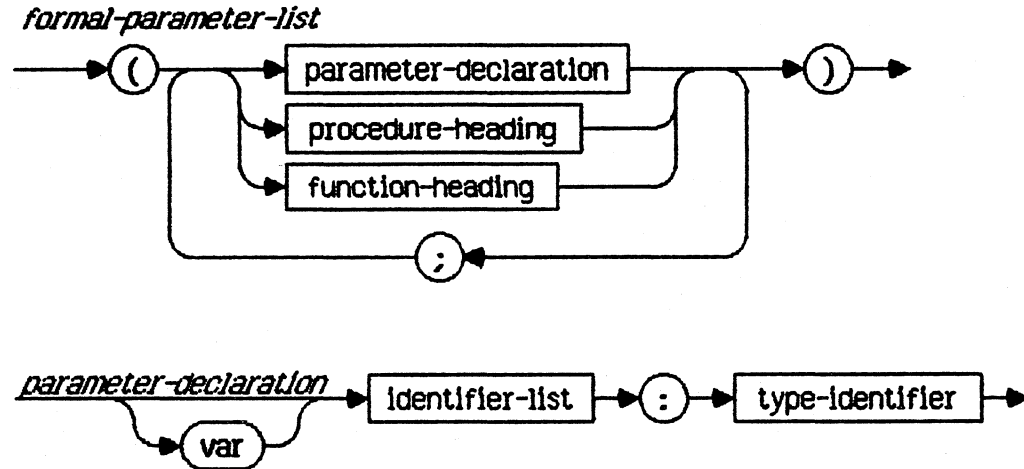
7.3 Parameters

A formal-parameter-list may be part of a procedure-declaration or function-declaration, or it may be part of the declaration of a procedural or functional parameter.

If it is part of a procedure-declaration or function-declaration, it declares the formal parameters of the procedure or function. Each parameter so declared is local to the procedure or function being declared, and can be referenced by its identifier in the block associated with the procedure or function.

If it is part of the declaration of a procedural or functional parameter, it declares the formal parameters of the procedural or functional parameter. In

this case there is no associated block and the identifiers of parameters in the formal-parameter-list are not significant (see Sections 7.3.3 and 7.3.4 below).



There are four kinds of parameters: value parameters, variable parameters, procedural parameters, and functional parameters. They are distinguished as follows:

- A parameter-group preceded by `var` is a list of variable parameters.
- A parameter-group without a preceding `var` is a list of value parameters.
- A procedure-heading or function-heading denotes a procedural or functional parameter; see Sections 7.3.3 and 7.3.4 below.

NOTE

The types of formal-parameters are denoted by type-identifiers. In other words, only a simple identifier can be used to denote a type in a formal-parameter-list. To use a type such as `array[0..255] of char` as the type of a parameter, you must declare a type-identifier for this type:

```
type chararray = array[0..255] of char;
```

The identifier `chararray` can then be used in a formal-parameter-list to denote the type.

NOTE

The word `file` (for an "untyped" file) is not allowed as a type-identifier in a parameter-declaration, since it is a reserved word. To use a parameter of this type, declare some other identifier for the type `file` — for example,

```
type phyle = file;
```

The identifier `phyle` can then be used in a formal-parameter-list to denote the type `file`.

7.3.1 Value Parameters

For a value-parameter, the corresponding actual-parameter in a procedure-statement or function-call (see Sections 5.2 and 6.1.2) must be an expression, and its value must not be of file-type or of any structured-type that contains a file-type. The formal value-parameter denotes a variable local to the procedure or function. The current value of the expression is assigned to the formal value-parameter upon activation of the procedure or function. The actual-parameter must be assignment-compatible with the type of the formal value-parameter.

7.3.2 Variable Parameters

For a variable-parameter, the corresponding actual-parameter in a procedure-statement or function-call (see Sections 5.2 and 6.1.2) must be a variable-reference. The formal variable-parameter denotes this actual variable during the entire activation of the procedure or function.

Within the procedure or function, any reference to the formal variable-parameter is a reference to the actual-parameter itself. The type of the actual-parameter must be *identical* to that of the formal variable-parameter.

NOTE

If the reference to an actual variable-parameter involves indexing an array or finding the object of a pointer, these actions are executed before the activation of the procedure or function.

Components of variables of any packed structured type (including string-types) cannot be used as actual variable parameters.

7.3.3 Procedural Parameters

When the formal-parameter is a procedure-heading, the corresponding actual-parameter in a procedure-statement or function-call (see Sections 5.2 and 6.1.2) must be a procedure-identifier. The identifier in the formal procedure-heading represents the actual procedure during execution of the procedure or function receiving the procedural parameter.

Example of procedural parameters:

```

program passProc;
  var i: integer;

  procedure a(procedure x) {x is a formal procedural
                           parameter.}
  begin
    write('About to call x ');
    x {call the procedure passed as parameter}
  end;

  procedure b;
  begin
    write('In procedure b')
  end;

  function c(procedure x): integer;
  begin
    x: {call the procedure passed as parameter}
    c:=2
  end;

begin
  a(b); {call a, passing b as parameter}
  i:= c(b) {call c, passing b as parameter}
end.

```

If the actual procedure and the formal procedure have formal-parameter-lists, the formal-parameter-lists must be compatible (see Section 7.3.5). However, only the identifier of the actual procedure is written as an actual parameter; any formal-parameter-list is omitted.

Example of procedural parameters with their own formal-parameter-lists:

```

program test;
  procedure xAsPar(y: integer);
  begin
    writeln('y=', y)
  end;

  procedure callProc(procedure xAgain(z: integer));
  begin
    xAgain(1)
  end;

begin {body of program}
  callProc(xAsPar)
end.

```

If the procedural parameter, upon activation, accesses any non-local entity (by variable-reference, procedure-statement, function-call, or label), the entity accessed must be one that was accessible to the procedure when the procedure was passed as an actual parameter.

To see what this means, consider a procedure `pp` which is known to another procedure, `firstPasser`. Suppose that the following sequence takes place:

1. `firstPasser` is executing.
2. `firstPasser` calls a procedure named `firstReceiver`, passing `pp` as an actual parameter.
3. `firstReceiver` calls `secondReceiver`, again passing `pp` as an actual parameter.
4. `secondReceiver` calls `pp` (first execution of `pp`).
5. `secondReceiver` calls `thirdReceiver`, again passing `pp` as an actual parameter.
6. `thirdReceiver` calls `firstPasser` (indirect recursion), and passes `pp` to `firstPasser` as an actual parameter.
7. `firstPasser` (executing recursively) calls `pp` (second execution of `pp`).

Thus the procedure `pp` is called first from `secondReceiver`, and then from the second (recursive) execution of `firstPasser`.

Suppose that `pp` accesses an entity named `xxx`, which is not local to `pp`; and suppose that each of the other procedures has a local entity named `xxx`.

Each time `pp` is called, which `xxx` does it access? The answer is that in *each* case, `pp` accesses the `xxx` that is local to the *first* execution of `firstPasser` — that is, the `xxx` that was accessible when `pp` was originally passed as an actual parameter.

7.3.4 Functional Parameters

When the formal parameter is a function-heading, the actual-parameter must be a function-identifier. The identifier in the formal function-heading represents the actual function during the execution of the procedure or function receiving the functional parameter.

Functional parameters are exactly like procedural parameters, with the additional rule that corresponding formal and actual functions must have *identical* result-types.

7.3.5 Parameter List Compatibility

Parameter list compatibility is required of the parameter lists of corresponding formal and actual procedural or functional parameters.

Two formal-parameter-lists are compatible if they contain the same number of parameters and if the parameters in corresponding positions match. Two parameters match if one of the following is true:

- They are both value parameters of *identical* type.
- They are both variable parameters of *identical* type.
- They are both procedural parameters with compatible parameter lists.
- They are both functional parameters with compatible parameter lists and *identical* result-types.

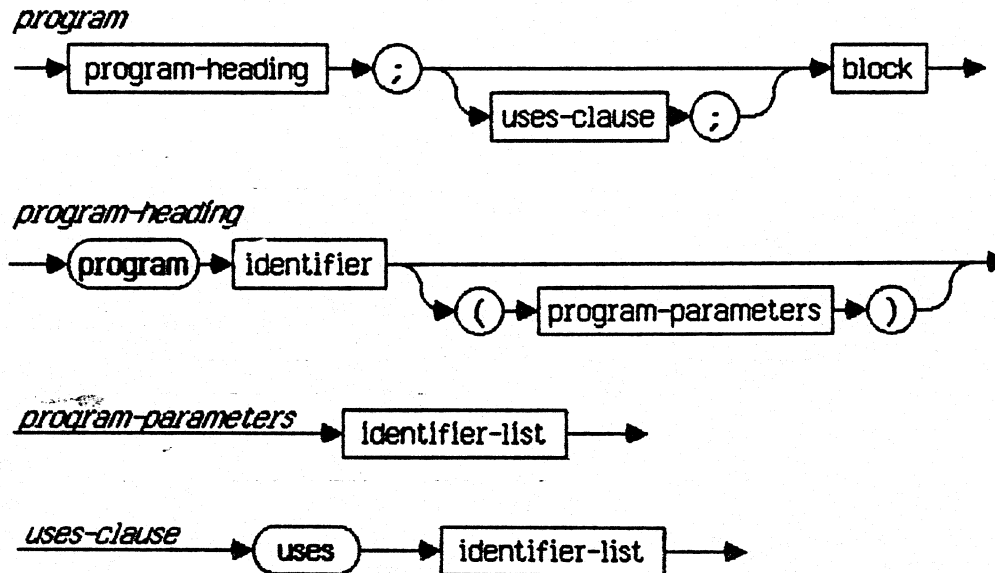
Chapter 8 PROGRAMS

8.1	Syntax.....	8-3
8.2	Program-Parameters	8-3
8.3	Segmentation	8-3

PROGRAMS

8.1 Syntax

A Pascal program has the form of a procedure declaration except for its heading and an optional uses-clause.



The occurrence of an identifier immediately after the word **program** declares it as the program's identifier.

The **uses-clause** identifies all units required by the program, including units that it uses directly and other units that are used by those units.

8.2 Program-Parameters

Currently, any program-parameters are purely decorative and are totally ignored by the compiler.

8.3 Segmentation

The code of a program's main body is always placed in a run-time segment whose name is a string of blanks (the "blank segment"). Any other block can be placed in a different segment by using the **\$\$** compiler command (see Chapter 12 and Appendix A). If no **\$\$** command is used in the program, all code is placed in the blank segment. Code from a program can be placed in the same segment with code from a regular-unit, but it cannot be mixed with code from an intrinsic-unit (see Chapter 9).

Chapter 9 UNITS

9.1	Regular-Units	9-3
9.1.1	Writing Regular-Units	9-3
9.1.2	Using Regular-Units	9-6
9.2	Intrinsic-Units	9-6
9.3	Units that Use Other Units	9-6

UNITS

A unit is a separately compiled, non-executable object file that can be linked with other object files to produce complete programs. There are two kinds of units, called *regular-units* and *intrinsic-units*. In the current implementation of the workshop, you can use intrinsic-units that are provided, but you cannot write new ones.

Each unit used by a program (or another unit) must be compiled, and its object file must be accessible to the compiler, before the host program (or unit) can be compiled.

9.1 Regular-Units

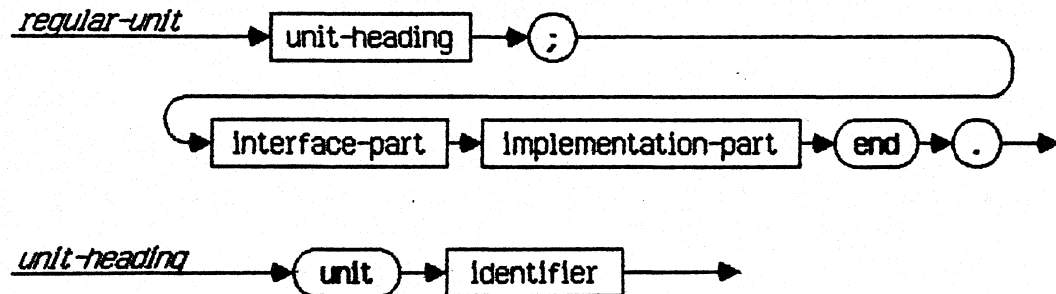
Regular-units can be used as a means of modularizing large programs, or of making code available for incorporation in various programs, without making the source available.

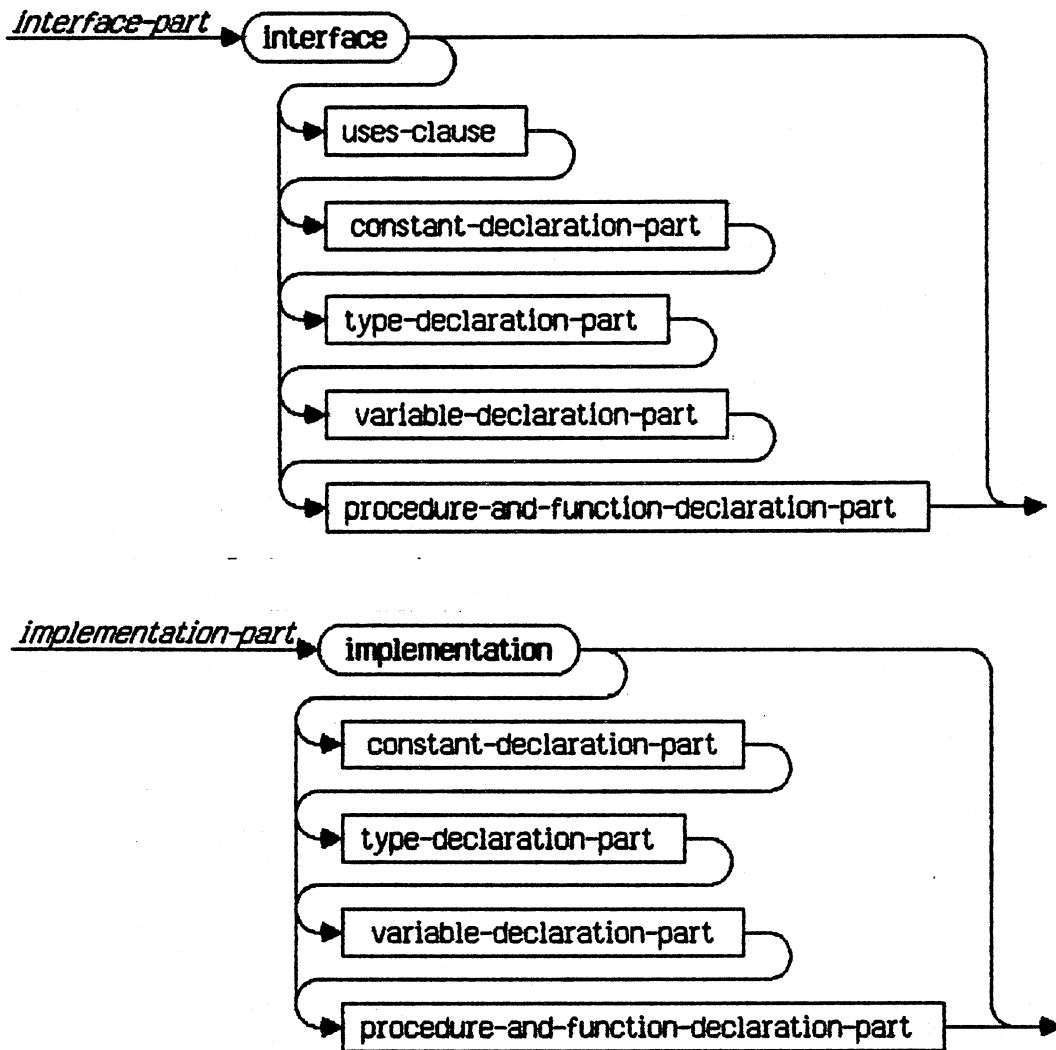
When a program or unit (called the *host*) uses a regular-unit, the Linker inserts a copy of the compiled code from the regular-unit into the host's object file.

By default, the code copied from the regular-unit is placed in the blank segment (see Chapter 8). The code of the entire unit, or of blocks within the unit, can be placed in one or more different segments by using the `$$` compiler command (see Chapter 12).

9.1.1 Writing Regular-Units

The syntax for a regular-unit is:





The interface-part declares constants, types, variables, procedures, and functions that are "public," i.e. available to the host.

The host can access these entities just as if they had been declared in the host. Procedures and functions declared in the interface-part are abbreviated to nothing but the procedure or function name, parameter specifications, and function result-type.

NOTE

Since the interface-part may contain a uses-clause, a unit can use another unit (see Section 9.3).

The implementation-part, which follows the last declaration in the interface-part, begins by declaring any constants, types, variables, procedures, or functions that are "private," i.e. not available to the host.

The public procedures and functions are re-declared in the implementation-part. The parameters and function result types are omitted from these declarations, since they were declared in the interface-part; and the procedure and function blocks, omitted in the interface-part, are included in the implementation-part.

In effect, the procedure and function declarations in the interface are like forward declarations, although the `forward` directive is not used. Therefore, these procedures and functions can be defined and referenced in any sequence in the implementation.

NOTES

There is no "Initialization" section in Pascal units on the Lisa (unlike Apple II and Apple III Pascal). If a unit requires initialization of its data, it should define a public procedure that performs the initialization, and the host should call this procedure.

Also note that global labels cannot be declared in a unit.

A short example of a unit is:

```

unit Simple;
INTERFACE                                {public objects declared}
  const FirstValue=1;
  procedure AddOne(var Incr:integer);
  function Add1(Incr:integer):integer;
IMPLEMENTATION
  procedure AddOne;                        {note lack of parameters...}
  begin
    Incr:=Incr+1
  end;
  function Add1;                            {...and lack of function result type}
  begin
    Add1:=Incr+1
  end
end.

```

9.1.2 Using Regular-Units

The syntax for a uses-clause is given in Chapter 8. Note that in a host program, the uses-clause (if any) must immediately follow the program-heading. In a host unit, the uses-clause (if any) immediately follows the symbol interface. Only one uses-clause may appear in any host program or unit; it declares all units used by the host program or unit.

See Section 9.3 for the case where a host uses a unit that uses another unit.

It is necessary to specify the file to be searched for regular units. The \$U compiler command specifies this file. See Chapter 12 for more details.

Assume that the example unit `Simple` (see above) is compiled to an object file named `APPL:SIMPLE.OBJ`. The following is a short program that uses `Simple`. It also uses another unit named `Other`, which is in file `APPL:OTHER.OBJ`.

```

program CallSimple;
  uses {$U APPL:SIMPLE.OBJ}    {file to search for units}
       Simple,                {use unit Simple}
       {$U APPL:OTHER.OBJ}    {file to search for units}
       Other;                 {use unit Other}
  var i:integer;
  begin
    i:=FirstValue;            {FirstValue is from Simple}
    write('i+1 is ',Add1(i)); {Add1 is defined in Simple}
    write(xyz(i))             {xyz is defined in Other}
  end.

```

9.2 Intrinsic-Units

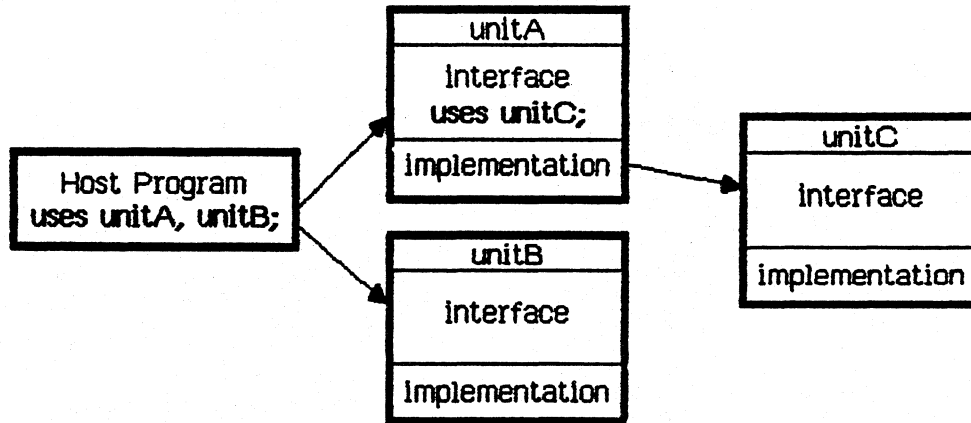
The only intrinsic-units you can use are the ones provided with the Workshop software.

Intrinsic-units provide a mechanism for Pascal programs to share common code, with only one copy of the code in the system. The code is kept on disk, and when loaded into memory it can be executed by any program that declares the intrinsic-unit (via a uses-clause, the same as for regular-units).

By default, the system looks up all intrinsic-units in the system intrinsics library file, `INTRINSIC.LIB`. All intrinsic-units are referenced in this library, so the \$U filename compiler command is not needed with intrinsic-units.

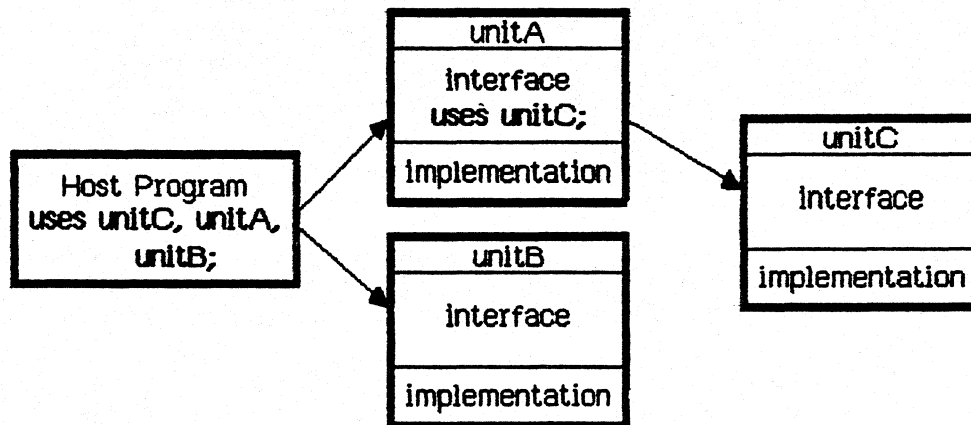
9.3 Units that Use Other Units

As explained above, the uses-clause in the host must name all units that are used by the host. Here "used" means that the host directly references something in the interface of the unit. Consider the first diagram on the next page.



The host program directly references the interfaces of `unitA` and `unitB`; the uses-clause names both of these units. The implementation-part of `unitA` also references the interface of `unitC`, but it is not necessary to name `unitC` in the host-program's uses-clause.

In some cases, the uses-clause must also name a unit that is not directly referenced by the host. The following diagram is exactly like the previous one except that this time the *interface* of `unitA` references the interface of `unitC`, and `unitC` must be named in the host-program's uses-clause. Note that `unitC` must be named *before* `unitA`.



In a case like this, the documentation for `unitA` should state that `unitC` must be named in the uses-clause before `unitA`.

Chapter 10

INPUT/OUTPUT

10.1	Introduction to I/O	10-3
10.1.1	Device Types.....	10-4
10.1.2	External File Species	10-4
10.1.3	The Reset Procedure	10-5
10.1.4	The Rewrite Procedure	10-7
10.1.5	The Close Procedure	10-8
10.1.6	The Ioresult Function	10-9
10.1.7	The Eof Function	10-9
10.2	Record-Oriented I/O.....	10-10
10.2.1	The Get Procedure	10-10
10.2.2	The Put Procedure	10-11
10.2.3	The Seek Procedure	10-11
10.3	Text-Oriented I/O	10-12
10.3.1	The Read Procedure.....	10-13
10.3.1.1	Read with a Char Variable	10-14
10.3.1.2	Read with an Integer or Longint Variable	10-14
10.3.1.3	Read with a Real Variable	10-15
10.3.1.4	Read with a String Variable	10-15
10.3.1.5	Read with a Packed Array of Char Variable	10-16
10.3.2	The ReadIn Procedure.....	10-16
10.3.3	The Write Procedure	10-17
10.3.3.1	Output-Specs	10-18
10.3.3.2	Write with a Char Value	10-18
10.3.3.3	Write with an Integer or Longint Value	10-18
10.3.3.4	Write with a Real Value	10-18
10.3.3.5	Write with a String Value	10-19
10.3.3.6	Write with a Packed Array of Char Value	10-19
10.3.3.7	Write with a Boolean Value	10-19
10.3.4	The WriteIn Procedure	10-20
10.3.5	The Eoln Function	10-20
10.3.6	The Page Procedure	10-21
10.3.7	Keyboard Testing and Screen Cursor Control	10-21
10.3.7.1	The Keypress Function.....	10-21
10.3.7.2	The Gotoxy Procedure	10-21
10.4	Untyped File I/O.....	10-21
10.4.1	The Blockread Function.....	10-22
10.4.2	The Blockwrite Function.....	10-22

INPUT/OUTPUT

This chapter describes the standard ("built-in") I/O procedures and functions of Pascal on the Lisa.

Standard procedures and functions are predeclared. Since all predeclared entities act as if they were declared in a "block" surrounding the program, no conflict arises from a declaration that redefines the same identifier within the program.

NOTE

Standard procedures and functions cannot be used as actual procedural and functional parameters.

This chapter and Chapter 11 use a modified BNF notation, instead of syntax diagrams, to indicate the syntax of actual-parameter-lists for standard procedures and functions.

Example:

Parameter List: new(p [, t1, ... tn])

This represents the syntax of the actual-parameter-list of the standard procedure `new`, as follows:

- `p`, `t1`, and `tn` stand for actual-parameters. Notes on the types and interpretations of the parameters accompany the syntax description.
- The notation `t1, ... tn` means that any number of actual-parameters can appear here, separated by commas.
- Square brackets `[]` indicate parts of the syntax that can be omitted.

Thus the syntax shown here means that the `p` parameter is required. Any number of `t` parameters may appear, with separating commas, or there may be no `t` parameters.

10.1 Introduction to I/O

This section covers the I/O concepts and procedures that apply to all file types. This includes the types `text` (see Section 10.3) and "untyped" files (see Section 10.4).

To use a Pascal file variable (any variable whose type is a file-type), it must be associated with an external file. The external file may be a named collection of information stored on a peripheral device, or (for certain file-types) it may be the peripheral device itself.

The association of a file variable with an external file is made by opening the file. An existing file is opened via the reset procedure, and a new file is created and opened via the rewrite procedure.

NOTE

Pascal on the Lisa does not provide automatic I/O checking. To check the result of any particular I/O operation, use the `ioresult` function described in Section 10.1.6.

10.1.1 Device Types

For purposes of Pascal I/O, there are two types of peripheral devices:

- A file-structured device is one that stores files of data, such as a diskette.
- A character device is one whose input and output are streams of individual bytes, such as the Lisa screen and keyboard or a printer.

10.1.2 External File Species

There are three "species" of external files that can be used in Pascal I/O operations:

- A datafile is any file that is stored on a file-structured device and was *not* originally created in association with a file variable of type `text`.
- A textfile is a file that is stored on a file-structured device and was originally created in association with a file variable of type `text`. Textfiles are stored in a specialized format (see Section 10.3).
- A character device can be treated as a file.

Table 10-1 summarizes the effects of all possible combinations of different file variable types and external file species. The "ordinary cases" in the table reflect the basic intent of the various file-types. Other combinations, such as block-oriented access to a textfile via a variable of type `file`, are legal but may require cautious programming.

Table 10-1
 Combinations of File Variable Types with External File Species
 and Categories

	var f: file of someType;	var f: text;	var f: file;
datafile	<u>Ordinary case.</u> After reset, f [^] = 1st record file.	(Textfile format assumed!) After reset*, f [^] is unspecified.	<u>Ordinary case.</u> Block access.
textfile	(Textfile format not assumed!) After reset*, f [^] = 1st record of file (as declared).	<u>Ordinary case.</u> Textfile format assumed. After reset, f [^] is unspecified.	(Textfile format not assumed!) Block access.
character device	After reset, f [^] = 1st char. from device (system waits for it!). I/O error if file record type not byte-sized.	<u>Ordinary case.</u> After reset, f [^] is unspeci- fied (no wait for input char).	Block access, if allowed by device.
* In these cases, the <code>loresult</code> function will return a "warning" (i.e., a negative number) immediately after the reset operation.			

10.1.3 The Reset Procedure

Opens an existing file.

Parameter List: `reset(f, title)`

1. `f` is a variable-reference that refers to a variable of file-type. The file must not be open.
2. `title` is an expression with a string value. The string should be a valid pathname for a file on a file-structured device, or a pathname for a character device.

NOTE

Both parameters are required (unlike Apple II and Apple III Pascal, where the second parameter is optional).

Reset(f, title) finds an existing external file with the pathname **title**, and associates **f** with this external file. (If there is no existing external file with the pathname **title**, an I/O error occurs; see Section 10.1.6.)

If **title** is the pathname of a character device, then

- **Eof(f)** becomes **false**.
- If **f** is of type **text**, the value of f^{\wedge} is unspecified. The next **read** or **readln** on **f** will wait until a character is available for input, and begin reading with that character.
- If **f** is of type **file** and the device is one that allows block access, there is no file buffer variable f^{\wedge} and the "current file position" is set to the first block (block 0) of the file. If the device does not allow block access, an I/O error occurs (see Section 10.1.6).
- If **f** is not of type **text** or **file**, its component-type must be a "byte-size" type such as the type **-128..127**. Note that **char** is not a byte-size type! If the component-type of **f** is not byte-size, an I/O error occurs (see Section 10.1.6).

If no I/O error occurs, the system waits until a character is available from the device and then assigns the character's 8-bit code to f^{\wedge} .

If **title** is the pathname for an existing file on a file-structured device, then

- **Eof(f)** becomes **false** if the external file is not empty. If the external file is empty, **eof(f)** becomes **true**.
- If **f** is not of type **text** or **file**, **reset** sets the "current file position" to the first record in the external file, and assigns the value of this record to the file buffer variable f^{\wedge} . If the external file is a textfile, the **ioresult** function will return a negative number as a warning (see Section 10.1.6).
- If **f** is of type **text**, the value of f^{\wedge} is unspecified. If the file is a textfile, the next **read** or **readln** on **f** will begin at the first character of **f**. If the file is a datafile, it will be treated as if it were a textfile (see Section 10.3) and the **ioresult** function will return a negative number as a warning (see Section 10.1.6).
- If **f** is of type **file**, there is no file buffer variable f^{\wedge} and the "current file position" is set to the first block (block 0) of the file.

10.1.4 The Rewrite Procedure

Creates and opens a new file.

Parameter List: `rewrite(f, title)`

1. `f` is a variable-reference that refers to a variable of file-type.
2. `title` is an expression with a string value. The string should be a valid pathname for a file on a file-structured device, or a pathname for a character device.

If `f` is already open, an I/O error occurs (see Section 10.1.6).

If `title` is the pathname of a character device, then

- `Eof(f)` becomes false.
- `Rewrite(f, title)` simply associates `f` with the device and opens `f`.
- The status of the device is not affected.
- The value of `f^` becomes unspecified.

If `title` is the pathname for a new file on a file-structured device, then

- `Eof(f)` becomes true.
- `Rewrite(f, title)` creates a new external file with the pathname `title`, and associates `f` with the external file. This is the only way to create a new external file.
- The species of the new external file is set according to the type of `f` -- "textfile" for type `text`, or "datafile" for any other type.
- The value of `f^` becomes unspecified.
- If `f` is not of type `file`, the "current file position" is set to "just before" the first record or character position of the new external file.
- If `f` is of type `file`, the "current file position" is set to block 0 (the first block in the file).
- If `f` is subsequently closed with any option other than `lock` or `crunch` (see Section 10.1.5), the new external file is discarded at that time. Closing `f` with `lock` or `crunch` is the only way to make the new external file permanent.
- If `title` is the pathname of an existing external file, the existing file will be discarded only when `f` is subsequently closed with the `lock` or `crunch` option (see Section 10.1.5).

Unspecified effects are caused if the current file position of a file `f` is altered while the file-buffer `f^` is an actual variable parameter, or an element of the record-variable-reference list of a `with`-statement, or both.

10.1.5 The Close Procedure

Closes a file.

Parameter List: `close(f [, option])`

1. `f` is a variable-reference that refers to a variable of file-type.
2. `option` (may be omitted) is an identifier from the list given below. If omitted, the effect is the same as using the identifier **normal**.

`close(f, option)` closes `f`, if `f` is open. The association between `f` and its external file is broken and the file system marks the external file "closed". If `f` is not open, the `close` procedure has no effect.

The `option` parameter controls the disposition of the external file, if it is not a character device. If it is a character device, `f` is closed and the status of the device is unchanged.

The identifiers that can be used as actual-parameters for `option` are as follows:

- **normal** — If `f` was opened using `rewrite`, it is deleted from the directory. If `f` was opened with `reset`, it remains in the directory. This is the default option, in the case where the `option` parameter is omitted.
- **lock** — If the external file was opened with `rewrite`, it is made permanent in the directory.

If `f` was opened with `rewrite` and a title that matches an existing file, the old file is deleted (unless the safety switch is "on"). If the old file has the safety switch "on," it remains in the directory and the new file is deleted.

If `f` was opened with `reset`, a **normal** close is done.

- **purge** — The external file is deleted from the directory (unless the safety switch is "on"). In the special case of a file that already exists and is opened with `rewrite`, the original file remains in the directory, unchanged.
- **crunch** — This is like **lock** except that it locks the end-of-file to the point of last access; i.e., everything after the last record or character accessed is thrown away.

All closes regardless of the `option` will cause the file system to mark the external file "closed" and will make the value of `f` unspecified.

If a program terminates with a file open (i.e., if `close` is omitted), the system automatically closes the file with the **normal** option.

NOTE

If you open an existing file with `reset` and modify the file with any write operation, the contents are immediately changed no matter what close option you specify.

10.1.6 The Ioresult Function

Pascal on the Lisa does not provide automatic I/O checking. To check the result of any particular I/O operation, you must use the `ioresult` function.

Result type: integer

Parameter List: no parameters

`ioresult` returns an integer value which reflects the status of the last completed I/O operation. The codes are given in the *Workshop Reference Manual for the Lisa*. Note that the code 0 indicates successful completion, positive codes indicate errors, and negative codes are "warnings" (see Table 10-1).

Note that the codes returned by `ioresult` are not the same as the codes used in Apple II and Apple III Pascal.

NOTES

The `read`, `readln`, `write`, and `writeln` procedures described in Section 10.3 may actually perform multiple I/O operations on each call. After one of these procedures has executed, `ioresult` will return a code for the status of the *last* of the multiple operations.

Also, beware of the following common error in diagnostic code:

```
read(foo);
writeln('ioresult=', ioresult)
```

The intention is to write out the status of the read operation, but instead the status written out will be that of the write operation on the string 'ioresult='.

10.1.7 The Eof Function

Detects the end of a file.

Result Type: boolean

Parameter List: eof [(f)]

1. `f` is a variable-reference that refers to a variable of file-type.

If the parameter-list is omitted, the function is applied to the standard file input (see Section 10.3).

After a `get` or `put` operation, `eof(f)` returns `true` if the current file position is beyond the last external file record, or the external file contains no records; otherwise, `eof(f)` returns `false`. Specifically, this means the following:

- After a `get`, `eof(f)` returns `true` if the `get` attempted to read beyond the last file record (or the file is empty).
- After a `put`, `eof(f)` returns `true` if the record written by the `put` is now the last file record.

If `f` is a character device, `eof(f)` will always return `false`.

See Section 10.3 for the behavior of `eof(f)` after a `read` or `readln` operation.

NOTE

Whenever `eof(f)` is `true`, the value of the file buffer variable `f^` is unspecified.

10.2 Record-Oriented I/O

This section covers the `get`, `put`, and `seek` procedures, which perform record-oriented I/O; that is, they consider a file to be a sequence of variables of the type specified in the file-type. These procedures are not allowed with files of type `file`.

The effects of `get` and `put` are unspecified with files of type `text`, and `seek` has no effect with files of type `text`. The `text` type is supported by specialized procedures described in Section 10.3.

10.2.1 The Get Procedure

Reads the next record in a file.

Parameter List: `get(f)`

1. `f` is a variable-reference that refers to a variable of file-type. The file must be open.

If `eof(f)` is `false`, `get(f)` advances the current file position to the next file record, and assigns the value of this record to `f^`. If no next component exists, then `eof(f)` becomes `true`, and the value of `f^` becomes unspecified.

If `eof(f)` is `true` when `get(f)` is called, then `eof(f)` remains `true`, and the value of `f^` becomes unspecified.

If the external file is a character device, `eof(f)` is always `false` and there is no "current file position." In this case, `get(f)` waits until a value is ready for input and then assigns the value to `f^`.

10.2.2 The Put Procedure

Writes the current record in a file.

Parameter List: `put(f)`

1. `f` is a variable-reference that refers to a variable of file-type. The file must be open.

If `eof(f)` is false, `put(f)` advances the current file position to the next file record and then writes the value of `f^` to `f` at the new file position. If the new file position is beyond the end of the file, `eof(f)` becomes true, and the value of `f^` becomes unspecified.

If `eof(f)` is true, `put(f)` appends the value of `f^` to the end of `f` and `eof(f)` remains true.

If the external file is a character device, `eof(f)` is always false, there is no "current file position," and the value of `f^` is sent to the device.

NOTE

If `put` is called immediately after a file is opened with `reset`, the `put` will write the *second* record of the file (since the `reset` sets the current position to the first record and `put` advances the position before writing). To get around this and write the first record, use the `seek` procedure (see Section 10.2.3).

10.2.3 The Seek Procedure

Allows access to an arbitrary record in a file.

Parameter List: `seek(f, n)`

1. `f` is a variable-reference that refers to a variable of file-type. The file must be open.
2. `n` is an expression with an integer value that specifies a record number in the file. Note that records in files are numbered from 0.

If the file is a character device or is of type `text`, `seek` does nothing. Otherwise, `seek(f, n)` affects the action of the next `get` or `put` from the file, forcing it to access file record `n` instead of the "next" record. `Seek(f, n)` does not affect the file-buffer `f^`.

A `get` or `put` *must* be executed between `seek` calls. The result of two consecutive `seeks` with no intervening `get` or `put` is unspecified. Immediately after a `seek(f, n)`, `eof(f)` will return false; a following `get` or `put` will cause `eof` to return the appropriate value.

NOTE

The record number specified in a `seek` call is not checked for validity. If the number is not the number of a record in the file and the program tries to get the specified record, the value of the file-buffer becomes unspecified and `eof` becomes true.

10.3 Text-Oriented I/O

This section describes input and output using file variables of the standard type `text`. Note that in Pascal on the Lisa, the type `text` is distinct from file of `char` (see Section 3.2.4).

When a `text` file is opened, the external file is interpreted in a special way. It is considered to represent a sequence of characters, usually formatted into lines by CR characters (ASCII 13).

The Lisa keyboard and the Workshop screen appear to a Pascal program to be built-in files of type `text` named `input` and `output` respectively. These files need not be declared and need not be opened with `reset` or `rewrite`, since they are always open.

When a program is taking input from `input`, typed characters are echoed on the Workshop screen. In addition to the `input` file, the Lisa keyboard is also represented as the character device `-KEYBOARD`. To get keyboard input without echoing on the screen, you can open a file variable of type `text` with `-KEYBOARD` as the external file pathname.

Other interactive devices can also be represented in Pascal programs as files of type `text`.

When a `text` file is created on a file-structured device, the external file is a textfile. It contains information other than the actual sequence of characters represented, as follows:

- The stored file is a sequence of 1024-byte pages.
- Each page contains some number of *complete* lines of text and is padded with null characters (ASCII 0) after the last line.
- Two 512-byte header blocks are also present at the beginning of the file.
- A sequence of spaces in the text *may* be compressed into a two-byte code, namely a DLE character (ASCII 16) followed by a byte containing 32 plus the number of spaces represented.

All of this special formatting is invisible to a Pascal program if the file is accessed via a file variable of type `text` (but visible via a file variable of any other file-type).

Certain things that can be done with a record-structured file are impossible with a file variable of type `text`:

- The `seek` procedure does nothing with a file variable of type `text`.
- The effects of `get` and `put` are unspecified with a file variable of type `text`.
- The contents of the file buffer variable are unspecified with a file variable of type `text`.
- A file variable of type `text` that is opened with `reset` cannot be used for output, and one opened with `rewrite` cannot be used for input. Results are unspecified if either of these operations is attempted.

In place of these capabilities, text-oriented I/O provides the following:

- Automatic conversion of each input CR character into a space.
- The `eofln` function to detect when the end of an input line has been reached.
- The `read` procedure, which can read `char` values, string values, packed array of `char` values, and numeric values (from textual representations).
- The `write` procedure, which can write `char` values, string values, packed array of `char` values, numeric values, and boolean values (as textual representations).
- Line-oriented reading and writing via the `readln` and `writeln` procedures.
- The `page` procedure, which outputs a form-feed character to the external file.
- Automatic conversion of input DLE-codes to the sequences of spaces that they represent. Note that output sequences of spaces are not converted to DLE-codes.
- Automatic skipping of header blocks and null characters during input.
- Automatic generation of textfile header blocks, and automatic padding of textfile pages with null characters on output.

10.3.1 The Read Procedure

Reads one or more values from a `text` file into one or more program variables.

Parameter List: `read([f,] v1 [, v2, ... vn])`

The syntax of the parameter-list of `read` allows an indefinite number of actual-parameters. Consecutive actual-parameters are separated by commas, just as in a normal parameter-list.

1. `f` (may be omitted) is a variable-reference that refers to a variable of type `text`. The file must be open. If `f` is omitted, the procedure reads from the standard `text` file `input`, which represents the Lisa keyboard.

2. $v_1 \dots v_n$ are input variables. Each is a variable parameter, used as a destination for data read from the file. Each input variable must be a variable-reference that refers to a variable of one of the following types:
- `char`, `integer`, or `longint` (or a subrange of one of these)
 - `real`
 - a string-type or a packed array of `char` type.

These are the types of data that can be read (as textual representations) from a file. At least one input variable must be present.

`Read(f,v1,...,vn)` is equivalent to:

```
begin
  read(f, v1);
  ...
  read(f, vn)
end
```

NOTE

`Read` can also be used to read from a file `fil` that is not a `text` file. In this case `read(fil,x)` is equivalent to:

```
begin
  x := fil^;
  get(fil)
end
```

10.3.1.1 Read with a Char Variable

If `f` is of type `text` and `v` is of type `char`, the following things are true immediately after `read(f,v)`:

- `Eof(f)` will return `true` if the read attempted to read beyond the last character in the external file.
- `Eoln(f)` will return `true`, and the value of `v` will be a space, if the character read was the CR character. `Eoln(f)` will also return `true` if `eof(f)` is `true`.

10.3.1.2 Read with an Integer or Longint Variable

If `f` is of type `text` and `v` is of type `integer`, subrange of `integer`, or `longint`, then `read(f,v)` implies the reading from `f` of a sequence of characters that form a signed whole number according to the syntax of Section 1.4 (except that hexadecimal notation is not allowed). If the value read is assignment-compatible with the type of `v`, it is assigned to `v`; otherwise an error occurs.

In reading the sequence of characters, preceding blanks and CRs are skipped. Reading ceases as soon as a character is reached that, together with the characters already read, does not form part of a signed whole number.

An error occurs if a signed whole number is not found after skipping any preceding blanks and CRs.

If *f* is of type *text*, the following things are true immediately after `read(f,v)`:

- `Eof(f)` will return **true** if the last character in the numeric string was the last character in the external file.
- `Eoln(f)` will return **true** if the last character in the numeric string was the last character on the line (not counting the CR character). `Eoln(f)` will also return **true** if `eof(f)` is **true**.

10.3.1.3 Read with a Real Variable

If *f* is of type *text* and *v* is of type *real*, then `read(f,v)` implies the reading from *f* of a sequence of characters that represents a *real* value. The *real* value is assigned to the variable *v*.

In reading the sequence of characters, preceding blanks and CRs are skipped. Reading ceases as soon as a character is reached that, together with the characters already read, does not form a valid representation. A "valid representation" is either of the following:

- A finite *real*, *integer*, or *longint* value represented according to the signed-number syntax of Section 1.4 (except that hexadecimal notation is not allowed). An *integer* or *longint* value is converted to type *real*.
- An infinite value or Nan represented as described in Appendix D.

An error occurs if a valid representation is not found after skipping any preceding blanks and CRs.

Immediately after `read(f,v)` where *v* is a real variable, the status of `eof(f)` and `eoln(f)` are the same as for an *integer* variable (see Section 10.3.1.2 above).

10.3.1.4 Read with a String Variable

If *f* is of type *text* and *v* is of string-type, then `read(f,v)` implies the reading from *f* of a sequence of characters up to *but not including* the next CR or the end of the file. The resulting character-string is assigned to *v*. An error occurs if the number of characters read exceeds the size attribute of *v*.

NOTE

Read with a string variable does not skip to the next line after reading, and the CR is left waiting in the input buffer. For this reason, you cannot use successive **read** calls to read a sequence of strings, as they will never get past the first CR — after the first **read**, each subsequent **read** will see the CR and will read a zero-length string.

Instead, use **readln** to read string values (see Section 10.3.2). **Readln** skips to the beginning of the next line after reading.

The following things are true immediately after **read(f,v)**:

- **Eof(f)** will return true if the line read was the last line in the file.
- **Eoln(f)** will always return true.

10.3.1.5 Read with a Packed Array of Char Variable

If **f** is of type **text** and **v** is a packed array of **char**, then **read(f,v)** implies the reading from **f** of a sequence of characters. Characters are read into successive character positions in **v** until all positions have been filled, or until a CR or the end of the file is encountered. If a CR or the end-of-file is encountered, it is not read into **v**; the remaining positions in **v** are filled with spaces.

10.3.2 The Readln Procedure

The **readln** procedure is an extension of **read**. Essentially it does the same thing as **read**, and then skips to the next line in the input file.

Parameter List: The syntax of the parameter list of **readln** is the same as that of **read**, except as follows:

- A **readln** call with no input variables is allowed. *Example:*

readln(sourcefile)

- The parameter-list can be omitted altogether.

If the first parameter does not specify a file, or if the parameter-list is omitted, the procedure reads from the standard file **input**, which represents the Lisa keyboard.

Readln(f), with no input-variables, causes a skip to the beginning of the next line (if there is one, else to the end-of-file).

`Readln` can *only* be used on a `text` file. Except for this restriction, `readln(f,v1,...,vn)` is equivalent to:

```
begin
  read(f, v1, ..., vn);
  readln(f)
end
```

The following things are true immediately after `readln(f,v)`, regardless of the type of `v`:

- `Eof(f)` will return `true` if the line read was the last line in the external file.
- `Eoln(f)` will always return `false`.

10.3.3 The Write Procedure

Writes one or more values to a `text` file.

Parameter List: `write([f,] p1 [, p2, ... pn])`

The syntax of the parameter list of `write` allows an indefinite number of actual-parameters.

1. `f` (may be omitted) is a variable-reference that refers to a variable of type `text`. The file must be open. If `f` is omitted, the procedure writes to the standard file `output`, which represents the Workshop screen.
2. `p1 ... pn` are output-specs. Each output-spec includes an output expression, whose value is to be written to the file. As explained below, an output-spec may also contain specifications of field-width and number of decimal places. Each output expression must have a result of type `integer`, `longint`, `real`, `boolean`, `char`, a string-type, or a packed array of `char` type. These are the types of data that can be written (as textual representations) to a file. At least one output-spec must be present.

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

```
begin
  write(f,p1);
  ...
  write(f,pn)
end
```

Immediately after `write(f)`, both `eof(f)` and `eoln(f)` will return `true`.

 NOTE

`write` can also be used to write onto a file `fil` that is not a text file. In this case `write(fil,x)` is equivalent to:

```
begin
  fil^ := x;
  put(fil)
end
```

10.3.3.1 Output-Specs

Each output-spec has the form

```
OutExpr [ : Minwidth [ : DecPlaces ] ]
```

where `OutExpr` is an output expression. `Minwidth` and `DecPlaces` are expressions with integer or longint values.

`Minwidth` specifies the *minimum* field width, with a default value that depends on the type of the value of `OutExpr` (see below). `Minwidth` should be greater than zero; otherwise, the results are unspecified. Exactly `Minwidth` characters are written (using leading spaces if necessary), except when `OutExpr` has a *numeric* value that requires more than `Minwidth` characters; in this case, enough characters are written to represent the value of `OutExpr`.

`DecPlaces` specifies the number of decimal places in a fixed-point representation of a real value. It can be specified only if `OutExpr` has a real value, and if `Minwidth` is also specified. If `DecPlaces` is not specified, a floating-point representation is written.

10.3.3.2 Write with a Char Value

If `OutExpr` has a char value, the character is written on the file `f`. The default value for `Minwidth` is one.

10.3.3.3 Write with an Integer or Longint Value

If `OutExpr` has an integer or longint value, its decimal representation is written on the file `f`. The default value for `Minwidth` is 8. The representation consists of the digits representing the value, prefixed by a minus sign if the value is negative, and any leading spaces that may be required to satisfy `Minwidth`. If the representation requires more than `Minwidth` characters, `Minwidth` is ignored.

10.3.3.4 Write with a Real Value

If `OutExpr` has a real value, the default value for `Minwidth` is 12.

If `OutExpr` has an infinite value, it is output as a string of at least two "+" characters or at least two "-" characters. If `OutExpr` is a NaN, it is output as the character string "NaN", possibly followed by a string of characters enclosed by single-quotes. See Section 10.3.3.5 for details on string output.

If `OutExpr` has a finite value, its decimal representation is written on the file `f`. This representation is the nearest possible decimal representation, depending on `MinWidth` and `DecPlaces`. If the unrounded value is exactly half way between two possible representations, the representation whose least significant digit is even is written out.

If `DecPlaces` is not specified, a floating-point representation is written as follows:

- If `MinWidth` is less than 6, then its value is set to 6 (internally). This is the minimum usable width for writing a floating-point representation.
- If the sign of the value of `OutExpr` is negative, a minus sign is written; otherwise, a space is written.
- If `MinWidth` \geq 8, the significant digits are written with one digit to the left of the decimal point and $(\text{MinWidth} - 7)$ digits to the right of the decimal point.
- If `MinWidth` $<$ 8, the most significant digit is written and the decimal point is omitted.
- The exponent is written as the letter "E", an explicit "+" or "-" sign, and two digits.

If `DecPlaces` is specified, a fixed-point representation is written as follows:

- Enough leading spaces are written to satisfy `MinWidth`.
- If the value is negative, the minus sign "-" is written; if it is not negative, a space is written.
- If `DecPlaces` $>$ 0, the significant digits are written with the integer part of the value to the left of the decimal point. The next `DecPlaces` digits are written to the right of the decimal point.
- If `DecPlaces` \leq 0, only the integer part of the value is written and no decimal point is written.

10.3.3.5 Write with a String Value

If the value of `OutExpr` is of string type with length `L`, the default value for `MinWidth` is `L`. If `MinWidth` \geq `L`, the value is written on the file `f` preceded by $(\text{MinWidth} - L)$ spaces. If `MinWidth` $<$ `L`, the first `MinWidth` characters of the string are written.

10.3.3.6 Write with a Packed Array of Char Value

If `E` is of type `packed array of char`, the effect is the same as writing a string whose length is the number of elements in the array.

10.3.3.7 Write with a Boolean Value

If the value of `OutExpr` is of type `boolean`, the string "TRUE" (with a leading space) or the string "FALSE" is written on the file `f`. The default value of

`Minwidth` is 5. If `Minwidth`>5, leading spaces are added; if `Minwidth`<5, the first `Minwidth` characters of the string are written. This is equivalent to:

```
write(f, ' TRUE' :Minwidth)
  or
write(f, ' FALSE' :Minwidth)
```

10.3.4 The `writeln` Procedure

The `writeln` procedure is an extension of `write`. Essentially it does the same thing as `write`, and then writes a CR character to the output file (ending the line).

Parameter List: The syntax of the parameter list of `writeln` is the same as that of `write`, except as follows:

- A `writeln` call with no output-specs is allowed. *Example:*

```
writeln(outputfile)
```

- The parameter-list can be omitted altogether.

If the first parameter does not specify a file, or if the parameter-list is omitted, the procedure writes to the standard file `output`, which represents the workshop screen.

`writeln(f)` writes a CR character to the file `f`.

`writeln` can *only* be used on a text file. Except for this restriction, `writeln(f,p1,...,pn)` is equivalent to:

```
begin
  write(f,p1, ...,pn);
  writeln(f)
end
```

Immediately after `writeln(f)`, both `eof(f)` and `eoln(f)` will return `true`.

10.3.5 The `Eoln` Function

Result Type: `boolean`

Parameter List: `eoln[(f)]`

1. `f` is a variable-reference that refers to a variable of type `text`. The file must be open.

The actual-parameter-list can be omitted entirely. In this case, the function is applied to the standard file `input` (the Lisa keyboard).

`Eoln(f)` returns `true` "if the end of a line has been reached in `f`." The meaning of this depends on whether the external file is a character device, on which I/O procedure was executed last, and on what type of variable was used to receive an input value. For details, see Sections 10.3.1 through 10.3.4.

The end of the file is considered to be the end of a line; therefore `eoln(f)` will return `true` whenever `eof(f)` is `true`.

10.3.6 The Page Procedure

Parameter List: `page(f)`

1. `f` is a variable-reference that refers to a variable of type `text`. The file must be open.

The actual-parameter `f` cannot be omitted. `Page(f)` outputs a form-feed character to the file `f`. This will cause a skip to the top of a new page when `f` is printed.

Note that `page(output)` sends a form-feed to the Workshop screen, but in general this will not clear the screen. For methods of clearing the screen, see the *Workshop Reference Manual for the Lisa*.

10.3.7 Keyboard Testing and Screen Cursor Control

10.3.7.1 The Keypress Function

Tests the Lisa keyboard to see if it has a character awaiting input.

Parameter List: no parameters.

Result Type: `boolean`.

`Keypress` returns `true` if a character has been typed on the Lisa keyboard but has not yet been read, or `false` otherwise. This is done by testing the typeahead queue; if the queue is empty, `keypress` is `false`, else it is `true`.

10.3.7.2 The Gotoxy Procedure

Moves the Workshop screen cursor to a specified location on the screen.

Parameter List: `gotoxy(x, y)`

1. `x` is an expression with an `integer` value. If `x < 0`, the value 0 will be used; if `x > 79`, the value 79 will be used.
2. `y` is an expression with an `integer` value. If `y < 0`, the value 0 will be used; if `y > 31`, the value 31 will be used.

`Gotoxy(x, y)` moves the cursor to the point `(x,y)` on the screen. Note that the point `(0,0)` is the upper left corner of the screen.

10.4 Untyped File I/O

Untyped file I/O operates on an "untyped file," i.e., a variable of type `file` (no component type). An untyped file is treated as a sequence of 512-byte blocks; the bytes are not type-checked but considered as raw data. This can be useful for applications where the data need not be interpreted at all during I/O operations.

The blocks in an untyped file are considered to be numbered sequentially starting with 0. The system keeps track of the current block number; this is block 0 immediately after the file is opened. Each time a block is read, the

current block number is incremented. By default, each I/O operation begins at the current block number; however, an arbitrary block number can be specified.

An untyped file has no file-buffer, and it cannot be used with `get`, `put`, or any of the text-oriented I/O procedures. It can only be used with `reset`, `rewrite`, `close`, `eof`, and the `blockread` and `blockwrite` functions described below.

To use untyped file I/O, an untyped file is opened with `reset` or `rewrite`, and the `blockread` and `blockwrite` functions are used for input and output.

10.4.1 The Blockread Function

Reads one or more 512-byte blocks of data from an untyped file to a program variable, and returns the number of blocks read.

Result Type: Integer

Parameter List: `blockread(f, databuf, count [, blocknum])`

1. `f` is a variable-reference that refers to a variable of type `file`. The file must be open.
2. `databuf` is a variable-reference that refers to the variable into which the blocks of data will be read. The size and type of this variable are not checked; if it is not large enough to hold the data, other program data may be overwritten and the results are unpredictable.
3. `count` is an expression with an integer value. It specifies the maximum number of blocks to be transferred. `Blockread` will read as many blocks as it can, up to this limit.
4. `blocknum` (may be omitted) is an expression with an integer value. It specifies the starting block number for the transfer. If it is omitted, the transfer begins with the current block. Thus the transfers are sequential if the `blocknumber` parameter is never used; if a `blocknumber` parameter is used, it provides random access to blocks.

`Blockread(f, databuf, count, blocknum)` reads blocks from `f` into `databuf`, starting at block `blocknum`. `Count` is the maximum number of blocks read; if the end-of-file is encountered before `count` blocks are read, the transfer ends at that point. The value returned is the number of blocks actually read.

If the last block in the file was read, the current block number is unspecified and `eof(f)` is `true`. Otherwise, `eof(f)` is `false` and the current block number is advanced to the block after the last block that was read.

10.4.2 The Blockwrite Function

Writes one or more 512-byte blocks of data from a program variable to an untyped file, and returns the number of blocks written.

Result Type: Integer

Parameter List: `blockwrite(f, databuf, count [, blocknum])`

1. **f** is a variable-reference that refers to a variable of type **file**. The file must be open.
2. **databuf** is a variable-reference that refers to the variable from which the blocks of data will be written. The size and type of this variable are not checked.
3. **count** is an expression with an **integer** value. It specifies the maximum number of blocks to be transferred. **Blockwrite** will write as many blocks as it can, up to this limit.
4. **blocknum** (may be omitted) is an expression with an **integer** value. It specifies the starting block number for the transfer. If it is omitted, the transfer begins with the current block. Thus the transfers are sequential if the **blocknumber** parameter is never used; if a **blocknumber** parameter is used, it provides random access to blocks.

Blockwrite(f, databuf, count, blocknum) writes blocks into **f** from **databuf**, starting at block **blocknum**. **Count** is the maximum number of blocks written; if disk space runs out before **count** blocks are written, the transfer ends at that point. The value returned is the number of blocks actually written.

If disk space ran out, the current block number is unspecified. Otherwise, the current block number is advanced to the block after the last block that was written.

NOTE

Unlike Apple II and Apple III Pascal, this Pascal does not allow **blockwrite** to write a block at a position beyond the first position after the current end of the file. In other words, you cannot create a block file with gaps in it.

Chapter 11

STANDARD PROCEDURES AND FUNCTIONS

11.1	Exit and Halt Procedures	11-3
11.1.1	The Exit Procedure	11-3
11.1.2	The Halt Procedure	11-3
11.2	Dynamic Allocation Procedures	11-4
11.2.1	The New Procedure	11-4
11.2.2	The HeapResult Function	11-5
11.2.3	The Mark Procedure	11-5
11.2.4	The Release Procedure	11-5
11.2.5	The Memavail Function	11-5
11.3	Transfer Functions	11-6
11.3.1	The Trunc Function	11-6
11.3.2	The Round Function	11-6
11.3.3	The Ord4 Function	11-6
11.3.4	The Pointer Function	11-7
11.4	Arithmetic Functions	11-7
11.4.1	The Odd Function	11-7
11.4.2	The Abs Function	11-7
11.4.3	The Sqr Function	11-8
11.4.4	The Sin Function	11-8
11.4.5	The Cos Function	11-8
11.4.6	The Exp Function	11-8
11.4.7	The Ln Function	11-9
11.4.8	The Sqrt Function	11-9
11.4.9	The Arctan Function	11-9
11.4.10	The Pwroften Function	11-9
11.5	Ordinal Functions	11-10
11.5.1	The Ord Function	11-10
11.5.2	The Chr Function	11-10
11.5.3	The Succ Function	11-10
11.5.4	The Pred Function	11-11
11.6	String Procedures and Functions	11-11
11.6.1	The Length Function	11-11
11.6.2	The Pos Function	11-11
11.6.3	The Concat Function	11-12
11.6.4	The Copy Function	11-12
11.6.5	The Delete Procedure	11-12
11.6.6	The Insert Procedure	11-12
11.7	Byte-Oriented Procedures and Functions	11-13
11.7.1	The Moveleft Procedure	11-13
11.7.2	The Moveright Procedure	11-14
11.7.3	The Sizeof Function	11-14

11.8	Packed Array of Char Procedures and Functions	11-14
11.8.1	The Scaneq Function	11-15
11.8.2	The Scanne Function	11-15
11.8.3	The Fillchar Procedure	11-15

STANDARD PROCEDURES AND FUNCTIONS

This chapter describes all the standard ("built-in") procedures and functions in Pascal on the Lisa, except for the I/O procedures and functions described in Chapter 10.

Standard procedures and functions are predeclared. Since all predeclared entities act as if they were declared in a block surrounding the program, no conflict arises from a declaration that redefines the same identifier within the program.

NOTE

Standard procedures and functions cannot be used as actual procedural and functional parameters.

This chapter uses a modified BNF notation, instead of syntax diagrams, to indicate the syntax of actual-parameter-lists for standard procedures and functions. The notation is explained at the beginning of Chapter 10.

11.1 Exit and Halt Procedures

11.1.1 The Exit Procedure

Exits immediately from a specified procedure or function, or from the main program.

Parameter List: `exit(id)`

1. `id` is the identifier of a procedure or function, or of the main program. If `id` is an identifier defined in the program, it must be in the scope of the `exit` call. Note that this is more restricted than UCSD Pascal.

`Exit(id)` causes an immediate exit from `id`. Essentially, it causes a jump to the end of `id`.

NOTE

The `halt` procedure (see below) can be used to exit the main program from a `unit` without knowing the main program's identifier.

11.1.2 The Halt Procedure

Exits immediately from the main program.

Parameter List: no parameters

`Halt` causes an immediate exit from the main program.

11.2 Dynamic Allocation Procedures

These procedures are used to manage the **heap**, a memory area that is unallocated when the program starts running. The procedure **new** is used for all allocation of heap space by the program. The **mark** and **release** procedures are used together to deallocate heap space, and the **heapresult** function is used to return the status of the last preceding dynamic allocation operation..

11.2.1 The New Procedure

Allocates a new dynamic variable and sets a pointer variable to point to it.

Parameter List: **new(p [, t1, ... tn])**

1. **p** is a variable-reference that refers to a variable of any pointer-type. This is a variable parameter.
2. **t1, ... tn** are constants, used only when allocating a variable of record-type with variants (see below).

New(p) allocates a new variable of the base-type of **p**, and makes **p** point to it. The variable can be referenced as **p^**. Successive calls to **new** allocate contiguous areas.

If the heap does not contain enough free space to allocate the new variable, **p** is set to **nil** and a subsequent call to the **heapresult** function will return a non-zero result.

If the base-type of **p** is a record-type with variants, **new(p)** allocates enough space to allow for the largest variant. The form

new(p, t1, ...tn)

allocates a variable with space for the variants specified by the tag values **t1, ... tn** (instead of enough space for the largest variants). The tag values must be constants. They must be listed contiguously and in the order of their declaration. The tag values are not assigned to the tag-fields by this procedure.

Trailing tag values can be omitted. The space allocated allows for the largest variants for all tag-values that are not specified.

Warning: When a record variable is dynamically allocated with explicit tag values as shown above, you should not make assignments to any fields of variants that are not selected by the tag values. Also, you should not assign an entire record to this record. If you do either of these things, other data can be overwritten without any error being detected at compile time.

11.2.2 The Heapresult Function

Returns the status of the most recent dynamic allocation operation.

Result Type: integer

Parameter List: no parameters

Heapresult returns an integer code that reflects the status of the most recent call on **new**, **mark**, **release**, or **memavail**. The codes are given in the *Workshop Reference Manual*; note that the code for a successful operation is 0.

11.2.3 The Mark Procedure

Sets a pointer to a heap area.

Parameter List: **mark(p)**

1. **p** is a variable-reference that refers to a variable of any pointer-type. This is a variable parameter.

Mark(p) causes the pointer **p** to point to the lowest free area in the heap. The next call to **new** will allocate space beginning at the bottom of this area, and then **p** will be a pointer to this space. The pointer **p** is also placed on a stack-like list for subsequent use with the **release** procedure (see below).

11.2.4 The Release Procedure

Deallocates all variables in a marked heap area.

Parameter List: **release(p)**

1. **p** is a variable-reference that refers to a pointer variable. It must be a pointer that was previously set with the **mark** procedure. The pointer **p** must be on the list created by the **mark** procedure; otherwise an error occurs.

Release(p) removes pointers from the list, back to and including the pointer **p**. The heap areas pointed to by these pointers are deallocated. In other words, **release(p)** deallocates all areas allocated since the the pointer **p** was passed to the **mark** procedure.

11.2.5 The Memavail Function

Returns the maximum possible amount of available memory.

Result Type: longint

Parameter List: no parameters

Memavail returns the maximum number of words (not bytes) of heap and stack space that could ever be available to the program, allowing for possible automatic expansion of the program's data segment. Note that the result of **memavail** can change over time even if the program does not allocate any heap space, because of activities by the operating system or other processes in the system.

11.3 Transfer Functions

The procedures **pack** and **unpack**, described by Jensen and Wirth, are not supported.

11.3.1 The Trunc Function

Converts a real value to a **longint** value.

Result Type: **longint**

Parameter List: **trunc(x)**

1. **x** is an expression with a value of type **real**.

Trunc(x) returns a **longint** result that is the value of **x** rounded to the nearest whole number that is between 0 and **x** (inclusive). If **x** is exactly halfway between two whole numbers, the result is the *even* whole number.

11.3.2 The Round Function

Converts a real value to a **longint** value.

Result Type: **longint**

Parameter List: **round(x)**

1. **x** is an expression with a value of type **real**.

Round(x) returns a **longint** result that is the value of **x** rounded to the nearest whole number. If **x** is exactly halfway between two whole numbers, the result is the whole number with the greatest absolute magnitude.

11.3.3 The Ord4 Function

Converts an ordinal-type or pointer-type value to type **longint**.

Result Type: **longint**

Parameter List: **ord4(x)**

1. **x** is an expression with a value of ordinal-type or pointer-type.

Ord4(x) returns the value of **x**, converted to type **longint**. If **x** is of type **longint**, the result is the same as **x**.

If **x** is of pointer-type, the result is the corresponding physical address, of type **longint**.

If **x** is of type **integer**, the result is the same numerical value represented by **x**, but of type **longint**. This is useful in arithmetic expressions. For example, consider the expression

abc*xyz

where both **abc** and **xyz** are of type **integer**. By the rules given in Section 3.1.1.2, the result of this multiplication is of type **integer** (16 bits). If the

mathematical product of `abc` and `xyz` cannot be represented in 16 bits, the result is the low-order 16 bits. To avoid this, the expression can be written as

`ord4(abc)*xyz`

This expression causes 32-bit arithmetic to be used, and the result is a 32-bit `longint` value.

If `x` is of an ordinal-type other than `integer` or `longint`, the numerical value of the result is the ordinal number determined by mapping the values of the type onto consecutive non-negative integers starting at zero.

11.3.4 The Pointer Function

Converts an `integer` or `longint` value to pointer-type.

Result Type: pointer

Parameter List: `pointer(x)`

1. `x` is an expression with a value of type `integer` or `longint`.

`Pointer(x)` returns a pointer value that corresponds to the physical address `x`. This pointer is of the same type as `nil` and is assignment-compatible with any pointer-type.

11.4 Arithmetic Functions

In general, any real result returned by an arithmetic function is an approximation. There are two exceptions to this: the result of the `abs` function is exact, and the result of the `pwroften` function is exact when the parameter `n` is in the range $0 \leq n \leq 10$.

11.4.1 The Odd Function

Tests whether a whole-number value is odd.

Result Type: boolean

Parameter List: `odd(x)`

1. `x` is an expression with a value of type `integer` or `longint`.

`Odd(x)` returns `true` if `x` is odd; otherwise it yields `false`.

11.4.2 The Abs Function

Returns the absolute value of a numeric value.

Result Type: same as parameter

Parameter List: `abs(x)`

1. `x` is an expression with a value of type `real`, `integer`, or `longint`.

`Abs(x)` returns the absolute value of `x`.

11.4.3 The Sqr Function

Returns the square of a numeric value.

Result Type: depends on parameter (see below)

Parameter List: **sqr(x)**

1. **x** is an expression with a value of type **real**, **integer**, or **longint**.

Sqr(x) returns the square of **x**. If **x** is of type **real**, the result is **real**; if **x** is of type **longint**, the result is **longint**; and if **x** is of type **integer**, the result may be either **integer** or **longint**.

If **x** is of type **real** and floating-point overflow occurs, the result is $+\infty$.

11.4.4 The Sin Function

Returns the sine of a numeric value.

Result Type: **real**

Parameter List: **sin(x)**

1. **x** is an expression with a value of type **real**, **integer**, or **longint**. This value is assumed to represent an angle in radians.

Sin(x) returns the sine of **x**.

11.4.5 The Cos Function

Returns the cosine of a numeric value.

Result Type: **real**

Parameter List: **cos(x)**

1. **x** is an expression with a value of type **real**, **integer**, or **longint**. This value is assumed to represent an angle in radians.

Cos(x) returns the cosine of **x**.

11.4.6 The Exp Function

Returns the exponential of a numeric value.

Result Type: **real**

Parameter List: **exp(x)**

1. **x** is an expression with a value of type **real**, **integer**, or **longint**. All possible values are valid.

Exp(x) returns the value of e^x , where e is the base of the natural logarithms. If floating-point overflow occurs, the result is $+\infty$.

11.4.7 The Ln Function

Returns the natural logarithm of a numeric value.

Result Type: real

Parameter List: $\ln(x)$

1. x is an expression with a value of type **real**, **integer**, or **longint**. All non-negative values are valid; negative values are invalid.

If x is non-negative, $\ln(x)$ returns the natural logarithm (\log_e) of x .

If x is negative, a diagnostic NaN is produced and the Invalid Operation signal is set (see Appendix D).

11.4.8 The Sqrt Function

Returns the square root of a numeric value.

Result Type: real

Parameter List: $\text{sqrt}(x)$

1. x is an expression with a value of type **real**, **integer**, or **longint**. All non-negative values are valid; negative values are invalid.

If x is non-negative, $\text{sqrt}(x)$ returns the positive square root of x .

If x is negative, a diagnostic NaN is produced and the Invalid Operation signal is set (see Appendix D).

11.4.9 The Arctan Function

Returns the arctangent of a numeric value.

Result Type: real

Parameter List: $\text{arctan}(x)$

1. x is an expression with a value of type **real**, **integer**, or **longint**. All numeric values are valid, including $\pm\infty$.

$\text{Arctan}(x)$ returns the principal value, in radians, of the arctangent of x .

11.4.10 The Pwroften Function

Returns a specified power of 10.

Result Type: real

Parameter List: $\text{pwroften}(n)$

1. n is an expression with a value of type **integer**.

If $-45 \leq n \leq 38$, then $\text{pwroften}(n)$ returns 10^n . The result is mathematically exact for $0 \leq n \leq 10$. If $n \leq -46$, the result is 0; if $n \geq 39$, the result is $+\infty$.

11.5 Ordinal Functions

11.5.1 The Ord Function

Returns the ordinal number of an ordinal-type or pointer-type value.

Result Type: integer or longint

Parameter List: ord(x)

1. x is an expression with a value of ordinal-type or pointer-type.

If x is of type integer or longint, the result is the same as x.

If x is of pointer-type, the result is the corresponding physical address, of type longint.

If x is of another ordinal-type, the result is the ordinal number determined by mapping the values of the type onto consecutive non-negative whole numbers starting at zero.

For a parameter of type char, the result is the corresponding ASCII code. For a parameter of type boolean,

ord(false) returns 0

ord(true) returns 1

11.5.2 The Chr Function

Returns the char value corresponding to a whole-number value.

Result Type: char (but see below)

Parameter List: chr(x)

1. x is an expression with an integer or longint value.

Chr(x) returns the char value whose ordinal number (i.e., its ASCII code) is x, if x is in the range 0..255. If x is not in the range 0..255, the value returned is not within the range of the type char, and any attempt to assign it to a variable of type char will cause an error.

For any char value ch, the following is true:

chr(ord(ch)) = ch

11.5.3 The Succ Function

Returns the successor of a value of ordinal-type.

Result Type: same as parameter (but see below)

Parameter List: succ(x)

1. x is an expression with a value of ordinal-type.

Succ(x) returns the successor of x, if such a value exists according to the inherent ordering of values in the type of x.

If *x* is the last value in the type of *x*, it has no successor. In this case the value returned is not within the range of the type of *x*, and any attempt to assign it to a variable of this type will cause unspecified results.

11.5.4 The Pred Function

Returns the predecessor of a value of ordinal-type.

Result Type: same as parameter (but see below)

Parameter List: **pred(*x*)**

1. *x* is an expression with a value of ordinal-type.

Pred(*x*) returns the predecessor of *x*, if such a value exists according to the inherent ordering of values in the type of *x*.

If *x* is the first value in the type of *x*, it has no predecessor. In this case the value returned is not within the range of the type of *x*, and any attempt to assign it to a variable of this type will cause unspecified results.

11.6 String Procedures and Functions

The string procedures and functions do not accept packed array of char parameters, and they do not accept indexed string parameters.

11.6.1 The Length Function

Returns the current length of a value of string-type.

Result Type: integer

Parameter List: **length(*str*)**

1. *str* is an expression with a value of string-type.

Length(*str*) returns the current length of *str*.

11.6.2 The Pos Function

Searches a string for the first occurrence of a specified substring.

Result Type: integer

Parameter List: **pos(*substr*, *str*)**

1. *substr* is an expression with a value of string-type.
2. *str* is an expression with a value of string-type.

Pos(*substr*, *str*) searches for *substr* within *str*, and returns an integer value that is the index of the first character of *substr* within *str*.

If *substr* is not found, **pos(*substr*, *str*)** returns zero.

11.6.3 The Concat Function

Takes a sequence of strings and concatenates them.

Result Type: string-type

Parameter List: `concat(str1 [, str2, ... strn])`

- Each parameter is an expression with a value of string-type. Any practical number of parameters may be passed.

`Concat(str1, ..., strn)` concatenates all the parameters in the order in which they are written, and returns the concatenated string. Note that the number of characters in the result cannot exceed 255.

11.6.4 The Copy Function

Returns a substring of specified length, taken from a specified position within a string.

Result Type: string-type

Parameter List: `copy(source, index, count)`

1. `source` is an expression with a value of string-type.
2. `index` is an expression with an integer value.
3. `count` is an expression with an integer value.

`Copy(source, index, count)` returns a string containing `count` characters from `source`, beginning at `source[index]`.

11.6.5 The Delete Procedure

Deletes a substring of specified length from a specified position within the value of a string variable.

Parameter List: `delete(dest, index, count)`

1. `dest` is a variable-reference that refers to a variable of string-type. This is a variable parameter.
2. `index` is an expression with an integer value.
3. `count` is an expression with an integer value.

`Delete(dest, index, count)` removes `count` characters from the value of `dest`, beginning at `dest[index]`.

11.6.6 The Insert Procedure

Inserts a substring into the value of a string variable, at a specified position.

Parameter List: `insert(source, dest, index)`

1. `source` is an expression with a value of string-type.

2. **dest** is a variable-reference that refers to a variable of string-type. This is a variable parameter.
3. **index** is an expression with an integer value.

Insert(source, dest, index) inserts **source** into **dest**. The first character of **source** becomes **dest[index]**.

11.7 Byte-Oriented Procedures and Functions

These features allow a program to treat a program variable as a sequence of bytes, without regard to data types.

NOTE

The **sizeof** function can be used to determine the number of bytes in a variable.

These procedures do no type-checking on their **source** or **dest** actual-parameters. However, since these are variable parameters they *cannot be Indexed* if they are packed or if they are of string-type. If an unpacked "byte array" is desired, then a variable of the type

array [lo..hi] of -128..127

should be used for **source** or **dest**. The elements in an array of this type are stored in contiguous bytes, and, since it is unpacked, an array of this type can be used with an index as an actual-parameter for these routines.

IMPLEMENTATION NOTE

Currently, an array with elements of the type 0..255 or the type **char** has its elements stored in words, not bytes.

11.7.1 The Moveleft Procedure

Copies a specified number of contiguous bytes from a source range to a destination range (starting at the lowest address).

Parameter List: **moveleft(source, dest, count)**

1. **source** is a variable-reference that refers to a variable of any type except a file-type or a structured-type that contains a file-type. This is a variable parameter. The first byte allocated to **source** (lowest address within **source**) is the first byte of the source range.
2. **dest** is a variable-reference that refers to a variable of any type except a file-type or a structured-type that contains a file-type. This is a variable parameter. The first byte allocated to **dest** (lowest address within **dest**) is the first byte of the destination range.

3. **count** is an expression with an integer value. The source range and the destination range are each **count** bytes long.

Movelf(*source, dest, count*) copies **count** bytes from the source range to the destination range.

Movelf starts from the "left" end of the source range (lowest address). It proceeds to the "right" (higher addresses), copying bytes into the destination range, starting at the lowest address of the destination range.

The **count** parameter is not range-checked.

11.7.2 The Moveright Procedure

Moveright is exactly like **movelf** (see above), except that it starts from the "right" end of the source range (highest address). It proceeds to the "left" (lower addresses), copying bytes into the destination range, starting at the highest address of the destination range.

The reason for having both **movelf** and **moveright** is that the **source** and **destination** ranges may overlap. If they overlap, the order in which bytes are moved is critical: each byte must be moved before it gets overwritten by another byte.

11.7.3 The Sizeof Function

Returns the number of bytes occupied by a specified variable, or by any variable of a specified type.

Result Type: integer

Parameter List: **sizeof**(*id*)

1. **id** is either a variable-identifier or a type-identifier. It must not refer to a file-type or a structured-type that contains a file-type, or to a variable of such a type.

sizeof(*id*) returns the number of bytes occupied by **id**, if **id** is a variable-identifier; if **id** is a type-identifier, it returns the number of bytes occupied by any variable of type **id**.

11.8 Packed Array of Char Procedures and Functions

NOTE

These routines operate only on **packed arrays of char**. The **packed arrays of char** cannot be subscripted; the operations always begin at the first character in a **packed array of char**.

11.8.1 The Scaneq Function

Searches a packed array of char for the first occurrence of a specified character.

Result Type: integer

Parameter List: scaneq(limit, ch, paoc)

1. **limit** is an expression with a value of type **integer** or **longint**. It is truncated to 16 bits, and is not range-checked.
2. **ch** is an expression with a value of type **char**.
3. **paoc** is an expression with a value of type **packed array of char**. This is a variable parameter.

Scaneq(limit, ch, paoc) scans **paoc**, looking for the first occurrence of **ch**. The scan begins with the first character in **paoc**. If the character is not found within **limit** characters from the beginning of **paoc**, the value returned is equal to **limit**. Otherwise, the value returned is the number of characters scanned before **ch** was found.

11.8.2 The Scanne Function

This function is exactly like **scaneq**, except that it searches for a character that does *not* match the **ch** parameter.

11.8.3 The Fillchar Procedure

Fills a specified number of characters in a packed array of char with a specified character.

Parameter List: fillchar(paoc, count, ch)

1. **paoc** is an expression with a value of type **packed array of char**. This is a variable parameter.
2. **count** is an expression with a value of type **integer** or **longint**. It is truncated to 16 bits, and is not range-checked.
3. **ch** is an expression with a value of type **char**.

Fillchar(paoc, count, ch) writes the value of **ch** into **count** contiguous bytes of memory, starting at the first byte of **paoc**.

Since the **count** parameter is not range-checked, it is possible to write into memory outside of **paoc**, with unspecified results.

Chapter 12 THE COMPILER

12.1	Compiler Commands	12-3
12.2	Conditional Compilation	12-4
12.2.1	Compile-Time Variables and the \$DECL Command	12-4
12.2.2	The \$SETC Command	12-5
12.2.3	Compile-Time Expressions	12-5
12.2.4	The \$IFC, \$ELSEC, and \$ENDC Commands	12-6
12.3	Optimization of If-Statements	12-7
12.4	Optimization of While-Statements and Repeat-Statements	12-8
12.5	Efficiency of Case-Statements	12-8

THE COMPILER

The Pascal compiler translates Pascal source text to an intermediate code, and the code generator translates the intermediate code to MC68000 object code. Instructions for operating the compiler and code generator are given in the *Workshop Reference Manual for the Lisa*.

12.1 Compiler Commands

A compiler command is a text construction, embedded in source text, that controls compiler operation. Every compiler command is written within comment delimiters, {...} or (*...*). Every compiler command begins with the \$ character, which must be the first character inside the comment delimiters.

In this manual, compiler commands are shown in upper case to help distinguish them from Pascal program text; however, upper and lower case are interchangeable in compiler commands just as they are in Pascal program text.

The following compiler commands are available:

Input File Control

- | | |
|---------------------|---|
| \$I filename | Start taking source code from file filename . When the end of this file is reached, revert to the previous source file. The filename cannot begin with + or -. |
| \$U filename | Search the file filename for any units subsequently specified in the uses-clause. Does not apply to intrinsic-units. |

Control of Code Generation

- | | |
|---------------------|---|
| \$C+ or \$C- | Turn code generation on (+) or off (-). This is done on a procedure-by-procedure basis. These commands should be written between procedures; results are unspecified if they are written inside procedures. The default is \$C+ . |
| \$R+ or \$R- | Turn range checking on (+) or off (-). At present, range checking is done in assignment statements and array indexes and for string value parameters. The default is \$R+ . |
| \$S segname | Start putting code modules into segment segname . The default segment name is a string of blanks to designate the "blank segment," in which the main program and all built-in support code are always linked. All other code can be placed into any segment. |
| \$X+ or \$X- | Turn automatic run-time stack expansion on (+) or off (-). The default is \$X+ . |

Debugging

\$D+ or \$D- Turn the generation of procedure names in object code on (+) or off (-). These commands should be written between procedures; results are unspecified if they are written inside procedures. The default is **\$D+**.

Conditional Compilation

\$DECL list (see Section 12.2 below).
\$ELSEC (see Section 12.2 below).
\$ENDC (see Section 12.2 below).
\$IFC (see Section 12.2 below).
\$SETC (see Section 12.2 below).

Listing Control

\$E filename Start making a listing of compiler errors as they are encountered. Analogous to **\$L filename** (see below). The default is no error listing.

\$L filename Start listing the compilation on file **filename**. If a listing is being made already, that file is closed and saved prior to opening the new file. The default is no listing.

\$L+ or \$L- The first + or - following the **\$L** turns the source listing on (+) or off (-) without changing the list file. You must specify the listing file before using **\$L+**. The default is **\$L+**, but no listing is produced if no listing file has been specified.

12.2 Conditional Compilation

Conditional compilation is controlled by the **\$IFC**, **\$ELSEC**, and **\$ENDC** commands, which are used to bracket sections of source text. Whether a particular bracketed section of a program is compiled depends on the **boolean** value of a **compile-time expression**, which can contain **compile-time variables**.

12.2.1 Compile-Time Variables and the \$DECL Command

Compile-time variables are completely independent of program variables; even if a compile-time variable and a program variable have the same identifier, they can never be confused by the compiler.

A compile-time variable is declared when it appears in the identifier-list of a **\$DECL** command.

Example of compile-time variable declaration:

```
{ $DECL LIBVERSION, PROGVERSION }
```

This declares `LIBVERSION` and `PROGVERSION` as compile-time variables. Notice that no types are specified.

Note the following points about compile-time variables:

- Compile-time variables have no types, although their values do. The only possible types are `integer` and `boolean`.
- All compile-time variables must be declared before the end of the variable-declaration-part of the main program. In other words a `$DECL` command that declares a new compile-time variable must precede the main program's procedure and function declarations (if any). The new compile-time variable is then known throughout the remainder of the compilation.
- At any point in the program, a compile-time variable can have a new value assigned to it by a `$SETC` command.

12.2.2 The `$SETC` Command

The `$SETC` command has the form

```
{$SETC ID := EXPR}
```

where `ID` is the identifier of a compile-time variable and `EXPR` is a compile-time expression. `EXPR` is evaluated immediately. The value of `EXPR` is assigned to `ID`.

Example of assignment to compile-time variable:

```
{$SETC LIBVERSION := 5}
```

This assigns the value 5 to the compile-time variable `LIBVERSION`.

12.2.3 Compile-Time Expressions

Compile-time expressions appear in the `$SETC` command and in the `$IFC` command. A compile-time expression is evaluated by the compiler as soon as it is encountered in the text.

The only operands allowed in a compile-time expression are:

- Compile-time variables
- Constants of the types `integer` and `boolean`. (These are also the only possible types for results of compile-time expressions.)

All Pascal operators are allowed except as follows:

- The `in` operator is not allowed.
- The `@` operator is not allowed.
- The `/` operator is automatically replaced by `div`.

12.2.4 The \$IFC, \$ELSEC, and \$ENDC Commands

The \$ELSEC and \$ENDC commands take no arguments. The \$IFC command has the form

```
{$IFC EXPR}
```

where EXPR is a compile-time expression with a **boolean** value.

These three commands form constructions similar to the Pascal if-statement, except that the \$ENDC command is always needed at the end of the \$IFC construction. \$ELSEC is optional.

Example of conditionally compiled code:

```
{$IFC PROGVERSION >= LIBVERSION}
  k := kval1(data+indat);
{$ELSEC}
  k := kval2(data+cpindat^);
{$ENDC}
  writeln(k)
```

If the value of PROGVERSION is greater than or equal to the value of LIBVERSION, then the statement `k:=kval1(data+indat)` is compiled, and the statement `k:=kval2(data+cpindat^)` is skipped.

But if the value of PROGVERSION is less than the value of LIBVERSION, then the first statement is skipped and the second statement is compiled.

In either case, the `writeln(k)` statement is compiled because the conditional construction ends with the \$ENDC command.

\$IFC constructions can be nested within each other to 10 levels. Every \$IFC must have a matching \$ENDC.

When the compiler is skipping, all commands in the skipped text are ignored except the following:

```
$ELSEC
$ENDC
$I      (so that $ENDC's can be matched properly)
        (text from another file is scanned even if it is being
        skipped, in case it contains $ELSEC, $ENDC, or $IFC
        commands).
```

All program text is ignored during skipping. If a listing is produced, each source line that is skipped is marked with the letter S as its "lex level."

12.3 Optimization of If-Statements

When the compiler finds an if-statement controlled by a **boolean** constant, it may be unnecessary to compile the **then** part or the **else** part. For example, given the declarations

```
const always = true;
      never = false;
```

then the statement

```
if never then statement
```

will not be compiled at all. In the statement

```
if never then statement1
   else statement2
```

"statement1" is not compiled; only "statement2" is compiled. Similarly, in the statement

```
if always then statement1
   else statement2
```

only "statement1" is compiled.

The interaction between this optimization and conditional compilation can be seen from the following program:

```
program Foo;
  {$SETC FLAG := FALSE}
  const pi = 3.1415926;
        size = 512;
  {$IFC FLAG}
    debug = false; {a boolean constant, if FLAG=true}
  {$ENDC}
  var i, j, k, l, m, n: integer;
  {$IFC NOT FLAG}
    debug: boolean; {a boolean variable, if FLAG=false}
  {$ENDC}
  ...
  {$IFC NOT FLAG}
  procedure whatmode;
  begin
    {interactive procedure to set global boolean variable,
     debug}
  end;
  {$ENDC}
```



```
...
begin {main}
  {$IFC NOT FLAG}
    whatmode;
  {$ENDC}

  ...
  if debug then begin
    statement1
  end
  else begin
    statement2
  end

  ...
end.
```

The way this is compiled depends on the compile-time variable FLAG. If FLAG is false, then `debug` is a *boolean variable* and the `whatmode` procedure is compiled and called at the beginning of the main program. The `if debug` statement is controlled by a *boolean variable* and all of it is compiled, in the usual manner.

But if the value of FLAG is changed to *true*, then `debug` is a *constant* with the value *false*, and `whatmode` is neither compiled nor called. The `if debug` statement is controlled by a constant, so only its *else* part, "statement2", is compiled.

12.4 Optimization of While-Statements and Repeat-Statements

A while-statement or repeat-statement controlled by a *boolean constant* does not generate any conditional branches.

12.5 Efficiency of Case-Statements

A sparse or small case-statement will generate smaller and faster code than the corresponding sequence of *if*-statements.

Appendix A
COMPARISON TO APPLE II AND APPLE III
PASCAL

Extensions	A-3
Deletions	A-3
Other Differences	A-4

COMPARISON TO APPLE II AND APPLE III PASCAL

This appendix contains lists of the major differences between the Pascal language on the Lisa and the Pascal implemented on the Apple II and Apple III. Please note that these lists are not exhaustive.

Extensions

The following features have been added on the Lisa:

- **@ Operator** — returns the pointer to its operand (see Section 5.1.6)
- **Heapresult**, **pointer**, and **ord4** functions (see Sections 11.2.2, 11.3.3, and 11.3.4)
- **Keypress** function built into the language, with same effect as the **keypress** function in the **applestuff** unit of Apple II and Apple III Pascal (see Section 10.3.7.1)
- Hexadecimal constants (see Section 1.4)
- **Otherwise**-clause in case-statement (same as Apple III Pascal; see Section 6.2.2.2)
- Global **goto**-statement (see Section 6.1.3)
- A **file** of **char** type that is distinct from the **text** type (see Sections 3.2.4 and 10.3).
- Numerous compiler commands (see Section 12.1)
- Procedural and functional parameters (see Sections 7.3.3 and 7.3.4)
- Stronger type-checking (see Sections 3.4 and 7.3.5)

Deletions

The following features are not included on the Lisa:

- **Turtlegraphics**, **applestuff**, and other standard units of Apple II and Apple III Pascal.
- **Interactive** type (not needed, as the I/O procedures will do the right thing with a file of type **text** if it is opened on a character device).
- **Keyboard** file — same effect can be obtained by opening a file of type **text** on the device **-KEYBOARD** (see Section 10.3).
- Unit (device-oriented) I/O procedures.
- Recognition of the **ETX** character (control-C) to mean "end of file" in input from a character device.

- "Long Integer" data type, with length attribute in declaration. Replaced by the `longint` type (see Section 3.1.1.2)
- "Initialization" code in a unit (see Section 9).
- The ability to create new intrinsic-units and install them in the system. (See Section 9.)
- Reset procedure without an external file title, for use on a file that is already open (see Section 10.1.1). To obtain the same effect, close the file and reopen it.
- Treearch.
- `Bytestream`, `wordstream` (data types in Apple III Pascal)
- `Exit(program)` — The `exit(identifier)` form works, and the identifier can be the program-identifier. `Halt` can also be used for orderly exit from a program (see Section 11.1).
- Extended comparisons (See Section 5.1.5.)
- `Scan` function. Replaced by `scaneq` and `scanne` (see Section 11.8)
- Bit-wise `boolean` operations
- `Segment` keyword for procedures and functions. Use the `$$` command instead. (See Section 12.1.)
- The following compiler commands (see Section 12.1):
 - `$I+` and `$I-` (no automatic I/O checking; program must use `ioresult` function)
 - `$G` (`$G+` is the assumption on the Lisa)
 - `$N` and `$R` (for resident code segments)
 - `$P`
 - `$Q`
 - `$S+` and `$S++` for swapping
 - `$U+` and `$U-` (for User Program)
 - `$V`

In general, do not assume that a compiler command used in Apple II or Apple III Pascal is valid on the Lisa. Furthermore, do not assume that an Apple II or Apple III Pascal compiler command is "harmless" on the Lisa, as it may be implemented with a different meaning.

Other Differences

The following features of Pascal on the Lisa are different from the corresponding features of Apple II and Apple III Pascal:

- Size of all strings must be explicitly declared (see Section 3.1.1.6).
- **Mod** and **div** — Pascal on the Lisa truncates toward 0 (see Section 5.1.2).
- Apple II and Apple III Pascal ignore underscores; Pascal on the Lisa does not. They are legal characters in identifiers (see Section 1.2).
- A **goto**-statement cannot refer to a case-constant in Pascal on the Lisa (see Section 6.1.3).
- A program must begin with the word **program** in Pascal on the Lisa (see Chapter 8).
- **Trunc** is different (see Section 11.3.1).
- **Write(b)** where **b** is a **boolean** will write either 'TRUE' or 'FALSE' in Pascal on the Lisa (see Section 10.3.3).
- Whether a file is a textfile does not depend on whether its name ends with ".TEXT" when it is created. Instead, any external file opened with a file variable of type **text** is treated as a textfile, while a file opened with a file variable of type **file of char** is not; it is treated as a "datafile," i.e. a straight file of records which are of type **char** (see Sections 3.2.4 and 10.2).
- **Get**, **put**, and the contents of the file buffer variable are not supported on files of type **text**. Use only the text-oriented I/O procedures with textfiles.
- **Eoln** and **eof** functions on files of type **text** work as they do on **interactive** files in Apple II and Apple III Pascal.
- Pascal on the Lisa does not let you pass an element of a packed variable as a variable parameter (see Sections 7.3.2, 11.7, and 11.8).
- Limits on sets are different (see Section 3.2.3)
- The control variable of a **for**-statement must be a local variable (see Section 6.2.3.3)
- In a **write** or **writeln** call, the default field lengths for **integer** and **real** values are 8 and 12 respectively (see Section 10.3.3)

Appendix B

KNOWN ANOMALIES IN THE COMPILER

Scope of Declared Constants	B-3
Scope of Base-Types for Pointers	B-4

KNOWN ANOMALIES IN THE COMPILER

This appendix describes the known anomalies in the current implementation of the compiler.

Scope of Declared Constants

Consider the following program:

```
program cscope1;
  const ten=10;

  procedure p;
    const ten=ten; {THIS SHOULD BE AN ERROR}
  begin
    writeln(ten)
  end;

begin
  p
end.
```

The constant declaration in procedure *p* should cause a compiler error, because it is illegal to use an identifier within its own declaration (except for pointer identifiers). However, the error is not detected by the compiler. The effect is that the value of the global constant *ten* is used in defining the local constant *ten*, and the *writeln* statement writes "10".

A more serious anomaly of the same kind is illustrated by the following program:

```
program cscope2;
  const red=1; violet=2;

  procedure q;
    type arrayType=array[red..violet] of integer;
    color=(violet,blue,green,yellow,orange,red);
    var arrayVar:arrayType; c:color;
  begin
    arrayVar[1]:=1;
    c:=red;
    writeln(ord(c))
  end;

begin
  q
end.
```

Within the procedure `q`, the global constants `red` and `violet` are used to define an array index type; the effect of `array[red..violet]` is equivalent to `array[1..2]`. In the declaration of the type `color`, the constants `red` and `violet` are locally redefined; they are no longer equal to 1 and 2 respectively -- instead they are constants of type `color` with ordinalities 5 and 0 respectively. The `writeln` statement writes "5".

The use of `red` in the declaration of the type `color` should cause a compiler error but does not.

Consider the statement

```
arrayVar[1]:=1;
```

If this statement is replaced by

```
arrayVar[red]:=1;
```

a compiler error will result, as `red` is now an illegal index value for `arrayVar` -- even though `arrayVar` is of type `arrayType` and `arrayType` is defined by `array[red..violet]`.

To avoid this kind of situation, avoid redefinition of constant-identifiers in enumerated scalar types.

Scope of Base-Types for Pointers

Consider the following program:

```
program pscope1;
  type s=0..7;
  procedure makecurrent;
    type sptr=~s;
    s=record
      ch:char;
      bool:boolean
    end;
    var current:s;
        ptrs:sptr;
    begin
      new(ptrs);
      ptrs^:=current
    end;
  begin
    makecurrent
  end.
```

Here we have a global type `s`, which is a subrange of `integer`; we also have a local type `s`, which is a record-type. Within the procedure `makecurrent`, the type `sptr` is defined as a pointer to a variable of type `s`. The intention is that

this should refer to the local type `s`, defined on the next line of the program; unfortunately, however, the compiler does not yet know about the local type `s` and uses the global type `s`. Thus `ptrs` becomes a pointer to a variable of type `0..7` instead of a pointer to a record. Consequently the statement

```
ptrs^ := current
```

causes a compiler error since `ptrs^` and `current` are of incompatible types.

To avoid this kind of situation, re-declare the type `s` locally before declaring the pointer-type `sptr` based on `s`. Alternately, avoid re-declaration of identifiers that are used as base-types for pointer-types.

Appendix C SYNTAX OF THE LANGUAGE

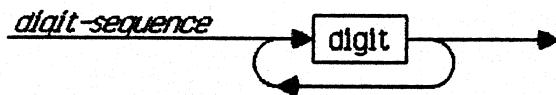
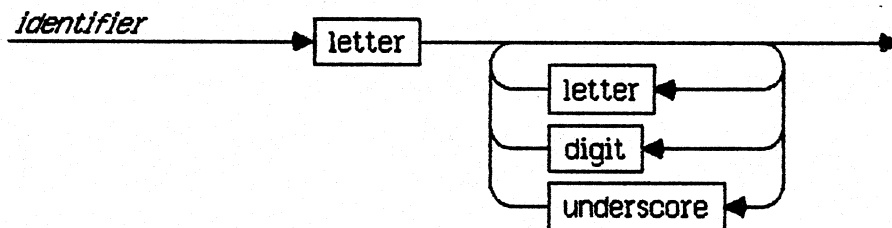
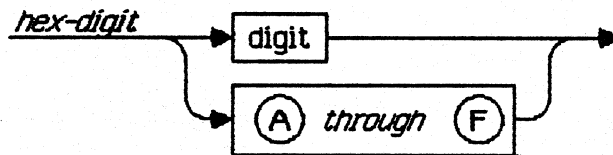
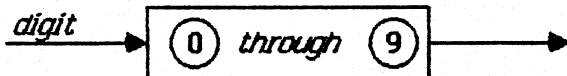
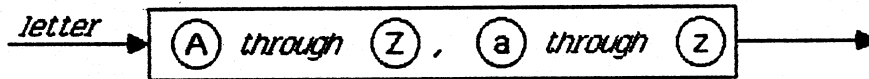
Introduction	C-3
Tokens and Constants	C-3
Blocks	C-6
Data Types	C-7
Variables	C-11
Expressions	C-12
Statements	C-14
Procedures and Functions	C-17
Programs	C-19
Units	C-20

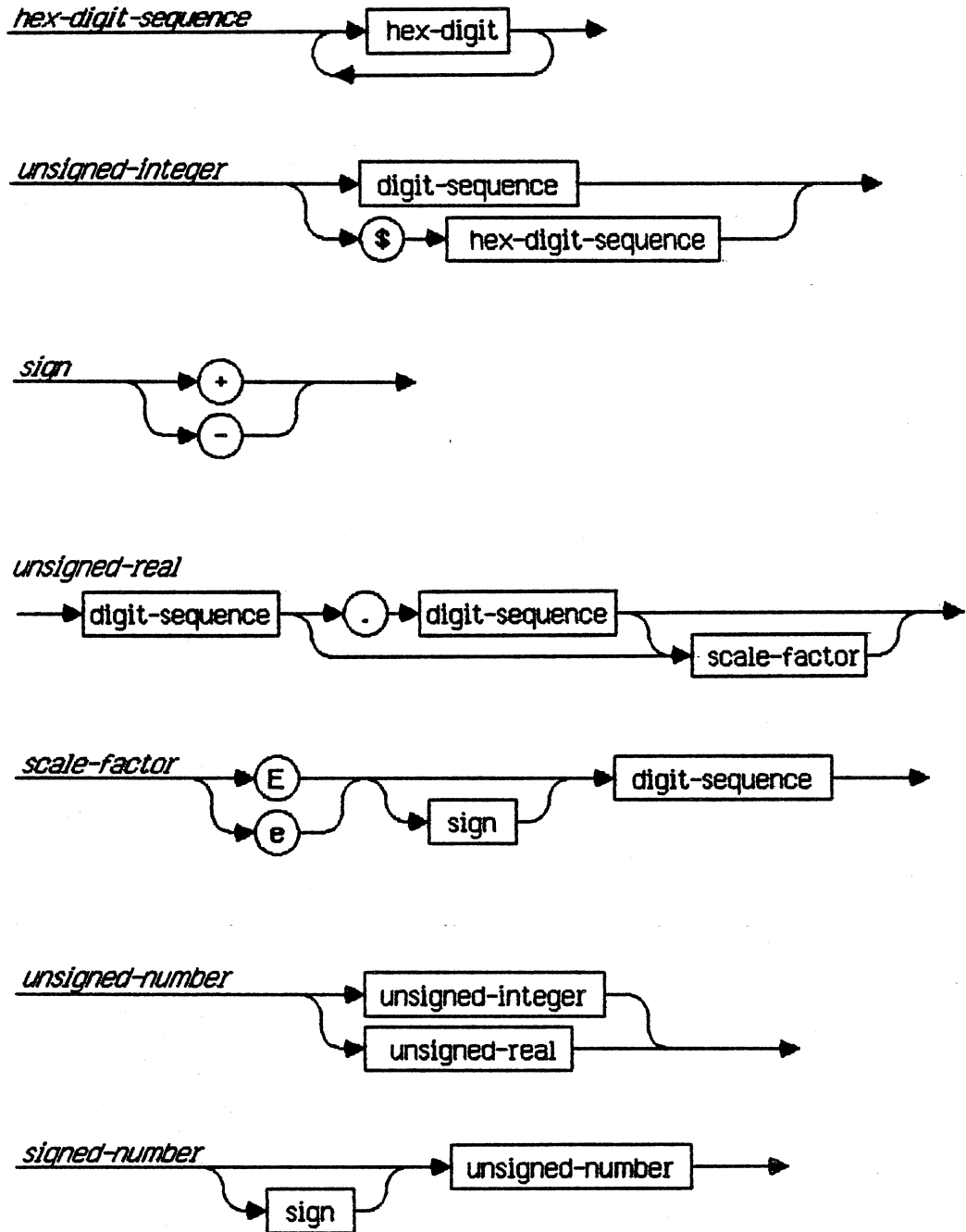
SYNTAX OF THE LANGUAGE

Introduction

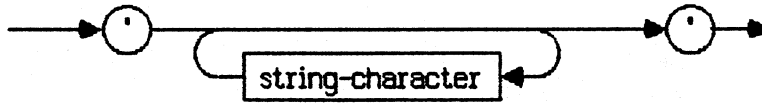
This appendix collects the syntax diagrams found in the main sections of this manual. See the Preface for an introduction to syntax diagrams.

Tokens and Constants (see Chapter 1)

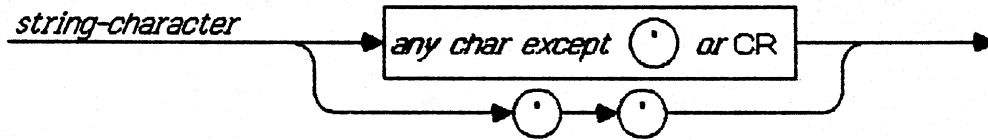




quoted-string-constant



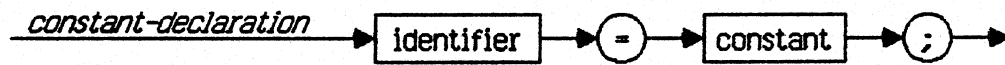
string-character



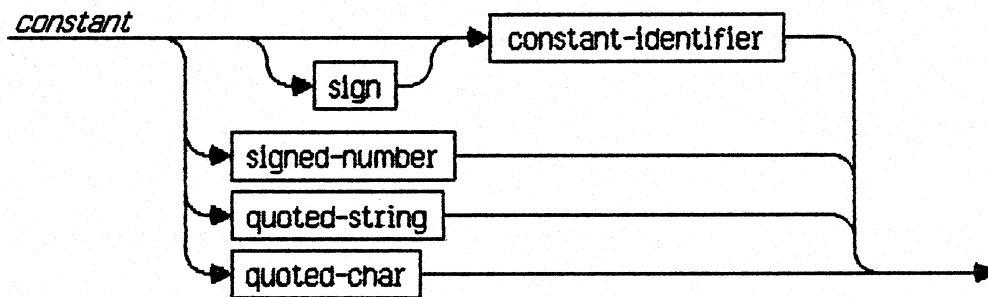
quoted-character-constant



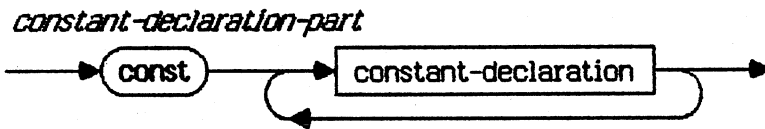
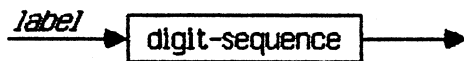
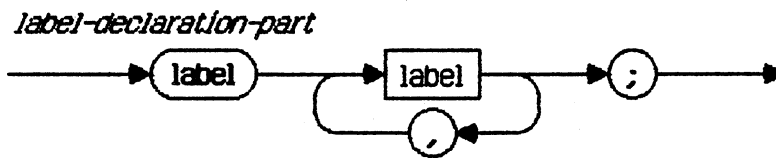
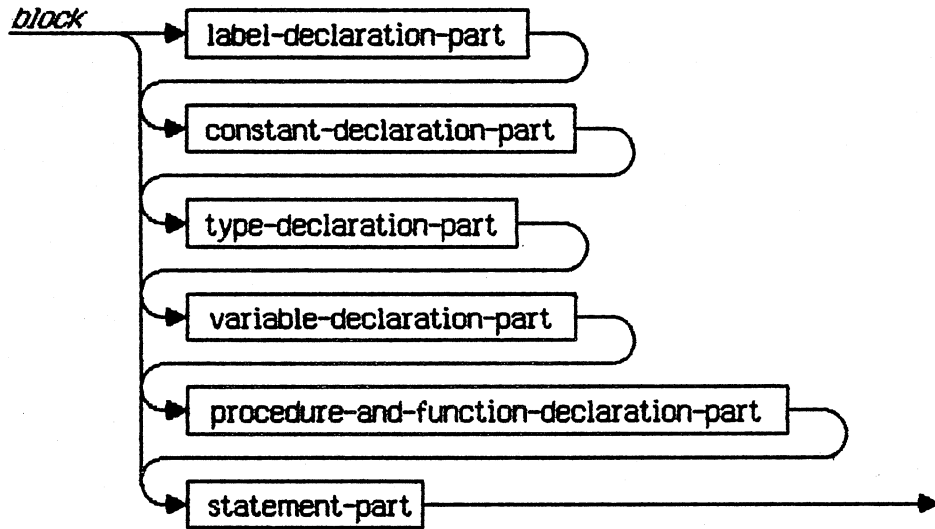
constant-declaration



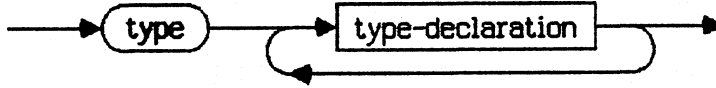
constant



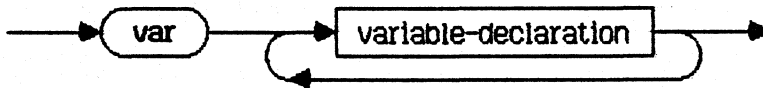
Blocks (see Chapter 2)



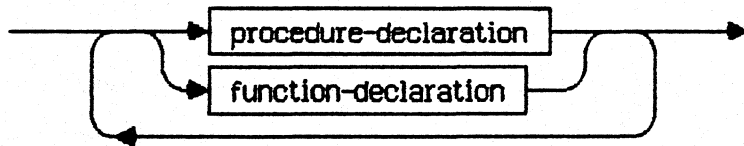
type-declaration-part



variable-declaration-part



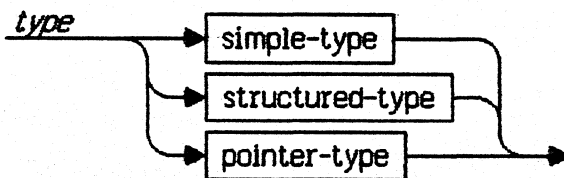
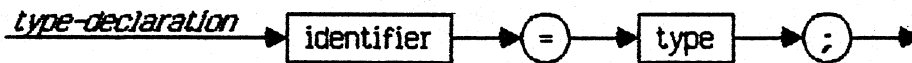
procedure-and-function-declaration-part

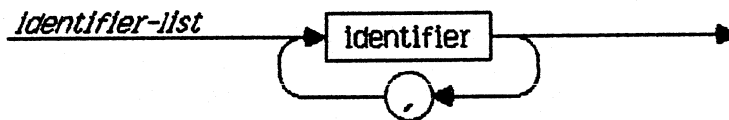
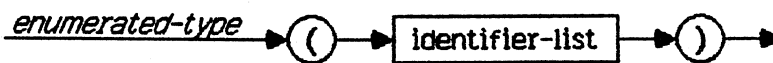
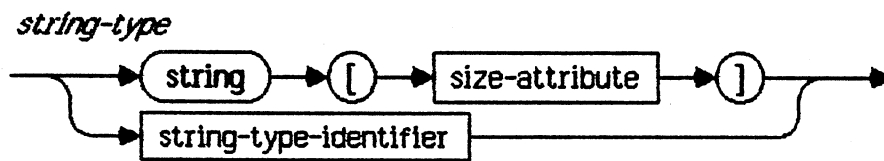
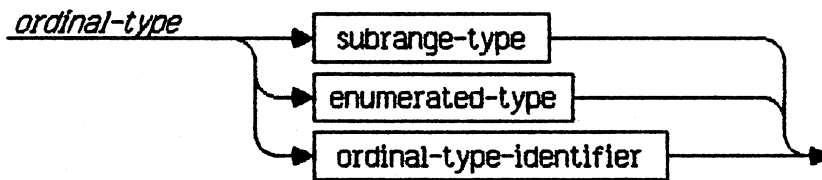
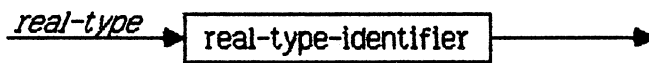
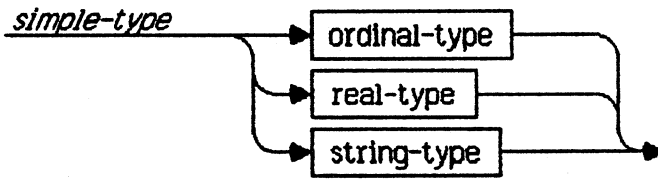


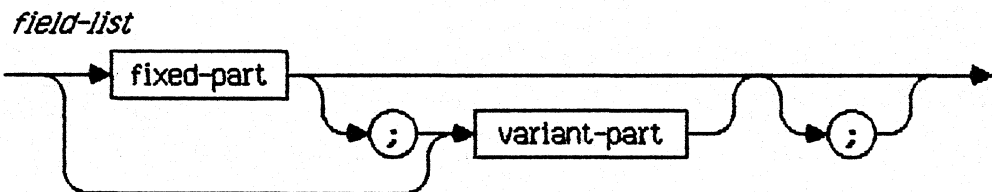
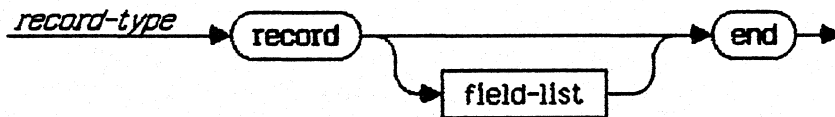
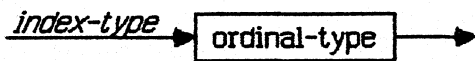
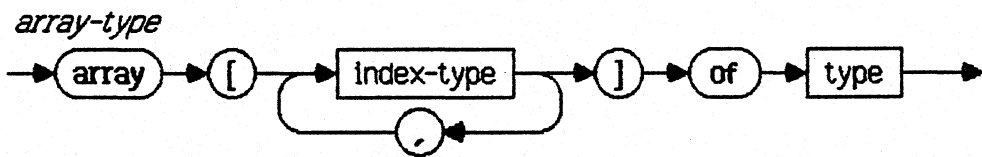
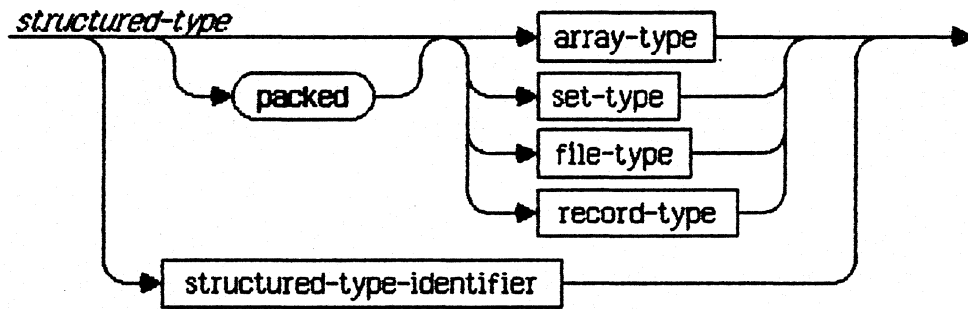
statement-part

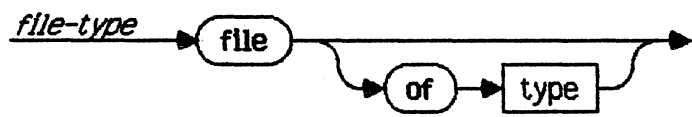
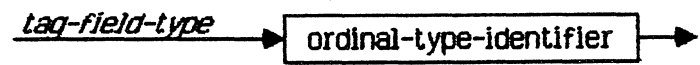
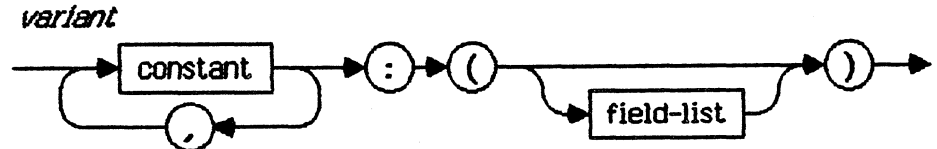
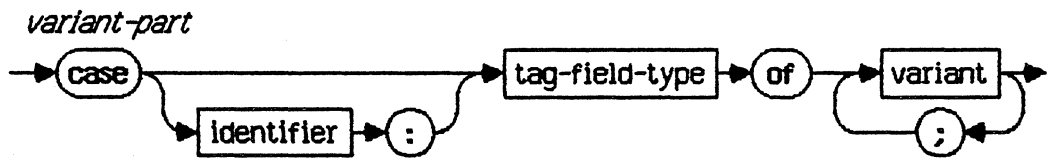
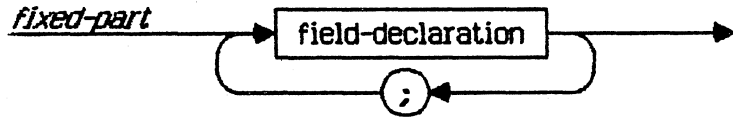


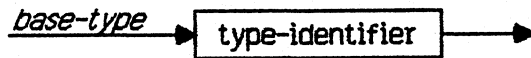
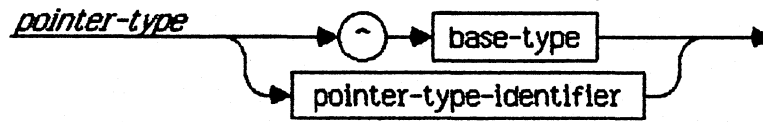
Data Types (see Chapter 3)



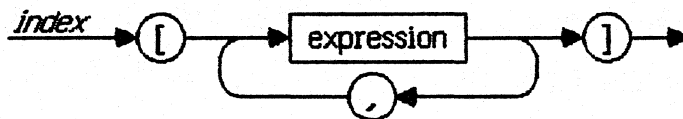
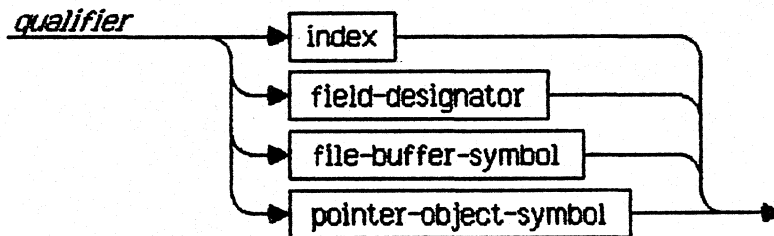
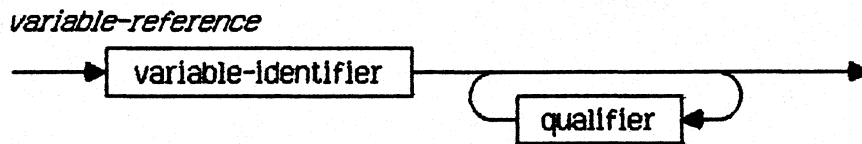
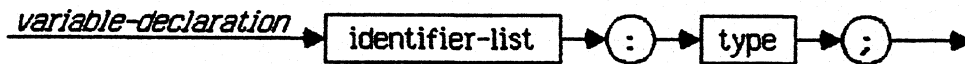


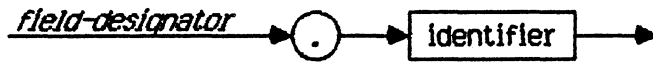




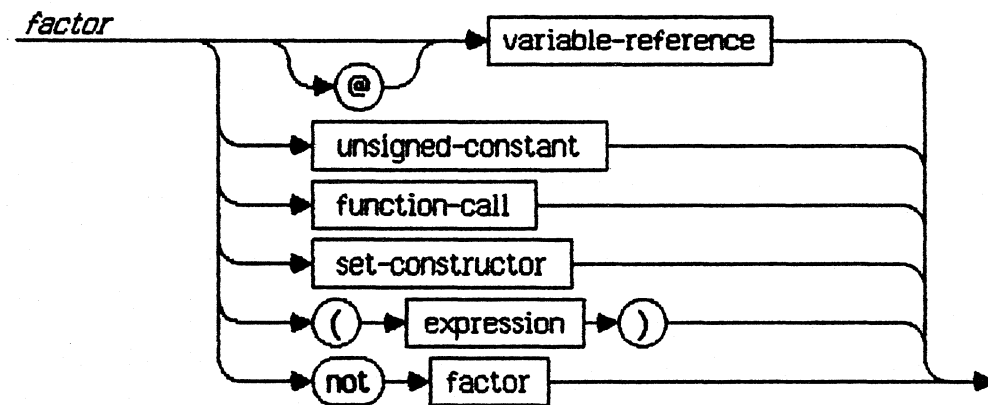
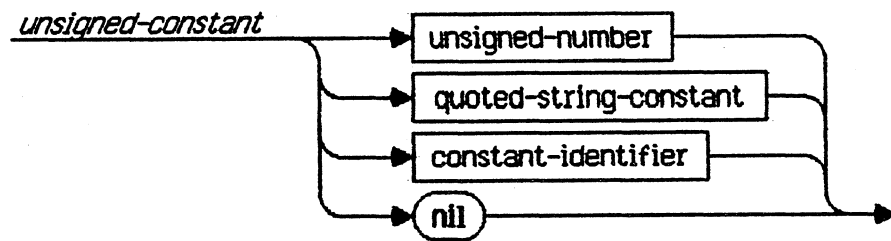


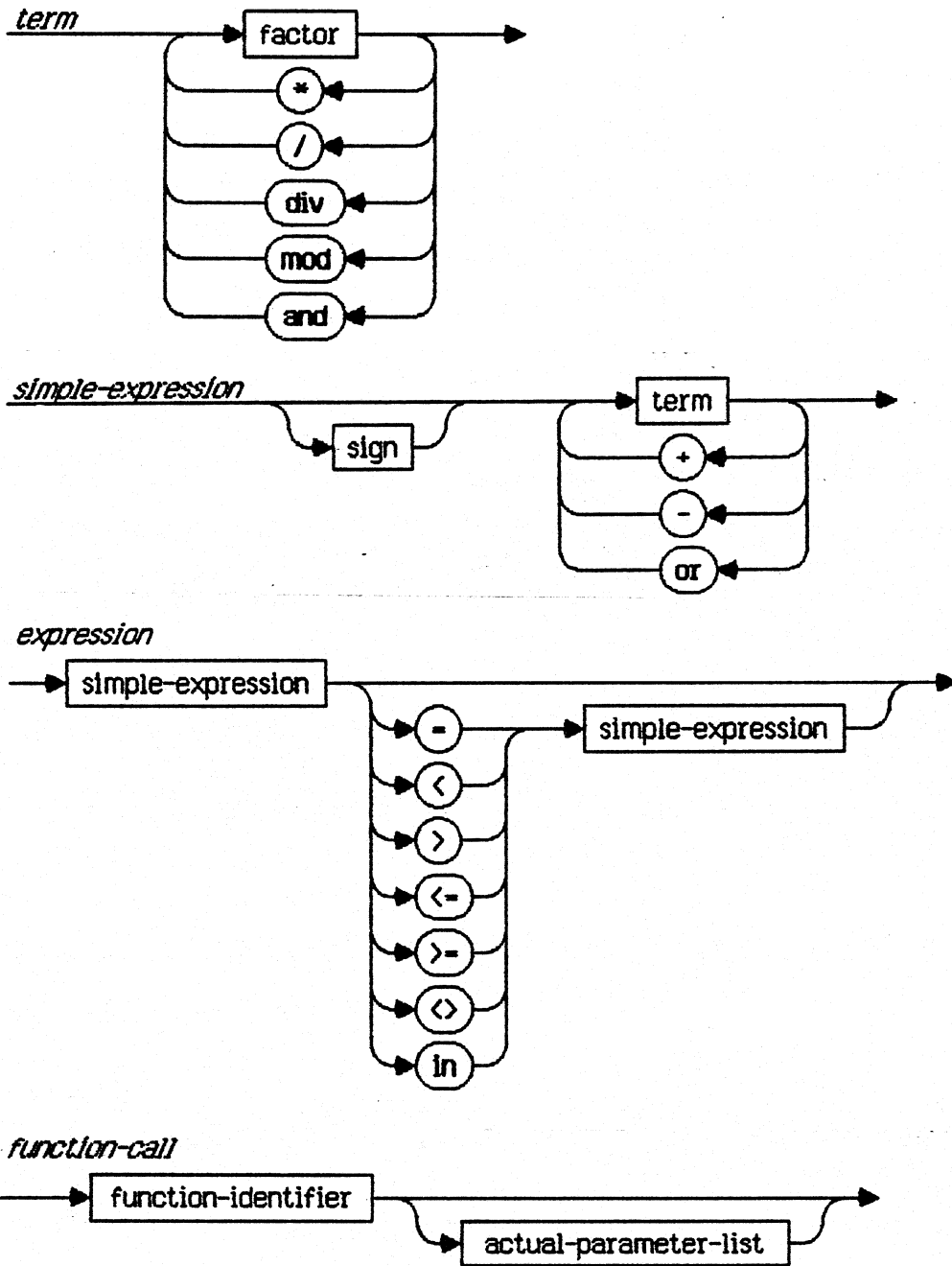
Variables (see Chapter 4)

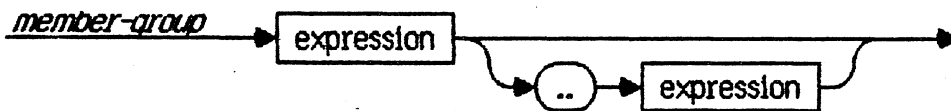
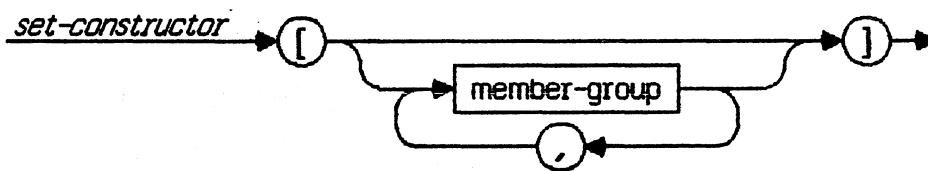
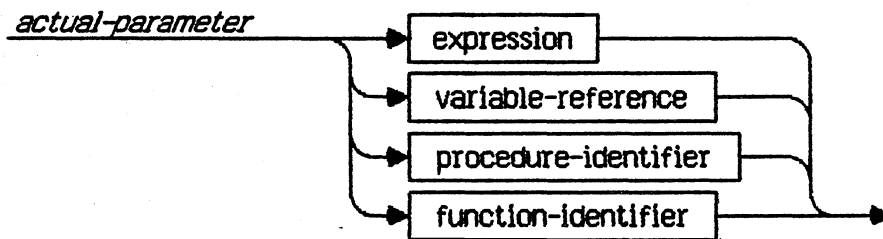
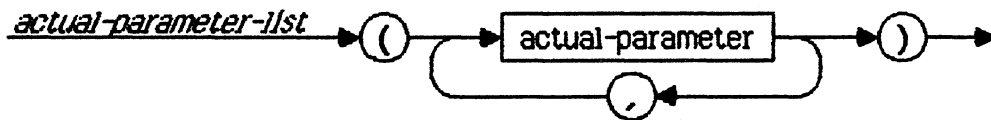




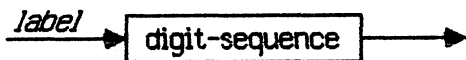
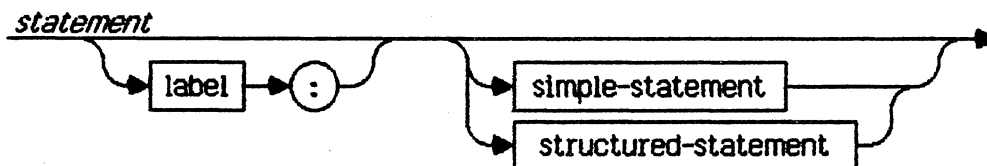
Expressions (see Chapter 5)

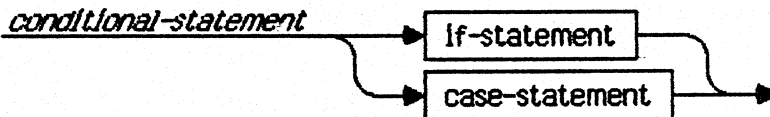
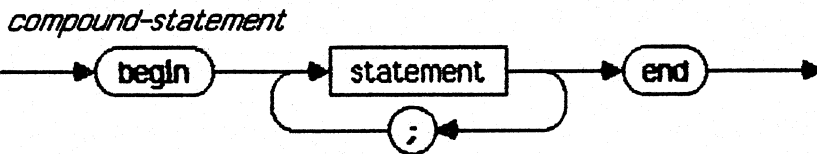
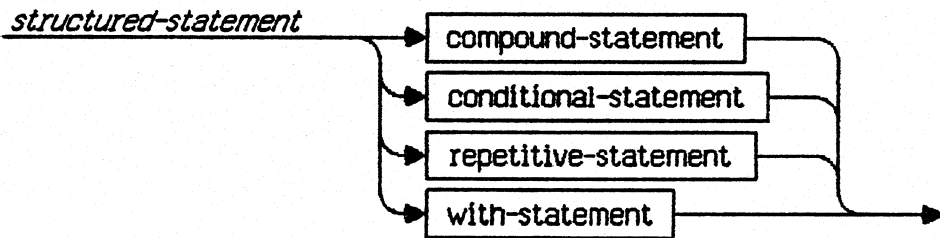
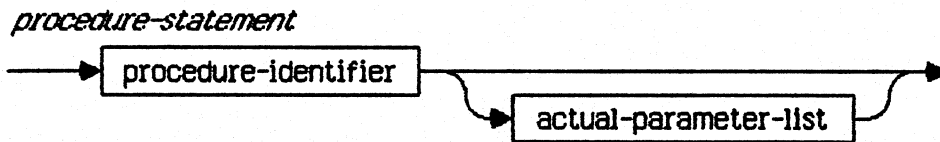
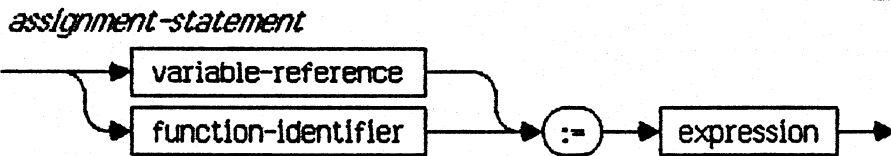
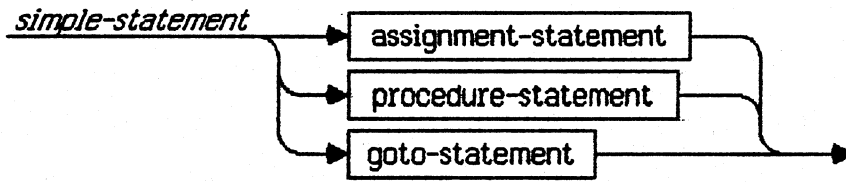


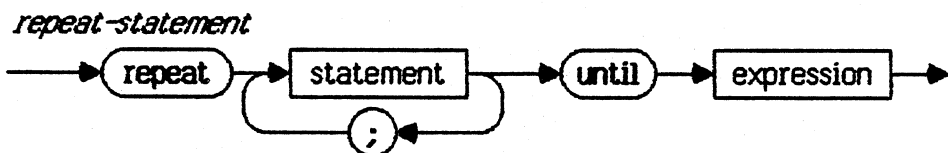
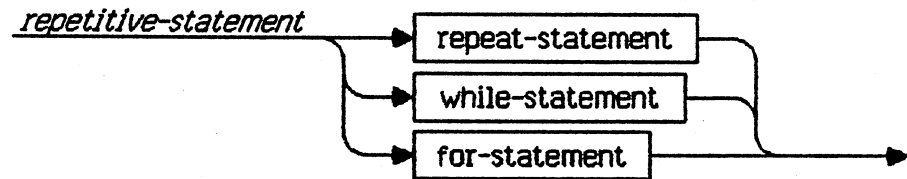
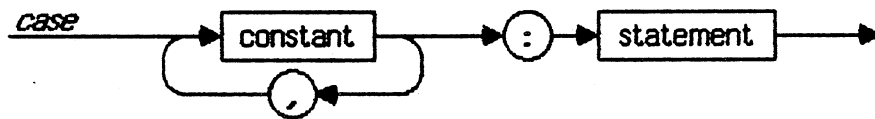
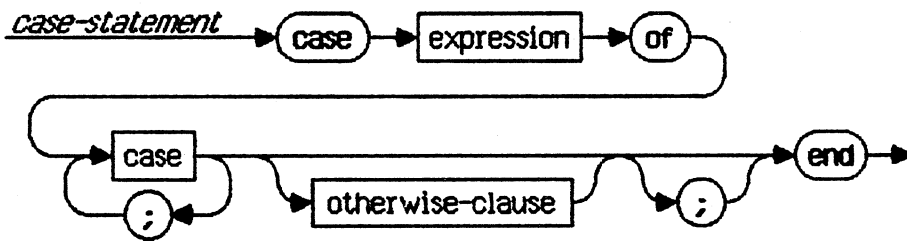
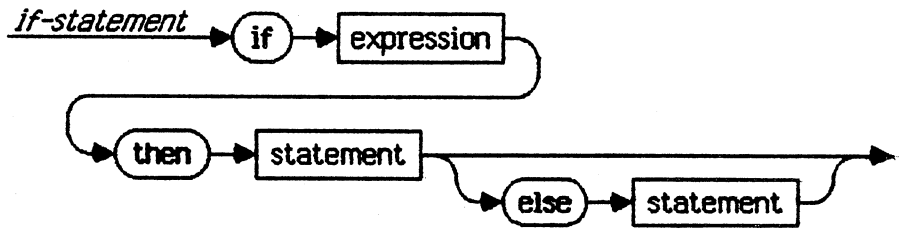


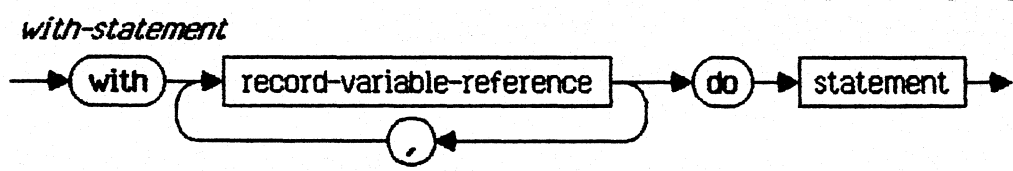
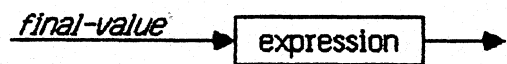
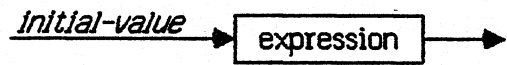
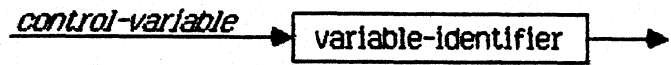
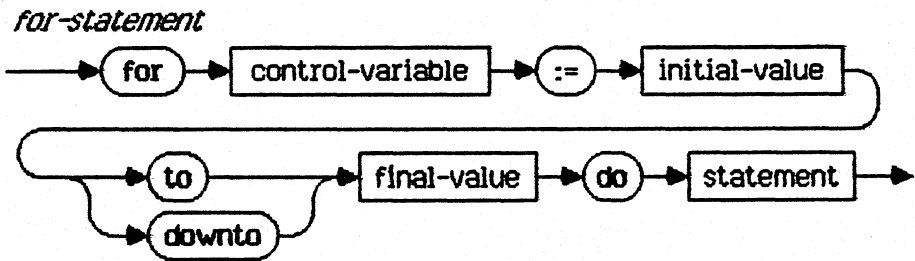
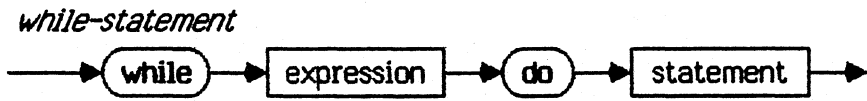


Statements (see Chapter 6)

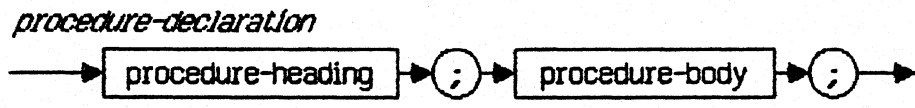


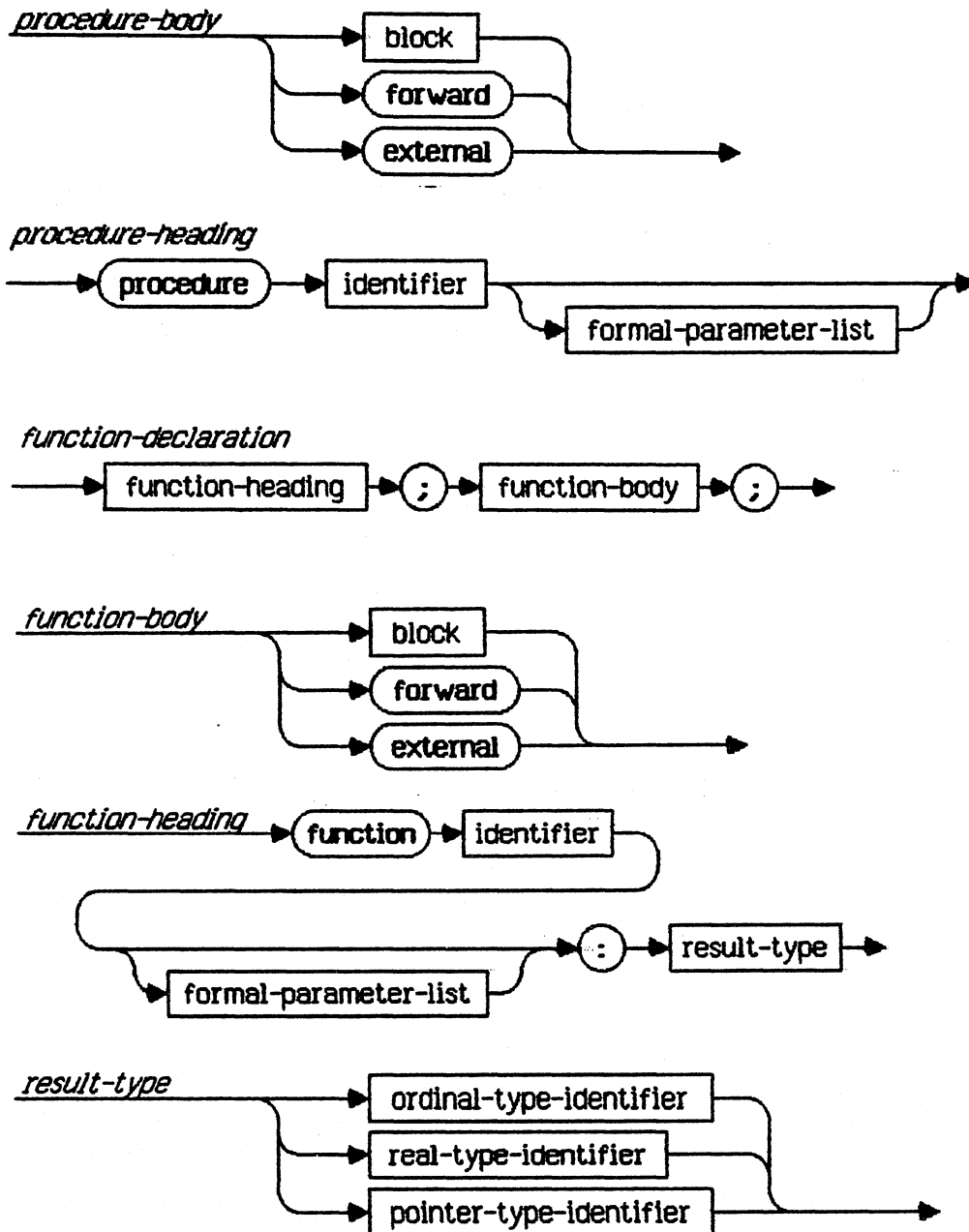


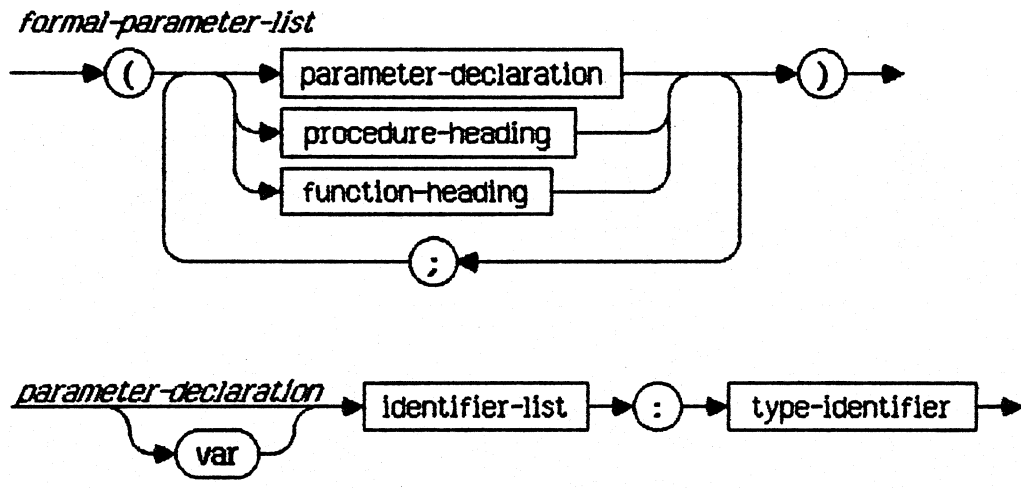




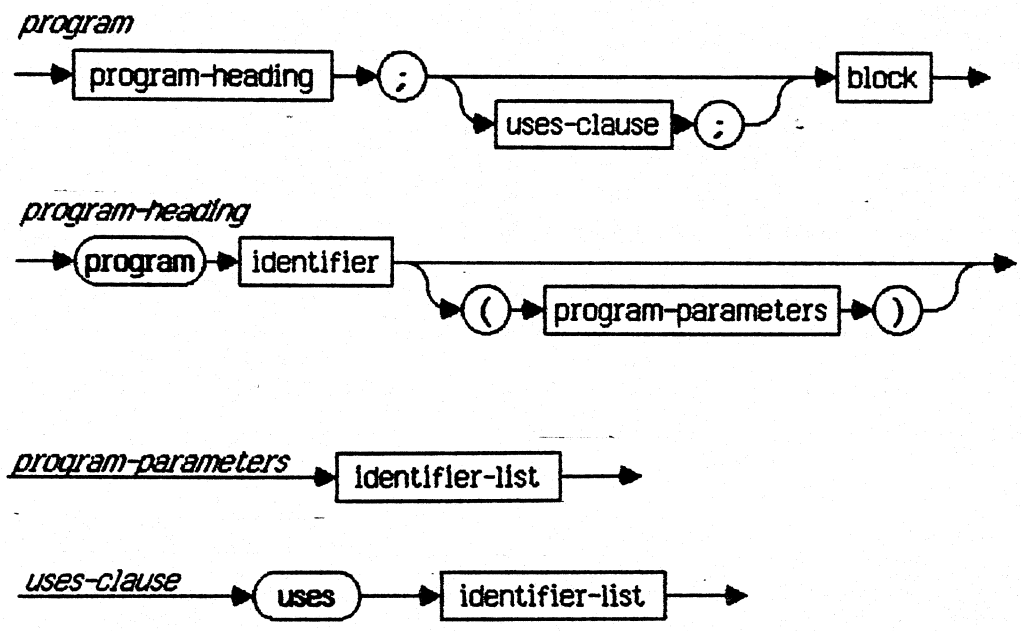
Procedures and Functions (see Chapter 7)



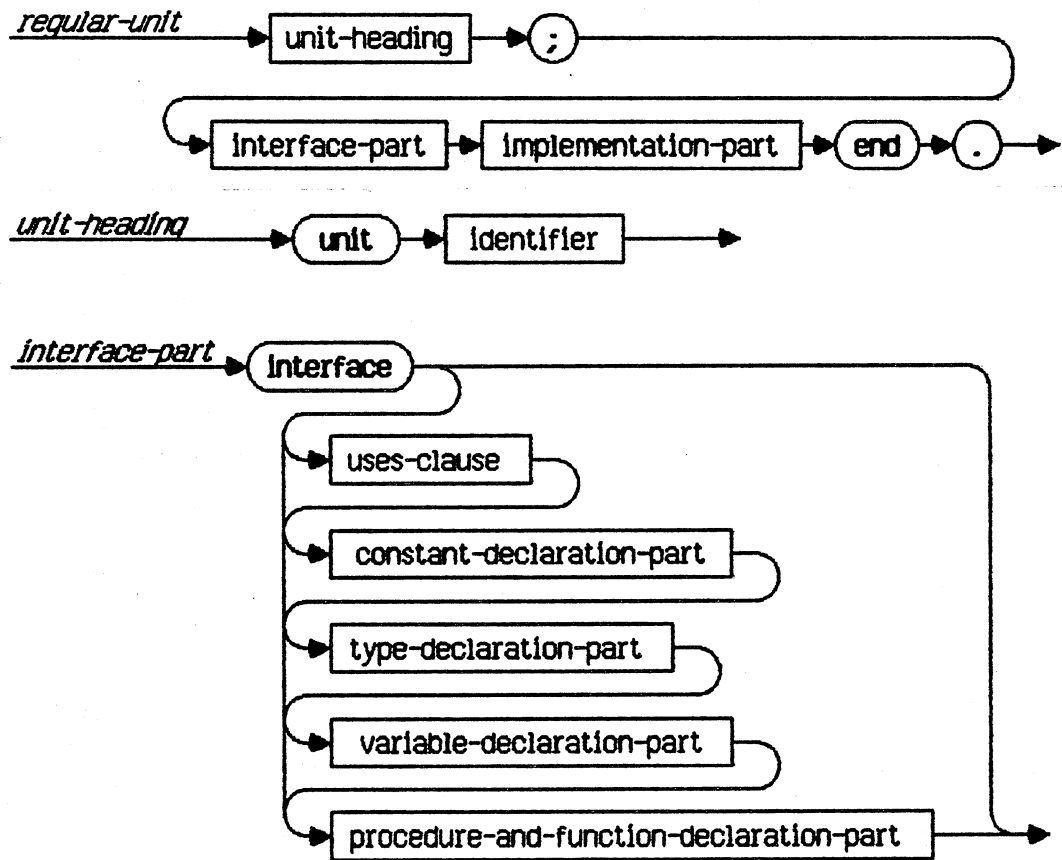


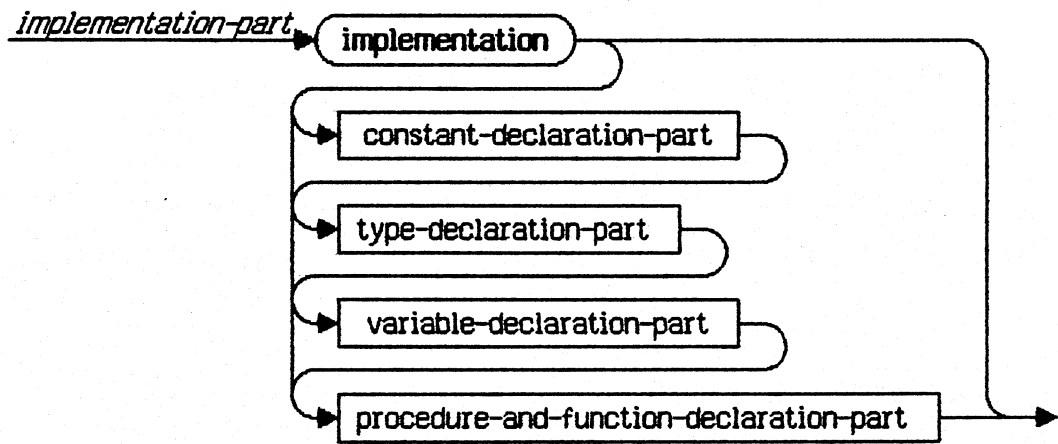


Programs (see Chapter 8)



Units (see Chapter 9)





Appendix D FLOATING-POINT ARITHMETIC

Introduction	D-3
Rounding of Real Results	D-3
Accuracy of Arithmetic Operations	D-4
Overflow and Division by Zero: Infinite Values	D-4
Invalid Operations: NaN Values	D-5
Integer Conversion Overflow	D-6
Text-Oriented I/O Conversions	D-6
Bibliography	D-6

FLOATING-POINT ARITHMETIC

Introduction

Floating-point arithmetic in Pascal on the Lisa (all arithmetic involving real values) conforms to most of the single-precision aspects of the IEEE's *Proposed Standard for Binary Floating-Point Arithmetic* (Draft 10.0 of IEEE Task P754).

IEEE Standard arithmetic provides better accuracy than many other floating-point implementations. It also reduces the problems of overflow, underflow, limited precision, and invalid operations by providing useful ways of dealing with them.

The FPLIB library unit contains the routines that perform floating-point arithmetic (including all the transcendental functions and the `sqrt` function). FPLIB must be linked into any program that uses floating-point arithmetic; however, it is not necessary to explicitly refer to FPLIB in a `uses` clause unless the program calls the specialized support procedures and functions declared in the interface of FPLIB.

This manual assumes that you do not explicitly use the FPLIB unit, and that therefore only the default options of IEEE arithmetic are applicable.

As a general rule, you can write Lisa Pascal programs that use floating-point arithmetic without worrying about the differences between IEEE Standard arithmetic and other floating-point implementations.

The following points apply if your program writes out floating-point numbers as textual representations (via `write` or `writeln`):

- Anything in the output that looks like a number will be correct (and possibly more accurate than under other implementations).
- If your output contains a string of two or more pluses or minuses, this indicates a value of ∞ , resulting from division by zero or some other operation that caused a floating-point overflow.
- If your output contains the string "NaN" (meaning Not a Number), this indicates the result of some invalid operation that would probably have caused a program halt or a wrong output under other implementations. Note that any real value in text output that does not include the string "NaN" is guaranteed not to have been affected by any invalid operation.

Rounding of Real Results

When a real result must be rounded, it is always rounded to the nearest representable real value. If the unrounded result is exactly halfway between two representable real values, it is rounded to the value that has a zero in the least significant digit of its binary fraction (the "even" value).

Accuracy of Arithmetic Operations

The arithmetic operations `+`, `-`, `*`, `/`, `round`, `trunc`, and `sqrt` are accurate to within half a unit in the last bit. Remainders are computed without rounding error.

Overflow and Division by Zero: Infinite Values

The result of floating-point overflow is either ∞ or $-\infty$. These are values of type `real` that can be used in further calculations and follow the mathematical conventions: for example, a finite number divided by ∞ yields zero.

Dividing a finite non-zero value by zero also yields ∞ or $-\infty$ (in floating-point arithmetic).

Infinite values have textual representations that can be read by `read` or `readln` or written out by `write` or `writeln`.

Tables D-1 and D-2 below show the results of arithmetic operations on infinities. Note that any operation involving a NaN as an operand produces a NaN as the result.

Table D-1
Results of Addition and Subtraction on Infinities

<i>Left Operand</i>		<i>Right Operand</i>		
		$-\infty$	finite	$+\infty$
$-\infty$	+	$-\infty$	$-\infty$	NaN
finite		$-\infty$	finite [†]	$+\infty$
$+\infty$		NaN	$+\infty$	$+\infty$
$-\infty$	-	NaN	$-\infty$	$-\infty$
finite		$+\infty$	finite [†]	$-\infty$
$+\infty$		$+\infty$	$+\infty$	NaN

† Result is an infinity if the operation overflows.

Table D-2
Results of Multiplication and Division on Infinities

<i>Left Operand</i>		<i>Right Operand</i>		
		± 0	finite	$\pm\infty$
± 0	*	± 0	± 0	NaN
finite		± 0	finite [†]	$\pm\infty$
$\pm\infty$		NaN	$\pm\infty$	$\pm\infty$
± 0	/	NaN	± 0	± 0
finite		$\pm\infty$	finite [†]	± 0
$\pm\infty$		$\pm\infty$	$\pm\infty$	NaN

† Result is an infinity if the operation overflows.

Note: Sign of result is determined by the usual mathematical rules.

Invalid Operations: NaN Values

An invalid operation (such as dividing zero by zero) does not cause a halt. Instead it returns a special diagnostic value, and execution continues. The result of an invalid operation is called a NaN, which stands for "not a number."

A NaN resulting from an invalid operation is a propagating NaN. This means that if the NaN is used as an operand in another operation, the result of the operation will be the same NaN. NaNs can be written out via `write` or `writeln` and read via `read` or `readln`; the textual representation is "NaN" (optionally followed by a quoted string).

The following operations are invalid and return a NaN value:

- $\infty - \infty$ OR $\infty + (-\infty)$
- $0 * \pm\infty$
- $0/0$
- $\pm\infty/\pm\infty$
- The `ln` and `sqrt` functions, when the arguments are inappropriate. See Sections 11.4.7 and 11.4.8.

Integer Conversion Overflow

Integer conversion overflow can occur in `trunc` or `round` (see Chapter 11) if the actual-parameter exceeds the bounds of the predeclared type `integer`. The result returned is unspecified.

Text-Oriented I/O Conversions

The `read`, `readln`, `write`, and `writeln` procedures require the conversion of numbers from decimal to binary on input and from binary to decimal on output. The error in these conversions is less than 1 unit of the result's least significant digit. (In the past, base conversions have rarely been done accurately in a way that permits simple error bounds to be put on the results.)

Real values appear as character strings in two different contexts: as source code processed by the compiler (real constants), and in text files written and read by Pascal programs. The signed-number syntax of Chapter 1 applies in both cases. However, the Compiler does not accept infinities and NaN's.

For `read` and `write`, $+\infty$ is represented by a string of at least two plus signs, and $-\infty$ by a string of at least two minus signs. NaNs are represented by the characters "NaN", with an optional leading sign, and an optional trailing quoted string of characters; an example is

`-NaN'12:34'`

The character string is sometimes used to provide diagnostic data.

Bibliography

The following articles contain detailed information and discussion of the proposed IEEE floating-point standard. (Articles are listed in order of importance.)

- "A Proposed Standard for Binary Floating-Point Arithmetic", *IEEE Computer*, Vol. 14, No. 3, March 1981.
- Coonen, J.: "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic", *IEEE Computer*, Vol. 13, No. 1, January 1980.
- ACM SIGNUM Newsletter, special issue devoted to the proposed IEEE floating-point standard, October 1979.

INDEX

Please note that the topic references in this Index are *by section number*.

\$C compiler commands: 12.1
\$D compiler commands: 12.1
\$DECL compiler command: 12.2.1
\$E compiler command: 12.1
\$ELSEC compiler command: 12.2.4
\$ENDC compiler command: 12.2.4
\$I compiler command: 12.1
\$IFC compiler command: 12.2.4
\$L compiler commands: 12.1
\$R compiler commands: 3.1.3, 12.1
\$S compiler command: 8.3, 9.1, 9.2, 12.1
\$SETC compiler command: 12.2.1
\$U compiler commands: 9.1.2, 9.2.2, 12.1
\$X compiler commands: 12.1
0, signed: 3.1.1.3
16-bit integer arithmetic: 3.1.1.1-2, 11.3.3
32-bit integer arithmetic: 3.1.1.2, 11.3.3
@ operator: 3.3, 5.1.6
-----A-----
abs function: 11.4.2
accuracy in real arithmetic: D
actual-parameter: 5.2, 7.1, 7.3
actual-parameter-list: 5.2
actual-parameters in procedure call: 6.1.2
anomalies in Lisa Pascal: B
Apple II & III Pascals: A
Apple II Pascal: A
Apple III Pascal: A
applestuff unit: A
arctan function: 11.4.9
arithmetic functions: 11.4
arithmetic operators: 5.1.2, D

array: 3.2.1, 4.3.1
- component: 3.2.1, 4.3.1
- reference: 4.3.1
array-type: 3.2.1
ascii: 3.1.1.5
assignment-compatibility: 3.4.3
assignment-statement: 6.1.1
-----B-----
base-type: 3.2.3, 3.3, 5.3
- of pointer-type: 3.3
- scope anomaly: B
- of set-type: 3.2.3, 5.3
bitwise boolean operations: A
blank character: 1.1
blank segment: 8.3, 9.1
block: 2
block-structured I/O: 3.2.4, 10.1.1-2, 10.4
blockread function: 3.2.4, 10.4.1
blockwrite function: 3.2.4, 10.4.2
boolean: 3.1.1.4, 5.1.3, 5.1.5.2, 10.3.3.7, 12.3-12.4
- comparisons: 5.1.5.2
- constants as control values: 12.3, 12.4
- operands, evaluation of: 5.1.3
- operators: 5.1.3
- type: 3.1.1.4
- values in text-oriented output: 10.3.3.7
buffer variable: 10.1.3, 10.1.7
built-in procedures & functions: 10, 11
byte array: 11.7
byte-oriented procedures & functions: 11.7
byte-size files: 3.2.4
bytestream type: A
-----C-----
case: 6.2.2.2
case-constant in case statement: 6.2.2.2
case-sensitivity: 1.1, 1.2, 1.4
case-statement: 6.2.2.2
- efficiency: 12.5

- char: 1.6.1, 3.1.1.5, 10.3.1.1, 10.3.3.2, 11.5
 - constant: 1.6.1
 - type: 3.1.1.5
 - values in text-oriented I/O: 10.3.1.1, 10.3.3.2
- character: 1.1, 3.2.4, 4.3.1
 - device: 3.2.4, 10.1.1-2
 - files: 3.2.4
 - in string: 4.3.1
 - set: 1.1
- chr function: 11.5.2
- close procedure: 10.1.5
- closing a file: 10.1.5
- code generation: 12.1
- comment: 1.8
- comparisons: 5.1.5
- compatibility of parameter lists: 7.3.5
- compatible types: 3.4
- compile-time expressions & variables: 12.2.1-3
- compiler: 1.8, 12, A
 - commands: 1.8, 12.1-2, A
- component of array: 3.2.1, 4.3.1
- component of file: 3.2.4, 4.3.3
- component-type of array: 3.2.1
- component-type of file: 3.2.4
- compound-statement: 6.2.1
- concat function: 11.6.3
- conditional compilation: 12.2
- conditional-statement: 6.2.2
- constant: 1.4-7
- constant-declaration: 1.7, 2.1, B
 - scope anomaly: B
- constant-declaration-part: 2.1
- control-variable: 6.2.3.3
- copy function: 11.6.4
- cos function: 11.4.5
- CR character: 1.1, 1.6, 10.3
 - in text-oriented I/O: 10.3
- crunch: 10.1.5
- current block number: 10.4
- current file position: 4.3.3
- cursor control: 10.3.7

-----D-----

data type: 3
datafile: 10.1.2
debugging: 12.1
defining declaration: 7.1
delete procedure: 11.6.5
device: 10.1.1-2
- character: 10.1.1, 10.1.2
- file-structured: 10.1.1, 10.1.2
- types: 10.1.1, 10.1.2
digit: 1.1
digit-sequence: 1.4
directive: 1.3
div operator: A
division by zero (real arithmetic): 3.1.1.3, D
DLE character: 10.3
dynamic allocation procedures: 11.2

-----E-----

efficiency, case-statements: 12.5
empty set: 5.3
enumerated-type: 3.1.2
eof function: 10.1.7
- and various procedures: 10.1.3-4, 10.1.7, 10.2.1-2, 10.2.4, 10.3.1-2, 10.4.1
eoln function: 10.3.5
- and read and readln procedures: 10.3.1, 10.3.2
ETX character: A
exit procedure: 11.1.1, A
exp function: 11.4.6
expression: 5
extended comparisons: A
external file: 10.1
external function: 7.2
external procedure: 7.1-2

-----F-----

factor: 5
field of record: 3.2.2, 4.3.2, 6.2.4
field-declaration: 3.2.2
field-designator: 4.3.2
field-list: 3.2.2

- file: 3.2.4, 4.3.3, 10
 - buffer: 4.3.3
 - buffer and eof function: 10.1.7
 - buffer and reset procedure: 10.1.3
 - component: 3.2.4, 4.3.3
 - identifier as parameter type: 7.3
 - of char: 3.2.4
 - position and reset procedure: 10.1.3
 - record: 10.2
 - reference: 4.3.3
 - species: 10.1.2
 - standard file-type identifier: 3.2.4, 10.1, 10.4
 - types and reset procedure: 10.1.3
 - variable: 3.2.4, 4.3.3, 10
- file-buffer-symbol: 4.3.3
- file-structured device: 3.2.4, 10.1.1-2, 10.4
- file-type: 3.2.4
- fillchar procedure: 11.8.3
- final-value: 6.2.3.3
- finite real values: 3.1.1.3
- fixed-part: 3.2.2
- fixed-point output of real value: 10.3.3.4
- floating-point arithmetic: D
- floating-point output of real value: 10.3.3.4, A
- for-statement: 6.2.3.3
- formal-parameter-list: 7.3
- formal-parameters and procedure call: 6.1.2
- forward declaration: 7.1-2
- function: 7.2-3
- function-body: 7.2

- function-call: 5, 5.2, 7.2, 7.3
- function-declaration: 7.2
- function-heading: 7.2
- functional parameter: 7.3.4
- G-----
- get procedure: 10.2.1, 10.2.3
- goto-statement: 6.2, A
- gotoxy procedure: 10.3.7.2

-----H-----
halt procedure: 11.1.2, A
heap: 11.2
heapresult function: 11.2.2
hex-digit: 1.1
hex-digit-sequence: 1.4
hexadecimal constants: 1.4
host program or unit: 9
host-type of subrange: 3.1.3
-----I, J-----
identical types: 3.4
identifier: 1.2
- of program: 8.1
identifier-list: 3.1.2
IEEE Floating-Point Standard: D
if-statement: 6.2.2.1
- optimization: 12.3
implementation-part: 9.1.1
in operator: 5.1.5.5
index: 4.3.1
- in variable-reference: 4.3.1
index-type: 3.2.1
infinities: 3.1.1.3, D
initial-value: 6.2.3.3
initialization-part: A
input (standard file): 10.1.7, 10.3
input file control (in compilation): 12.1
input variables in read procedure: 10.3.1
input/output: 10
insert procedure: 11.6.6
integer: 1.4, 3.1.1.1-2, 10.3.1.2, 10.3.3.3, 11.3-5, D
- arithmetic: 3.1.1.1, 3.1.1.2
- constant: 1.4
- conversion overflow: D
- type: 3.1.1.1, 3.1.1.2
- type conversions: 3.1, 3.1.1.5, 3.1.2, 11.5.1
- values in text-oriented I/O: 10.3.1.2, 10.3.3.3

interactive file-type: A
interface-part: 9.1.1
intrinsic-unit: 9.2
INTRINSIC.LIB: 9.2, 12.1
invalid operations in real arithmetic: D
ioresult function: 10.1.2, 10.1.6
-----K-----
keyboard: 3.2.4, 10.1.1, 10.3, 10.3.7.1
- echoing on input: 10.3
- physical: 3.2.4, 10.1.1, 10.3, 10.3.7.1
- testing: 10.3.7.1
keypress function: 10.3.7.1
-----L-----
label: 1.5, 2.1, 6
- on statement: 6
label-declaration-part: 2.1
length attribute: 3.1.1.6
length function: 11.6.1
letter: 1.1
Linker: 7.1
listing control: 12.1
ln function: 11.4.7
lock: 10.1.5
long integer type: A
longint: 1.4, 3.1.1.2, 10.3.1.2, 10.3.3.3, 11.3-5, D
- arithmetic: 3.1.1.2
- constant: 1.4, 1.6, 1.7
- type: 3.1.1.2
- type conversions: 11.3.3, 11.3.4
- values in text-oriented I/O: 10.3.3.3
-----M-----
mark procedure: 11.2.3, A
maxint: 3.1.1.1
memavail function: 11.2.5
member-group: 5.3
memory allocation procedures: 11.2
mod operator: A
moveleft procedure: 11.7.1
moveright procedure: 11.7.2

-----N-----

NaNs: 3.1.1.3, D
new procedure: 3.3, 11.2.1, A
nil: 3.3, 4.3.4, 11.2.1
normal: 10.1.5
number: 1.4
numerical comparisons: 5.1.5.1

-----O-----

object file: 9
object of pointer: 4.3.4
odd function: 11.4.1
opening a file: 10.1, 10.1.2-4
operands: 5
- compile-time: 12.2.3
- in expressions: 5
operators: 5
- compile-time: 12.2.3
- in expressions: 5
optimization of if, repeat, and while statements: 12.3, 12.4
ord function: 3.1, 3.1.1.5, 3.1.2, 11.5.1
ord4 function: 3.1.1.2, 11.3.3
order of evaluation of operands: 5.1.1
ordinal functions: 11.5
ordinal-type: 3.1
- and ord function: 11.5.1
- and ord4 function: 11.3.3
- and pred function: 11.5.4
- and succ function: 11.5.3
ordinal-type-identifier: 3
ordinality: 3.1
otherwise-clause: 6.2.2.2
output (standard file): 10.3
output expression in write procedure: 10.3.3
output file in write procedure: 10.3.3
output-specs in write procedure: 10.3.3
overflow (real arithmetic): 3.1.1.3, D

-----p-----
packed array of char: 5.1.5.6, 10.3.1.5, 10.3.3.6, 11.8
- comparisons: 5.1.5.6
- fillchar procedure: 11.8.3
- scanning functions: 11.8.1, 11.8.2
- text-oriented I/O: 10.3.1.5, 10.3.3.6
packed types: 3.1.1.6, 3.2
page procedure: 10.3.6
parameter: 7.1, 7.3
parameter list compatibility: 7.3.5
parameter-declaration: 7.3
parameters in procedure call: 6.1.2
Pascal compiler: 12
performance penalty for longint values: 3.1.1.2
pointer: 4.3.4, 11.2
pointer function: 3.3, 11.3.4
pointer-object-symbol: 4.3.4
pointer-reference: 4.3.4
pointer-type: 3.3
- conversions: 11.3.3, 11.3.4
pointer-type-identifier: 3
pos function: 11.6.2
precedence of operators: 5
pred function: 3.1, 11.5.4
predecessor: 3.1
procedural parameter: 7.3.3
procedure: 7.1, 7.3
procedure-and-function-declaration-part: 2.1
procedure-body: 7.1
procedure-declaration: 7.1
procedure-heading: 7.1
procedure-statement: 6.1.2, 7.1
program: 8
- identifier: 8.1
- segments: 8.3
program-heading: 8.1
program-parameters: 8.1, 8.2
purge: 10.1.5
put procedure: 10.2.2-3
pwoften function: 11.4.10

-----Q-----

qualifier: 4.3
quoted-character-constant: 1.6.1
quoted-string-constant: 1.6

-----R-----

range-checking: 3.1.3, 12.1
read procedure: 10.3.1
readln procedure: 10.3.2
real: 1.4, 3.1.1.3, 10.3.1.3, 10.3.3.4, 11.3-4, D
- arithmetic: D
- constant: 1.4
- type: 3.1.1.3, D
- type and round function: 11.3.2
- values: 3.1.1.3
- values and write procedure: D
- values in text-oriented I/O: 10.3.1.3, 10.3.3.4, D
real-type: 3.1
real-type-identifier: 3
record: 3.2.2, 4.3.2
- field: 3.2.2, 4.3.2
- number and seek procedure: 10.2.4
- of file: 10.2
- reference: 4.3.2
- reference in with statement: 6.2.4
record-oriented I/O: 10.2
record-type: 3.2.2
- new procedure: 11.2.1
recursion: 7.1-2
redeclaration of identifier: 2.2.2, 2.2.4
regular-unit: 9.1
relational operators: 5.1.5
release procedure: 11.2.4, A
repeat-statement: 6.2.3.1
- optimization: 12.4
repetitive-statement: 6.2.3
reserved words: 1.1
reset procedure: 10.1, 10.1.5, A
result-type: 7.2
rewrite procedure: 10.1.4
round function: 11.3.2, D
rounding in real arithmetic: D

-----S-----

scale-factor: 1.4
scan function: A
scaneq function: 11.8.1
scanne function: 11.8.2
scope: 2.2
- of standard objects: 2.2.5
screen: 10.3, 10.3.7.2
- cursor control: 10.3.7.2
- physical: 10.3
seek procedure: 10.2.3
segment keyword: A
segmentation: 8.3
segments: 8.3, 9.1, 9.2.1
selector in case statement: 6.2.2.2
set: 3.2.3, 5.1.4, 5.1.5.4, 5.3
- comparisons: 5.1.5.4
- membership testing: 5.1.5.5
- operators: 5.1.4
- values: 5.3
set-constructor: 5, 5.3
set-type: 3.2.3
sign: 1.4
signed zero: 3.1.1.3
signed-number: 1.4
simple-expression: 5
simple-statement: 6.1
simple-type: 3.1
simple-type-identifier: 3
sin function: 11.4.4
size-attribute: 3.1.1.6
sizeof function: 11.7.3
special symbols: 1.1
sqr function: 11.4.3
sqrt function: 11.4.8, D
stack space and memavail function: 11.2.5
standard procedures and functions: 10, 11
- for I/O: 10
standard simple-types: 3.1
statement: 6

statement-part: 2.1
string: 1.6, 3.1.1.6, 4.3.1, 5.1.5.3, 10.3.1.4, 10.3.3.5, 11.6, A
- character: 4.3.1
- comparisons: 5.1.5.3
- concatenation: 11.6.3
- constant: 1.6, 3.1.1.6
- constant comparisons: 5.1.5.3
- length function: 11.6.1
- procedures and functions: 11.6
- reference: 4.3.1
- substring copying: 11.6.4
- substring deletion: 11.6.5
- substring insertion: 11.6.6
- substring search: 11.6.2
- values in text-oriented I/O: 10.3.1.4, 10.3.3.5
string-character: 1.6
string-type: 3.1.1.6
string-type-identifier: 3
structured-statement: 6.2
structured-type: 3.2
structured-type-identifier: 3
subrange-type: 3.1.3
succ function: 3.1, 11.5.3
successor: 3.1
syntax diagrams, complete collection: C
syntax diagrams, explanation: Preface
system intrinsic library: 9.2.2, 12.1
-----T-----
tag constants in new and dispose procedures: 11.2.1-2
tag-field: 3.2.2
tag-field-type: 3.2.2
term: 5
testing set membership: 5.1.5.5
text type: 3.2.4, 10.1.2, 10.3
text-oriented I/O: 10.3
textfile: 10.1.2, 10.3, A
textfile format: 10.1.2, 10.3
transfer functions: 11.3
treesearch procedure: A

trunc function: 11.3.1, A, D
turtlegraphics unit: A
type: 3
- compatibility and identity: 3.4
type-declaration: 3
type-declaration-part: 2.1, 3.5
-----U-----
UCSD Pascal: A
unary arithmetic operators: 5.1.2
underscore character: A
unit: 9
- intrinsic: 9.2
- regular: 9.1
unit-heading: 9.1.1
unsigned-constant: 5
unsigned-integer: 1.4
unsigned-number: 1.4
unsigned-real: 1.4
untyped file: 3.2.4, 10.1.1-2, 10.4
- I/O: 10.4
uses-clause: 8.1, 9.1.1-2, 9.2, 9.3
-----V-----
value parameter: 7.3.1
variable: 4
variable parameter: 7.3.2, A
variable-declaration: 4.1
variable-declaration-part: 2.1
variable-identifier: 4.1
variable-reference: 4.2
variant: 3.2.2
- records, new procedure: 11.2.1
variant-part: 3.2.2
-----W, X, Y-----
while-statement: 6.2.3.2
- optimization: 12.4

with-statement: 6.2.4
wordstream type: A
write procedure: 10.3.3, A
- with real values: D
write-protection of file: 10.1.5
writeln procedure: 10.3.4, A
-----Z-----
zero, signed: 3.1.1.3

Appendix E

QUICKDRAW

E.1	About This Manual	E-3
E.2	About QuickDraw	E-4
E.2.1	How To Use QuickDraw	E-5
E.3	The Mathematical Foundation of QuickDraw	E-6
E.3.1	The Coordinate Plane	E-6
E.3.2	Points	E-7
E.3.3	Rectangles	E-7
E.3.4	Regions	E-9
E.4	Graphic Entities	E-11
E.4.1	The Bit Image	E-12
E.4.2	The BitMap	E-13
E.4.3	Patterns	E-15
E.4.4	Cursors	E-16
E.5	The Drawing Environment: GrafPort	E-17
E.5.1	Pen Characteristics	E-21
E.5.2	Text Characteristics	E-23
E.6	Coordinates in GrafPorts	E-25
E.7	General Discussion of Drawing	E-27
E.7.1	Transfer Modes	E-29
E.7.2	Drawing in Color	E-31
E.8	Pictures and Polygons	E-31
E.8.1	Pictures	E-32
E.8.2	Polygons	E-33
E.9	QuickDraw Routines	E-35
E.9.1	GrafPort Routines	E-35
E.9.2	Cursor-Handling Routines	E-40
E.9.3	Pen and Line-Drawing Routines	E-41
E.9.4	Text-Drawing Routines	E-44
E.9.5	Drawing in Color	E-46
E.9.6	Calculations with Rectangles	E-47
E.9.7	Graphic Operations on Rectangles	E-50
E.9.8	Graphic Operations on Ovals	E-50
E.9.9	Graphic Operations on Rounded-Corner Rectangles	E-51
E.9.10	Graphic Operations on Arcs and Wedges	E-53
E.9.11	Calculations with Regions	E-55
E.9.12	Graphic Operations on Regions	E-60
E.9.13	Bit Transfer Operations	E-61
E.9.14	Pictures	E-63
E.9.15	Calculations with Polygons	E-64
E.9.16	Graphic Operations on Polygons	E-66

E.9.17	Calculations with Points	E-67
E.9.18	Miscellaneous Utilities	E-69
E.10	Customizing QuickDraw Operations	E-72
E.11	Using QuickDraw from Assembly Language	E-75
E.11.1	Constants	E-76
E.11.2	Data Types	E-76
E.11.3	Global Variables.....	E-77
E.11.4	Procedures and Functions	E-78
E.12	Summary of QuickDraw	E-79
E.13	QuickDraw Sample Program	E-95
E.14	Glossary	E-104

QUICKDRAW

E.1 About This Manual

This manual describes QuickDraw, a set of graphics procedures, functions, and data types that allows a Pascal or assembly language programmer of Lisa to perform highly complex graphic operations very easily and very quickly. It covers the graphic concepts behind QuickDraw, as well as the technical details of the data types, procedures, and functions you will use in your programs.

We assume that you are familiar with the Lisa Workshop Manager, Lisa Pascal, and the Lisa Operating System's memory management. This graphics package is for programmers, not end users. Although QuickDraw may be used from either Pascal or assembly language, this manual gives all examples in their Pascal form, to be clear, concise, and more intuitive; Section E.11 describes the details of the assembly language interface to QuickDraw.

The manual begins with an introduction to QuickDraw and what you can do with it (Section E.2). It then steps back a little and looks at the mathematical concepts that form the foundation for QuickDraw: coordinate planes, points, and rectangles (Section E.3). Once you understand these concepts, read on to Section E.4, which describes the graphic entities based on them—how the mathematical world of planes and rectangles is translated into the physical phenomena of light and shadow.

Then comes some discussion of how to use several graphics ports (Section E.6), a summary of the basic drawing process (Section E.7), and a discussion of two more parts of QuickDraw, pictures and polygons (Section E.8).

Next, in Section E.9, there's a detailed description of all QuickDraw procedures and functions, their parameters, calling protocol, effects, side effects, and so on—all the technical information you'll need each time you write a program for the Lisa.

Following these descriptions are sections that will not be of interest to all readers. Special information is given in Section E.10 for programmers who want to customize QuickDraw operations by overriding the standard drawing procedures, and in Section E.11 for those who will be using QuickDraw from assembly language.

Finally, there's a summary of the QuickDraw data structures and routine calls (Section E.12), and a glossary that explains terms that may be unfamiliar to you (Section E.13).

E.2 About QuickDraw

QuickDraw allows you to organize the Lisa screen into a number of individual areas. Within each area you can draw many things, as illustrated in Figure E-1.

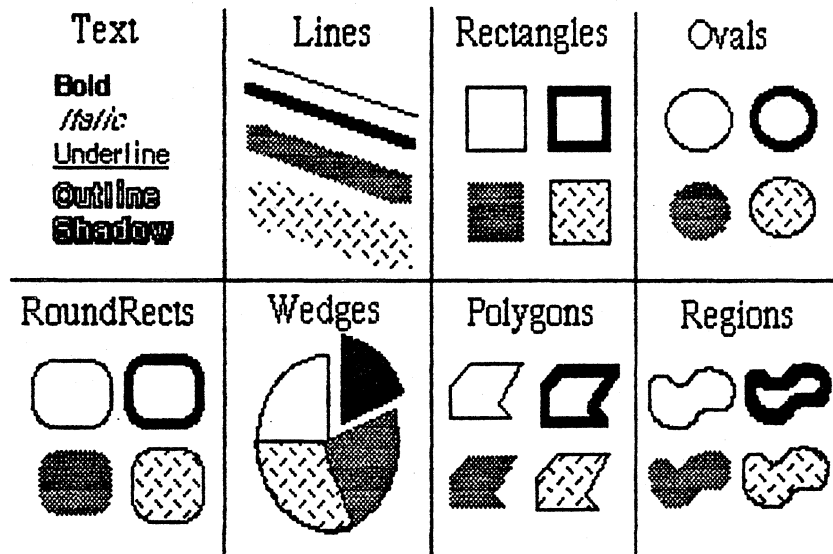


Figure E-1
Samples of QuickDraw's Abilities

You can draw:

- Text characters in a number of proportionally-spaced fonts, with variations that include boldfacing, italicizing, underlining, and outlining.
- Straight lines of any length and width.
- A variety of shapes, either solid or hollow, including: rectangles, with or without rounded corners; full circles and ovals or wedge-shaped sections; and polygons.
- Any other arbitrary shape or collection of shapes, again either solid or hollow.
- A picture consisting of any combination of the above items, with just a single procedure call.

In addition, QuickDraw has some other abilities that you won't find in many other graphics packages. These abilities take care of most of the "housekeeping"--the trivial but time-consuming and bothersome overhead that's necessary to keep things in order.

- The ability to define many distinct "ports" on the screen, each with its own complete drawing environment--its own coordinate system, drawing location, character set, location on the screen, and so on. You can easily switch from one such port to another.
- Full and complete "clipping" to arbitrary areas, so that drawing will occur only where you want. It's like a super-duper coloring book that won't let you color outside the lines. You don't have to worry about accidentally drawing over something else on the screen, or drawing off the screen and destroying memory.
- Off-screen drawing. Anything you can draw on the screen, you can draw into an off-screen buffer, so you can prepare an image for an output device without disturbing the screen, or you can prepare a picture and move it onto the screen very quickly.

And QuickDraw lives up to its name! It's very fast. The speed and responsiveness of the Lisa user interface is due primarily to the speed of the QuickDraw package. You can do good-quality animation, fast interactive graphics, and complex yet speedy text displays using the full features of QuickDraw. This means you don't have to bypass the general-purpose QuickDraw routines by writing a lot of special routines to improve speed.

E.2.1 How To Use QuickDraw

QuickDraw can be used from either Pascal or MC68000 machine language. It has no user interface of its own; you must write and compile (or assemble) a Pascal (or assembly-language) program that includes the proper QuickDraw calls, link the resulting object code with the QuickDraw code, and execute the linked object file.

A programming model is included with the Workshop software; it shows the structure of a properly organized QuickDraw program. What's best for beginners is to read through the text, and, using the superstructure of the program as a "shell", modify it to suit your own purposes. Once you get the hang of writing programs inside the presupplied shell, you can work on changing the shell itself.

QuickDraw includes only the graphics and utility procedures and functions you'll need to create graphics on the screen. Procedures for dealing with the Mouse, Cursors, Keyboard, and screen settings, as well as those allowing you to generate sounds, and read and set clocks and dates, are available in the unit "Hardware".

E.3 The Mathematical Foundation of QuickDraw

To create graphics that are both precise and pretty requires not supercharged features but a firm mathematical foundation for the features you have. If the mathematics that underlie a graphics package are imprecise or fuzzy, the graphics will be, too. QuickDraw defines some clear mathematical constructs that are widely used in its procedures, functions, and data types: the coordinate plane, the point, the rectangle, and the region.

E.3.1 The Coordinate Plane

All information about location, placement, or movement that you give to QuickDraw is in terms of coordinates on a plane. The coordinate plane is a two-dimensional grid, as illustrated in Figure E-2.

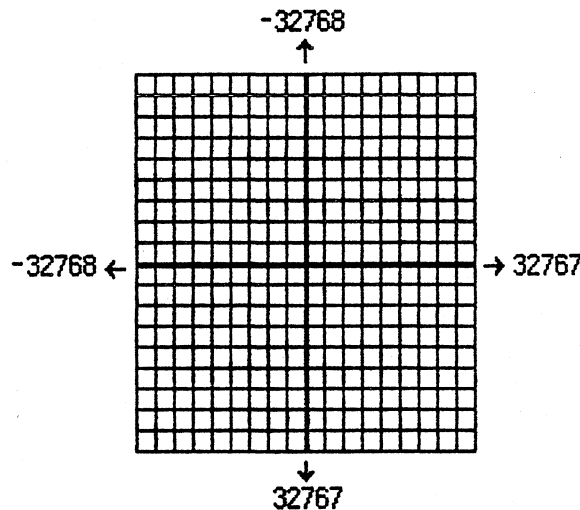


Figure E-2
The Coordinate Plane

There are two distinctive features of the QuickDraw coordinate plane:

- All grid coordinates are integers.
- All grid lines are infinitely thin.

These concepts are important! First, they mean that the QuickDraw plane is finite, not infinite (although it's very large). Horizontal coordinates range from -32768 to $+32767$, and vertical coordinates have the same range.

Second, they mean that all elements represented on the coordinate plane are mathematically pure. Mathematical calculations using integer arithmetic will

produce intuitively correct results. If you keep in mind that grid lines are infinitely thin, you'll never have "endpoint paranoia"--the confusion that results from not knowing whether that last dot is included in the line.

E.3.2 Points

On the coordinate plane are 4,294,967,296 unique points. Each point is at the intersection of a horizontal grid line and a vertical grid line. As the grid lines are infinitely thin, a point is infinitely small. Of course there are more points on this grid than there are dots on the Lisa screen: when using QuickDraw you associate small parts of the grid with areas on the screen, so that you aren't bound into an arbitrary, limited coordinate system.

The coordinate origin (0,0) is in the middle of the grid. Horizontal coordinates increase as you move from left to right, and vertical coordinates increase as you move from top to bottom. This is the way both a TV screen and a page of English text are scanned: from the top left to the bottom right.

You can store the coordinates of a point in a Pascal variable whose type is defined by QuickDraw. The type Point is a record of two integers, and has the following structure:

```

type  VHSelect = (V,H);
      Point    = record case integer of

          0: (v: integer;
              h: integer);

          1: (vh: array [VHSelect] of integer)

      end;
```

The variant part allows you to access the vertical and horizontal components of a point either individually or as an array. For example, if the variable `typegoodPt` were declared to be of type `Point`, the following would all refer to the coordinate parts of the point:

```

goodPt.v          goodPt.h
goodPt.vh[V]     goodPt.vh[H]
```

E.3.3 Rectangles

Any two points can define the top left and bottom right corners of a rectangle. As these points are infinitely small, the borders of the rectangle are infinitely thin (see Figure E-3).

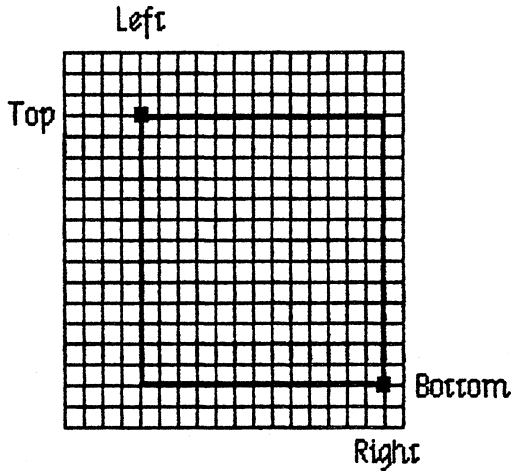


Figure E-3
A Rectangle

Rectangles are used to define active areas on the screen, to assign coordinate systems to graphic entities, and to specify the locations and sizes for various drawing commands. QuickDraw also allows you to perform many mathematical calculations on rectangles--changing their sizes, shifting them around, and so on.

NOTE

Remember that rectangles, like points, are mathematical concepts that have no direct representation on the screen. The association between these conceptual elements and their physical representations is made by a bitMap, described below.

The data type for rectangles is called `Rect`, and consists of four integers or two points:

```

type Rect = record case integer of
    0: (top: integer;
        left: integer;
        bottom: integer;
        right: integer);
    1: (topLeft: Point;
        botRight: Point)
end;
```

Again, the record variant allows you to access a variable of type `Rect` either as four boundary coordinates or as two diagonally opposing corner points. Combined with the record variant for points, all of the following references to the rectangle named `bRect` are legal:

<code>bRect</code>		{type <code>Rect</code> }
<code>bRect.topLeft</code>	<code>bRect.botRight</code>	{type <code>Point</code> }
<code>bRect.top</code>	<code>bRect.left</code>	{type <code>integer</code> }
<code>bRect.topLeft.v</code>	<code>bRect.topLeft.h</code>	{type <code>integer</code> }
<code>bRect.topLeft.vh[V]</code>	<code>bRect.topLeft.vh[H]</code>	{type <code>integer</code> }
<code>bRect.bottom</code>	<code>bRect.right</code>	{type <code>integer</code> }
<code>bRect.botRight.v</code>	<code>bRect.botRight.h</code>	{type <code>integer</code> }
<code>bRect.botRight.vh[V]</code>	<code>bRect.botRight.vh[H]</code>	{type <code>integer</code> }

WARNING

If the bottom coordinate of a rectangle is equal to or less than the top, or the right coordinate is equal to or less than the left, the rectangle is an empty rectangle (i.e., one that contains no bits).

E.3.4 Regions

Unlike most graphics packages that can manipulate only simple geometric structures (usually rectilinear, at that), QuickDraw can gather an arbitrary set of spatially coherent points into a structure called a region, and perform complex yet rapid manipulations and calculations on such structures. This remarkable feature not only will make your standard programs simpler and faster, but will

let you perform operations that would otherwise be nearly impossible; it is fundamental to the Lisa user interface.

You define a region by drawing lines, shapes such as rectangles and ovals, or even other regions. The outline of a region should be one or more closed loops. A region can be concave or convex, can consist of one area or many disjoint areas, and can even have "holes" in the middle. In Figure E-4, the region on the left has a hole in the middle, and the region on the right consists of two disjoint areas.



Figure E-4
Regions

Because a region can be any arbitrary area or set of areas on the coordinate plane, it takes a variable amount of information to store the outline of a region. The data structure for a region, therefore, is a variable-length entity with two fixed fields at the beginning, followed by a variable-length data field:

```
type Region = record
    rgnSize: integer;
    rgnBBox: Rect;
    {optional region definition data}
end;
```

The `rgnSize` field contains the size, in bytes, of the region variable. The `rgnBBox` field is a rectangle which completely encloses the region.

The simplest region is a rectangle. In this case, the `rgnBBox` field defines the entire region, and there is no optional region data. For rectangular regions (or empty regions), the `rgnSize` field contains 10.

The region definition data for nonrectangular regions is stored in a compact way which allows for highly efficient access by QuickDraw procedures.

As regions are of variable size, they are stored dynamically on the heap, and the Operating System's memory management moves them around as their sizes change. Being dynamic, a region can be accessed only through a pointer; but when a region is moved, all pointers referring to it must be updated. For this reason, all regions are accessed through handles, which point to one master pointer which in turn points to the region.

```

type RgnPtr    = ^Region;
     RgnHandle = ^RgnPtr;

```

When the memory management relocates a region's data in memory, it updates only the RgnPtr master pointer to that region. The references through the master pointer can find the region's new home, but any references pointing directly to the region's previous position in memory would now point at dead bits. To access individual fields of a region, use the region handle and double indirection:

```

myRgn ^^.rgnSize      {size of region whose handle is myRgn}
myRgn ^^.rgnBBox      {rectangle enclosing the same region}
myRgn ^^.rgnBBox.top  {minimum vertical coordinate of all
                      points in the region}
myRgn ^ .rgnBBox      {semantically incorrect; will not compile
                      if myRgn is a rgnHandle}

```

Regions are created by a QuickDraw function which allocates space for the region, creates a master pointer, and returns a rgnHandle. When you're done with a region, you dispose of it with another QuickDraw routine which frees up the space used by the region. Only these calls allocate or deallocate regions; do not use the Pascal procedure NEW to create a new region!

You specify the outline of a region with procedures that draw lines and shapes, as described in Section E.9, QuickDraw Routines. An example is given in the discussion of CloseRgn in Section E.9.11, Calculations with Regions.

Many calculations can be performed on regions. A region can be "expanded" or "shrunk" and, given any two regions, QuickDraw can find their union, intersection, difference, and exclusive-OR; it can also determine whether a given point or rectangle intersects a given region, and so on. There is of course a set of graphic operations on regions to draw them on the screen.

E.4 Graphic Entities

Coordinate planes, points, rectangles, and regions are all good mathematical models, but they aren't really graphic elements—they don't have a direct physical appearance. Some graphic entities that do have a direct graphic interpretation are the bit image, bitMap, pattern, and cursor. This section

describes the data structure of these graphic entities and how they relate to the mathematical constructs described above.

E.4.1 The Bit Image

A bit image is a collection of bits in memory which have a rectilinear representation. Take a collection of words in memory and lay them end to end so that bit 15 of the lowest-numbered word is on the left and bit 0 of the highest-numbered word is on the far right. Then take this array of bits and divide it, on word boundaries, into a number of equal-size rows. Stack these rows vertically so that the first row is on the top and the last row is on the bottom. The result is a matrix like the one shown in Figure E-5--rows and columns of bits, with each row containing the same number of bytes. The number of bytes in each row of the bit image is called the row width of that image.

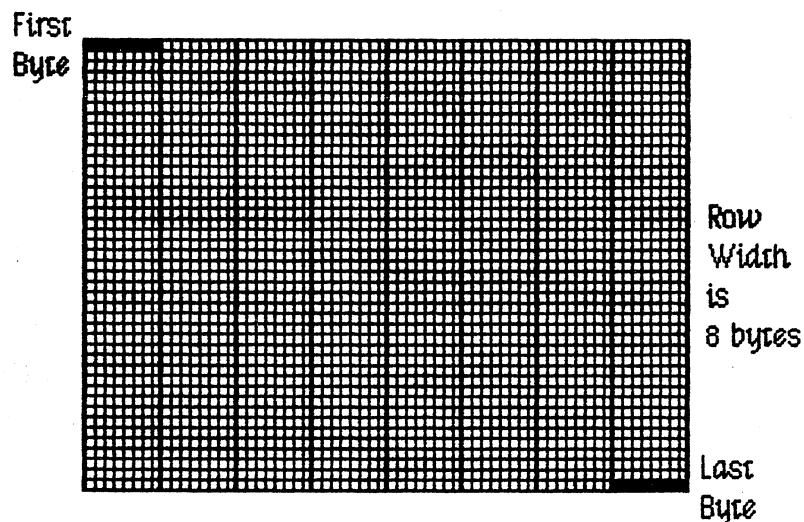


Figure E-5
A Bit Image

A bit image can be stored in any static or dynamic variable, and can be of any length that is a multiple of the row width.

The Lisa screen itself is one large visible bit image. There are 32,760 bytes of memory that are displayed as a matrix of 262,080 pixels on the screen, each bit corresponding to one pixel. If a bit's value is 0, its pixel is white; if the bit's value is 1, the pixel is black.

The screen is 364 pixels tall and 720 pixels wide, and the row width of its bit image is 90 bytes. Each pixel on the screen is one and a half times taller than it

is wide, meaning a rectangle 30 pixels wide by 20 tall looks square, and a 30 by 20 oval looks circular. There are 90 pixels per inch horizontally, and 60 per inch vertically.

NOTE

Since each pixel on the screen represents one bit in a bit image, wherever this document says "bit", you can substitute "pixel" if the bit image is the Lisa screen. Likewise, this document often refers to pixels on the screen where the discussion applies equally to bits in an off-screen bit image.

E.4.2 The BitMap

When you combine the physical entity of a bit image with the conceptual entities of the coordinate plane and rectangle, you get a bitMap. A bitMap has three parts: a pointer to a bit image, the row width (in bytes) of that image, and a boundary rectangle which gives the bitMap both its dimensions and a coordinate system. Notice that a bitMap does not actually include the bits themselves: it points to them.

There can be several bitMaps pointing to the same bit image, each imposing a different coordinate system on it. This important feature is explained more fully in Section E.6, Coordinates in GrafPorts.

As shown in Figure E-6, the data structure of a bitMap is as follows:

```
type BitMap = record
    baseAddr: QDPtr;
    rowBytes: integer;
    bounds: Rect
end;
```

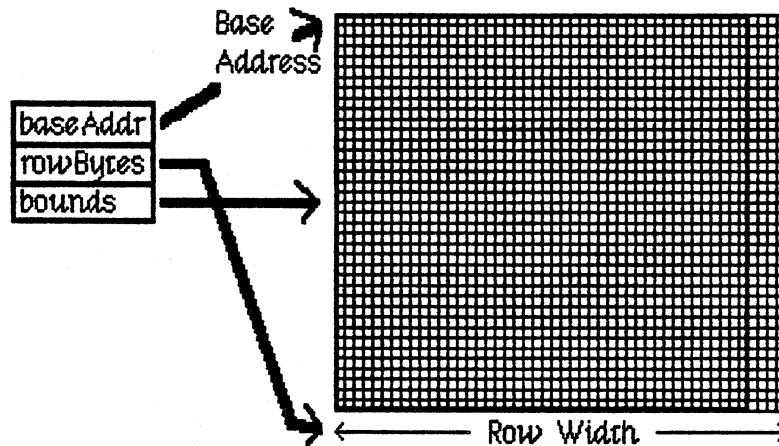


Figure E-6
A BitMap

The `baseAddr` field is a pointer to the beginning of the bit image in memory, and the `rowBytes` field is the number of bytes in each row of the image. Both of these should always be even: a bitMap should always begin on a word boundary and contain an integral number of words in each row.

The `bounds` field is a boundary rectangle that both encloses the active area of the bit image and imposes a coordinate system on it. The relationship between the boundary rectangle and the bit image in a bitMap is simple yet very important. First, a few general rules:

- Bits in a bit image fall between points on the coordinate plane.
- A rectangle divides a bit image into two sets of bits: those bits inside the rectangle and those outside the rectangle.
- A rectangle that is H points wide and V points tall encloses exactly $(H-1)*(V-1)$ bits.

The top left corner of the boundary rectangle is aligned around the first bit in the bit image. The width of the rectangle determines how many bits of one row are logically owned by the bitMap; the relationship

$$8 * \text{map.rowBytes} \geq \text{map.bounds.right} - \text{map.bounds.left}$$

must always be true. The height of the rectangle determines how many rows of the image are logically owned by the bitMap. To ensure that the number of bits

in the logical bitMap is not larger than the number of bits in the bit image, the bit image must be at least as big as

$(\text{map.bounds.bottom} - \text{map.bounds.top}) * \text{map.rowBytes}$.

Normally, the boundary rectangle completely encloses the bit image: the width of the boundary rectangle is equal to the number of bits in one row of the image, and the height of the rectangle is equal to the number of rows in the image. If the rectangle is smaller than the dimensions of the image, the least significant bits in each row, as well as the last rows in the image, are not affected by any operations on the bitMap.

The bitMap also imposes a coordinate system on the image. Because bits fall between coordinate points, the coordinate system assigns integer values to the lines that border and separate bits, not to the bit positions themselves. For example, if a bitMap is assigned the boundary rectangle with corners $(10, -8)$ and $(34, 8)$, the bottom right bit in the image will be between horizontal coordinates 33 and 34, and between vertical coordinates 7 and 8 (see Figure E-7).

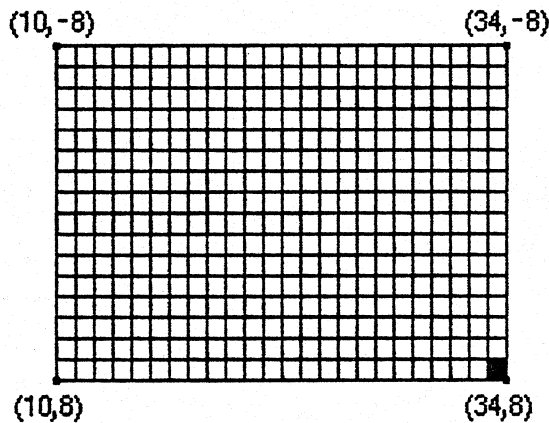


Figure E-7
Coordinates and BitMaps

E.4.3 Patterns

A pattern is a 64-bit image, organized as an 8-by-8-bit rectangle, which is used to define a repeating design (such as stripes) or tone (such as gray). Patterns can be used to draw lines and shapes or to fill areas on the screen.

When a pattern is drawn, it is aligned such that adjacent areas of the same pattern in the same graphics port will blend with it into a continuous, coordinated pattern. QuickDraw provides the predefined patterns white, black,

gray, ltGray, and dkGray. Any other 64-bit variable or constant can be used as a pattern, too. The data type definition for a pattern is as follows:

```
type Pattern = packed array [0..7] of 0..255;
```

The row width of a pattern is 1 byte.

E.4.4 Cursors

A cursor is a small image that appears on the screen and is controlled by the mouse. (It appears only on the screen, and never in an off-screen bit image.)

A cursor is defined as a 256-bit image, a 16-by-16-bit rectangle (remember that the pixels are not square). The row width of a cursor is 2 bytes. Figure E-8 illustrates four cursors.

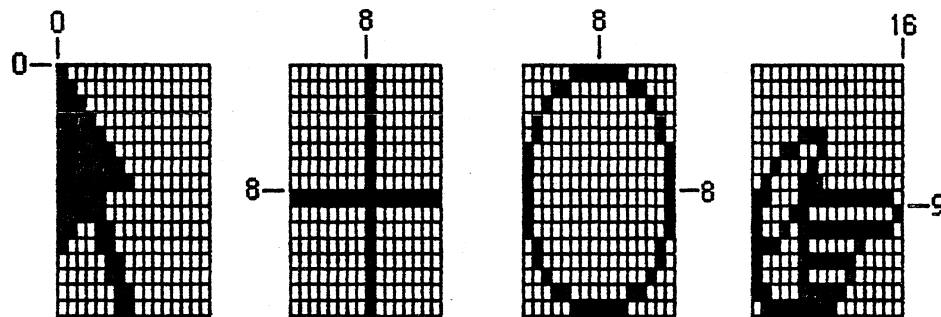


Figure E-8
Cursors

A cursor has three fields: a 16-word data field that contains the image itself, a 16-word mask field that contains information about the screen appearance of each bit of the cursor, and a hotSpot point that aligns the cursor with the position of the mouse.

```
type Cursor = record
    data:    array [0..15] of integer;
    mask:    array [0..15] of integer;
    hotSpot: Point
end;
```

The data for the cursor must begin on a word boundary.

The cursor appears on the screen as a 16-by-16-bit rectangle. The appearance of each bit of the rectangle is determined by the corresponding bits in the data

and mask and, if the mask bit is 0, by the pixel "under" the cursor (the one already on the screen in the same position as this bit of the cursor):

<u>Data</u>	<u>Mask</u>	<u>Resulting pixel on screen</u>
0	1	White
1	1	Black
0	0	Same as pixel under cursor
1	0	Inverse of pixel under cursor

Notice that if all mask bits are 0, the cursor is completely transparent, in that the image under the cursor can still be viewed: pixels under the white part of the cursor appear unchanged, while under the black part of the cursor, black pixels show through as white.

The hotSpot aligns a point in the image (not a bit, a point!) with the mouse position. Imagine the rectangle with corners (0,0) and (16,16) framing the image, as in each of the examples in Figure E-8; the hotSpot is defined in this coordinate system. A hotSpot of (0,0) is at the top left of the image. For the arrow in Figure E-8 to point to the mouse position, (0,0) would be its hotSpot. A hotSpot of (8,8) is in the exact center of the image; the center of the plus sign or oval in Figure E-8 would coincide with the mouse position if (8,8) were the hotSpot for that cursor. Similarly, the hotSpot for the pointing hand would be (16,9).

Whenever you move the mouse, the low-level interrupt-driven mouse routines move the cursor's hotSpot to be aligned with the new mouse position.

QuickDraw supplies a predefined arrow cursor, an arrow pointing north-northwest.

Refer to Appendix F, Hardware Interface, for more information on the mouse and cursor control.

E.5 The Drawing Environment: GrafPort

A grafPort is a complete drawing environment that defines how and where graphic operations will have their effect. It contains all the information about one instance of graphic output that is kept separate from all other instances. You can have many grafPorts open at once, and each one will have its own coordinate system, drawing pattern, background pattern, pen size and location, character font and style, and bitMap in which drawing takes place. You can instantly switch from one port to another. GrafPorts are the structures on which a program builds windows, which are fundamental to the Lisa's "overlapping windows" user interface.

A grafPort is a dynamic data structure, defined as follows:

```

type GrafPtr = ^GrafPort;
   GrafPort = record
       device:      integer;
       portBits:   BitMap;
       portRect:   Rect;
       visRgn:     RgnHandle;
       clipRgn:    RgnHandle;
       bkPat:      Pattern;
       fillPat:    Pattern;
       pnLoc:      Point;
       pnSize:     Point;
       pnMode:     integer;
       pnPat:      Pattern;
       pnVis:      integer;
       txFont:     integer;
       txFace:     Style;
       txMode:     integer;
       txSize:     integer;
       spExtra:    integer;
       fgColor:    longint;
       bkColor:    longint;
       colrBit:    integer;
       patStretch: integer;
       picSave:    QDHandle;
       rgnSave:    QDHandle;
       polySave:   QDHandle;
       grafProcs:  QDProcsPtr
   end;

```

All QuickDraw operations refer to grafPorts via grafPtrs. You create a grafPort with the Pascal procedure NEW and use the resulting pointer in calls to QuickDraw. You could, of course, declare a static var of type grafPort, and obtain a pointer to that static structure (with the @ operator), but as most grafPorts will be used dynamically, their data structures should be dynamic also.

NOTE

You can access all fields and subfields of a `grafPort` normally, but you should not store new values directly into them. QuickDraw has procedures for altering all fields of a `grafPort`, and using these procedures ensures that changing a `grafPort` produces no unusual side effects.

The `device` field of a `grafPort` is the number of the logical output device that the `grafPort` will be using. QuickDraw uses this information, since there are physical differences in the same logical font for different output devices. The default device number is 0, for the Lisa screen.

The `portBits` field is the `bitMap` that points to the bit image to be used by the `grafPort`. All drawing that is done in this `grafPort` will take place in this bit image. The default `bitMap` uses the entire Lisa screen as its bit image, with `rowBytes` of 90 and a boundary rectangle of (0,0,720,364). The `bitMap` may be changed to indicate a different structure in memory: all graphics procedures work in exactly the same way regardless of whether their effects are visible on the screen. A program can, for example, prepare an image to be printed on a printer without ever displaying the image on the screen, or develop a picture in an off-screen `bitMap` before transferring it to the screen. By altering the coordinates of the `portBits.bounds` rectangle, you can change the coordinate system of the `grafPort`; with a QuickDraw procedure call, you can set an arbitrary coordinate system for each `grafPort`, even if the different `grafPorts` all use the same bit image (e.g., the full screen).

The `portRect` field is a rectangle that defines a subset of the `bitMap` for use by the `grafPort`. Its coordinates are in the system defined by the `portBits.bounds` rectangle. All drawing done by the application occurs inside this rectangle. The `portRect` usually defines the "writable" interior area of a window, document, or other object on the screen.

The `visRgn` field indicates the region that is actually visible on the screen. It is reserved for use by future software, and should be treated as read-only. The default `visRgn` is set to the `portRect` (the entire screen).

The `clipRgn` is an arbitrary region that the application can use to limit drawing to any region within the `portRect`. If, for example, you want to draw a half circle on the screen, you can set the `clipRgn` to half the square that would enclose the whole circle, and go ahead and draw the whole circle. Only the half within the `clipRgn` will actually be drawn in the `grafPort`. The default `clipRgn` is set arbitrarily large, and you have full control over its setting. Notice that unlike

the visRgn, the clipRgn affects the image even if it is not displayed on the screen.

Figure E-9 illustrates a typical bitMap (as defined by portBits), portRect, visRgn, and clipRgn.

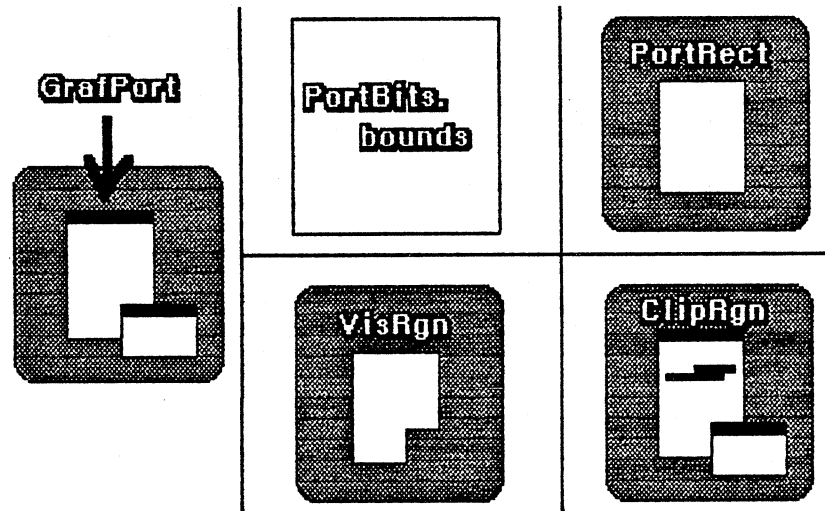


Figure E-9
GrafPort Regions

The bkPat and fillPat fields of a grafPort contain patterns used by certain QuickDraw routines. BkPat is the "background" pattern that is used when an area is erased or when bits are scrolled out of it. When asked to fill an area with a specified pattern, QuickDraw stores the given pattern in the fillPat field and then calls a low-level drawing routine which gets the pattern from that field. The various graphic operations are discussed in detail later in the descriptions of individual QuickDraw routines.

Of the next ten fields, the first five determine characteristics of the graphics pen and the last five determine characteristics of any text that may be drawn; these are described in subsections below.

The fgColor, bkColor, and colrBit fields contain values related to drawing in color, a capability that will be available in the future when Apple supports color output devices for the Lisa. FgColor is the grafPort's foreground color and bkColor is its background color. ColrBit tells the color imaging software which plane of the color picture to draw into. For more information, see Section E.7.2, Drawing in Color.

The `patStretch` field is used during output to a printer to expand patterns if necessary. The application should not change its value.

The `picSave`, `rgnSave`, and `polySave` fields reflect the state of picture, region, and polygon definition, respectively. To define a region, for example, you "open" it, call routines that draw it, and then "close" it. If no region is open, `rgnSave` contains `NIL`; otherwise, it contains a handle to information related to the region definition. The application should not be concerned about exactly what information the handle leads to; you may, however, save the current value of `rgnSave`, set the field to `NIL` to disable the region definition, and later restore it to the saved value to resume the region definition. The `picSave` and `polySave` fields work similarly for pictures and polygons.

Finally, the `grafProcs` field may point to a special data structure that the application stores into if it wants to customize QuickDraw drawing procedures or use QuickDraw in other advanced, highly specialized ways. (For more information, see Section E.10, Customizing QuickDraw Operations.) If `grafProcs` is `NIL`, QuickDraw responds in the standard ways described in this manual.

E.5.1 Pen Characteristics

The `pnLoc`, `pnSize`, `pnMode`, `pnPat`, and `pnVis` fields of a `grafPort` deal with the graphics pen. Each `grafPort` has one and only one graphics pen, which is used for drawing lines, shapes, and text. As illustrated in Figure E-10, the pen has four characteristics: a location, a size, a drawing mode, and a drawing pattern.

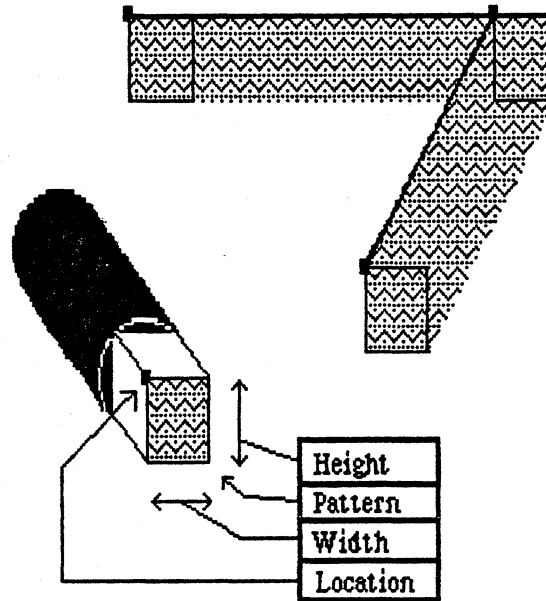


Figure E-10
A Graphics Pen

The pen location is a point in the coordinate system of the grafPort, and is where QuickDraw will begin drawing the next line, shape, or character. It can be anywhere on the coordinate plane: there are no restrictions on the movement or placement of the pen. Remember that the pen location is a point on the coordinate plane, not a pixel in a bit image!

The pen is rectangular in shape, and has a user-definable width and height. The default size is a 1-by-1-bit rectangle; the width and height can range from (0,0) to (32767,32767). If either the pen width or the pen height is less than 1, the pen will not draw on the screen.

- The pen appears as a rectangle with its top left corner at the pen location; it hangs below and to the right of the pen location.

The pnMode and pnPat fields of a grafPort determine how the bits under the pen are affected when lines or shapes are drawn. The pnPat is a pattern that is used as the "ink" in the pen. This pattern, like all other patterns drawn in the grafPort, is always aligned with the port's coordinate system: the top left corner of the pattern is aligned with the top left corner of the portRect, so that adjacent areas of the same pattern will blend into a continuous, coordinated pattern. Five

patterns are predefined (white, black, and three shades of gray); you can also create your own pattern and use it as the `pnPat`. (A utility procedure, called `StuffHex`, allows you to fill patterns easily.)

The `pnMode` field determines how the pen pattern is to affect what's already on the `bitMap` when lines or shapes are drawn. When the pen draws, `QuickDraw` first determines what bits of the `bitMap` will be affected and finds their corresponding bits in the pattern. It then does a bit-by-bit evaluation based on the pen mode, which specifies one of eight boolean operations to perform. The resulting bit is placed into its proper place in the `bitMap`. The pen modes are described in Section E.7.1, *Transfer Modes*.

The `pnVis` field determines the pen's visibility, that is, whether it draws on the screen. For more information, see the descriptions of `HidePen` and `ShowPen` in Section E.9.3, *Pen and Line-Drawing Routines*.

E.5.2 Text Characteristics

The `txFont`, `txFace`, `txMode`, `txSize`, and `spExtra` fields of a `grafPort` determine how text will be drawn--the font, style, and size of characters and how they will be placed on the `bitMap`.

`QuickDraw` can draw characters as quickly and easily as it draws lines and shapes, and in many prepared fonts. Figure E-11 shows two `QuickDraw` characters and some terms you should become familiar with.

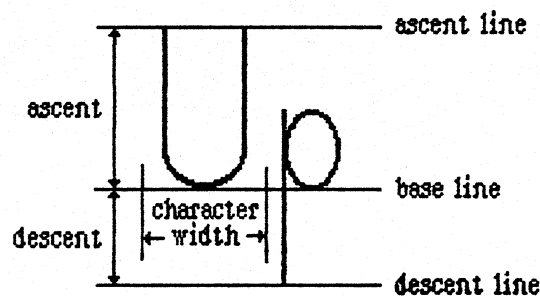


Figure E-11
QuickDraw Characters

`QuickDraw` can display characters in any size, as well as boldfaced, italicized, outlined, or shadowed, all without changing fonts. It can also underline the characters, or draw them closer together or farther apart.

The `txFont` field is a font number that identifies the character font to be used in the `grafPort`. The font number 0 represents the system font, and is the default established by `OpenPort`. The unit "QDSupport" includes definitions of other available font numbers.

A character font is defined as a collection of bit images: these images make up the individual characters of the font. The characters can be of unequal widths,

and they're not restricted to their "cells": the lower curl of a lowercase *j*, for example, can stretch back under the previous character (typographers call this kerning). A font can consist of up to 256 distinct characters, yet not all characters need be defined in a single font. Each font contains a missing symbol to be drawn in case of a request to draw a character that is missing from the font.

The `txFace` field controls the appearance of the font with values from the set defined by the Style data type:

type StyleItem = (bold, italic, underline, outline, shadow,
condense, extend);

Style - set of StyleItem;

You can apply these either alone or in combination (see Figure E-12). Most combinations usually look good only for large fonts.

Normal Characters

Bold Characters

Italic Characters

Underlined Characters xyz

Outlined Characters

Shadowed Characters

Condensed Characters

Extended Characters

Bold Italic Characters

Bold Outlined Underlined

... and in other fonts, too!

Figure E-12
Character Styles

If you specify bold, each character is repeatedly drawn one bit to the right an appropriate number of times for extra thickness.

Italic adds an italic slant to the characters. Character bits above the base line are skewed right; bits below the base line are skewed left.

Underline draws a line below the base line of the characters. If part of a character descends below the base line (as "y" in Figure E-12), the underline is not drawn through the pixel on either side of the descending part.

You may specify either outline or shadow. Outline makes a hollow, outlined character rather than a solid one. With shadow, not only is the character hollow and outlined, but the outline is thickened below and to the right of the character to achieve the effect of a shadow. If you specify bold along with outline or shadow, the hollow part of the character is widened.

Condense and extend affect the horizontal distance between all characters, including spaces. Condense decreases the distance between characters and extend increases it, by an amount which QuickDraw determines is appropriate.

The txMode field controls the way characters are placed on a bit image. It functions much like a pnMode: when a character is drawn, QuickDraw determines which bits of the bit image will be affected, does a bit-by-bit comparison based on the mode, and stores the resulting bits into the bit image. These modes are described in Section E.7.1, Transfer Modes. Only three of them—srcOr, srcXor, and srcBic—should be used for drawing text.

The txSize field specifies the type size for the font, in points (where "point" here is a printing term meaning 1/72 inch). Any size may be specified. If QuickDraw does not have the font in a specified size, it will scale a size it does have as necessary to produce the size desired. A value of 0 in this field directs QuickDraw to choose the size from among those it has for the font; it will choose whichever size is closest to the system font size.

Finally, the spExtra field is useful when a line of characters is to be drawn justified such that it is aligned with both a left and a right margin (sometimes called "full justification"). SpExtra is the number of pixels by which each space character should be widened to fill out the line.

E.6 Coordinates in GrafPorts

Each grafPort has its own local coordinate system. All fields in the grafPort are expressed in these coordinates, and all calculations and actions performed in QuickDraw use the local coordinate system of the currently selected port.

Two things are important to remember:

- Each grafPort maps a portion of the coordinate plane into a similarly-sized portion of a bit image.
- The portBits.bounds rectangle defines the local coordinates for a grafPort.

The top left corner of portBits.bounds is always aligned around the first bit in the bit image; the coordinates of that corner "anchor" a point on the grid to that bit in the bit image. This forms a common reference point for multiple grafPorts using the same bit image (such as the screen). Given a portBits.bounds rectangle for each port, you know that their top left corners coincide.

The interrelationship between the portBits.bounds and portRect rectangles is very important. As the portBits.bounds rectangle establishes a coordinate system for the port, the portRect rectangle indicates the section of the

coordinate plane (and thus the bit image) that will be used for drawing. The portRect usually falls inside the portBits.bounds rectangle, but it's not required to do so.

When a new grafPort is created, its bitMap is set to point to the entire Lisa screen, and both the portBits.bounds and the portRect rectangles are set to 720-by-364-bit rectangles, with the point (0,0) at the top left corner of the screen.

You can redefine the local coordinates of the top left corner of the grafPort's portRect, using the SetOrigin procedure. This changes the local coordinate system of the grafPort, recalculating the coordinates of all points in the grafPort to be relative to the new corner coordinates. For example, consider these procedure calls:

```
SetPort(gamePort);
SetOrigin(40,80);
```

The call to SetPort sets the current grafPort to gamePort; the call to SetOrigin changes the local coordinates of the top left corner of that port's portRect to (40,80) (see Figure E-13).

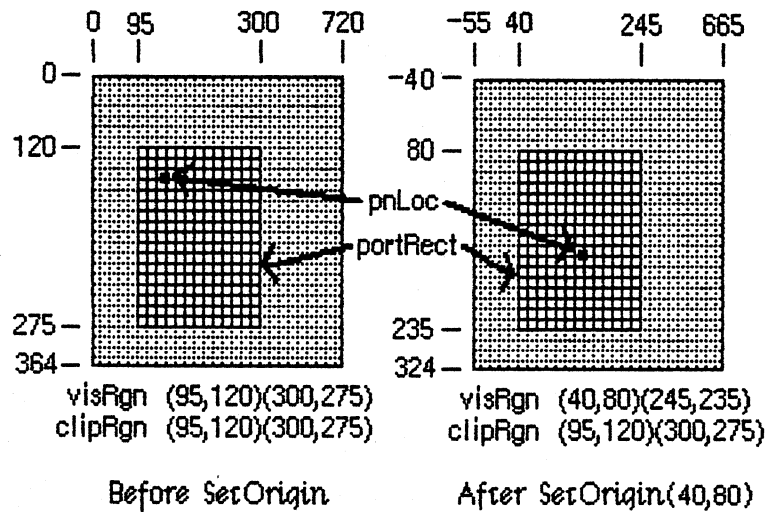


Figure E-13
Changing Local Coordinates

This recalculates the coordinate components of the following elements:

```
gamePort^.portBits.bounds      gamePort^.portRect
gamePort^.visRgn
```

These elements are always kept "in sync", so that all calculations, comparisons, or operations that seem right, work right.

Notice that when the local coordinates of a grafPort are offset, the visRgn of that port is offset also, but the clipRgn is not. A good way to think of it is that if a document is being shown inside a grafPort, the document "sticks" to the coordinate system, and the port's structure "sticks" to the screen. Suppose, for example, that the visRgn and clipRgn in Figure E-13 before SetOrigin are the same as the portRect, and a document is being shown. After the SetOrigin call, the top left corner of the clipRgn is still (95,120), but this location has moved down and to the right, and the location of the pen within the document has similarly moved. The locations of portBits.bounds, portRect, and visRgn did not change; their coordinates were offset. As always, the top left corner of portBits.bounds remains aligned around the first bit in the bit image (the first pixel on the screen).

If you are moving, comparing, or otherwise dealing with mathematical items in different grafPorts (for example, finding the intersection of two regions in two different grafPorts), you must adjust to a common coordinate system before you perform the operation. A QuickDraw procedure, LocalToGlobal, lets you convert a point's local coordinates to a global system where the top left corner of the bit image is (0,0); by converting the various local coordinates to global coordinates, you can compare and mix them with confidence. For more information, see the description of this procedure in Section E.9.17, Calculations with Points.

E.7 General Discussion of Drawing

Drawing occurs:

- Always inside a grafPort, in the bit image and coordinate system defined by the grafPort's bitMap.
- Always within the intersection of the grafPort's portBits.bounds and portRect, and clipped to its visRgn and clipRgn.
- Always at the grafPort's pen location.
- Usually with the grafPort's pen size, pattern, and mode.

With QuickDraw procedures, you can draw lines, shapes, and text. Shapes include rectangles, ovals, rounded-corner rectangles, wedge-shaped sections of ovals, regions, and polygons.

Lines are defined by two points: the current pen location and a destination location. When drawing a line, QuickDraw moves the top left corner of the pen

along the mathematical trajectory from the current location to the destination. The pen hangs below and to the right of the trajectory (see Figure E-14).

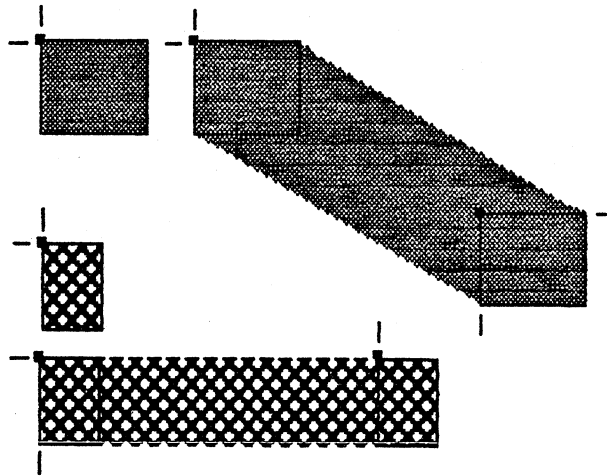


Figure E-14
Drawing Lines

NOTE

No mathematical element (such as the pen location) is ever affected by clipping; clipping only determines what appears where in the bit image. If you draw a line to a location outside your grafPort, the pen location will move there, but only the portion of the line that is inside the port will actually be drawn. This is true for all drawing procedures.

Rectangles, ovals, and rounded-corner rectangles are defined by two corner points. The shapes always appear inside the mathematical rectangle defined by the two points. A region is defined in a more complex manner, but also appears only within the rectangle enclosing it. Remember, these enclosing rectangles have infinitely thin borders and are not visible on the screen.

As illustrated in Figure E-15, shapes may be drawn either solid (filled in with a pattern) or framed (outlined and hollow).

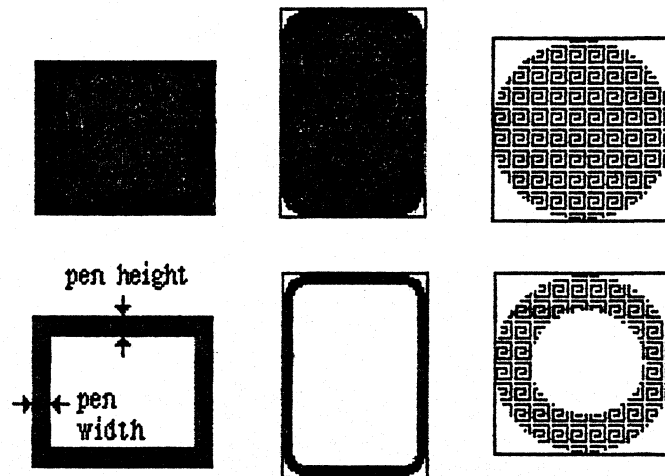


Figure E-15
Solid Shapes and Framed Shapes

In the case of framed shapes, the outline appears completely within the enclosing rectangle--with one exception--and the vertical and horizontal thickness of the outline is determined by the pen size. The exception is polygons, as discussed in "Pictures and Polygons" below.

The pen pattern is used to fill in the bits that are affected by the drawing operation. The pen mode defines how those bits are to be affected by directing QuickDraw to apply one of eight boolean operations to the bits in the shape and the corresponding pixels on the screen.

Text drawing does not use the `pnSize`, `pnPat`, or `pnMode`, but it does use the `pnLoc`. Each character is placed to the right of the current pen location, with the left end of its base line at the pen's location. The pen is moved to the right to the location where it will draw the next character. No wrap or carriage return is performed automatically.

The method QuickDraw uses in placing text is controlled by a mode similar to the pen mode. This is explained in Section E.7.1, Transfer Modes. Clipping of text is performed in exactly the same manner as all other clipping in QuickDraw.

E.7.1 Transfer Modes

When lines or shapes are drawn, the `pnMode` field of the `grafPort` determines how the drawing is to appear in the port's bit image; similarly, the `txMode` field determines how text is to appear. There is also a QuickDraw procedure that transfers a bit image from one bitMap to another, and this procedure has a mode

parameter that determines the appearance of the result. In all these cases, the mode, called a transfer mode, specifies one of eight boolean operations: for each bit in the item to be drawn, QuickDraw finds the corresponding bit in the destination bit image, performs the boolean operation on the pair of bits, and stores the resulting bit into the bit image.

There are two types of transfer mode:

- Pattern transfer modes, for drawing lines or shapes with a pattern.
- Source transfer modes, for drawing text or transferring any bit image between two bitMaps.

For each type of mode, there are four basic operations--Copy, Or, Xor, and Bic. The Copy operation simply replaces the pixels in the destination with the pixels in the pattern or source, "painting" over the destination without regard for what is already there. The Or, Xor, and Bic operations leave the destination pixels under the white part of the pattern or source unchanged, and differ in how they affect the pixels under the black part: Or replaces those pixels with black pixels, thus "overlaying" the destination with the black part of the pattern or source; Xor inverts the pixels under the black part; and Bic erases them to white.

Each of the basic operations has a variant in which every pixel in the pattern or source is inverted before the operation is performed, giving eight operations in all. Each mode is defined by name as a constant in QuickDraw (see Figure E-16).

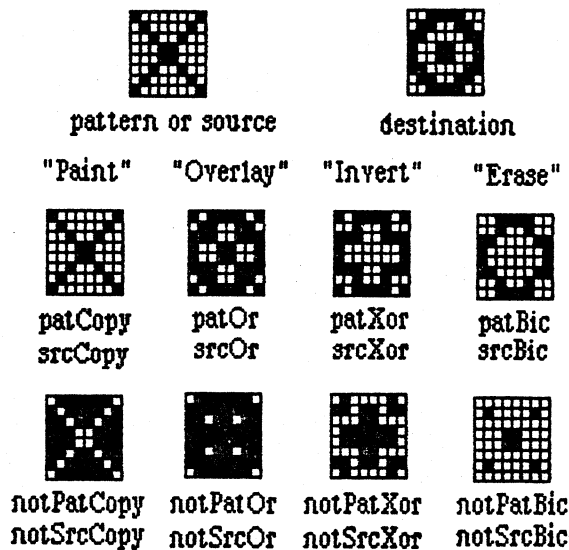


Figure E-16
Transfer Modes

Pattern transfer mode	Source transfer mode	Action on each pixel in destination:	
		If black pixel in pattern or source	If white pixel in pattern or source
patCopy	srcCopy	Force black	Force white
patOr	srcOr	Force black	Leave alone
patXor	srcXor	Invert	Leave alone
patBic	srcBic	Force white	Leave alone
notPatCopy	notSrcCopy	Force white	Force black
notPatOr	notSrcOr	Leave alone	Force black
notPatXor	notSrcXor	Leave alone	Invert
notPatBic	notSrcBic	Leave alone	Force white

E.7.2 Drawing in Color

Currently you can only look at QuickDraw output on a black-and-white screen or printer. Eventually, however, Apple will support color output devices. If you want to set up your application now to produce color output in the future, you can do so by using QuickDraw procedures to set the foreground color and the background color. Eight standard colors may be specified with the following predefined constants: `blackColor`, `whiteColor`, `redColor`, `greenColor`, `blueColor`, `cyanColor`, `magentaColor`, and `yellowColor`. Initially, the foreground color is `blackColor` and the background color is `whiteColor`. If you specify a color other than `whiteColor`, it will appear as black on a black-and-white output device.

To apply the table in the "Transfer Modes" section above to drawing in color, make the following translation: where the table shows "Force black", read "Force foreground color", and where it shows "Force white", read "Force background color". When you eventually receive the color output device, you'll find out the effect of inverting a color on it.

NOTE

QuickDraw can support output devices that have up to 32 bits of color information per pixel. A color picture may be thought of, then, as having up to 32 planes. At any one time, QuickDraw draws into only one of these planes. A QuickDraw routine called by the color-imaging software specifies which plane.

E.8 Pictures and Polygons

QuickDraw lets you save a sequence of drawing commands and "play them back" later with a single procedure call. There are two such mechanisms: one for drawing any picture to scale in a destination rectangle that you specify, and another for drawing polygons in all the ways you can draw other shapes in QuickDraw.

E.8.1 Pictures

A picture in QuickDraw is a transcript of calls to routines which draw something—anything—on a bitMap. Pictures make it easy for one program to draw something defined in another program, with great flexibility and without knowing the details about what's being drawn.

For each picture you define, you specify a rectangle that surrounds the picture; this rectangle is called the picture frame. When you later call the procedure that draws the saved picture, you supply a destination rectangle, and QuickDraw scales the picture so that its frame is completely aligned with the destination rectangle. Thus, the picture may be expanded or shrunk to fit its destination rectangle. For example, if the picture is a circle inside a square picture frame, and the destination rectangle is not square, the picture is drawn as an oval.

Since a picture may include any sequence of drawing commands, its data structure is a variable-length entity. It consists of two fixed fields followed by a variable-length data field:

```

type Picture = record
    picSize: integer;
    picFrame: Rect;
    {picture definition data}
end;
```

The picSize field contains the size, in bytes, of the picture variable. The picFrame field is the picture frame which surrounds the picture and gives a frame of reference for scaling when the picture is drawn. The rest of the structure contains a compact representation of the drawing commands that define the picture.

All pictures are accessed through handles, which point to one master pointer which in turn points to the picture.

```

type PicPtr    = ^Picture;
   PicHandle = ^PicPtr;
```

To define a picture, you call a QuickDraw function that returns a picHandle and then call the routines that draw the picture. There is a procedure to call when you've finished defining the picture, and another for when you're done with the picture altogether.

QuickDraw also allows you to intersperse picture comments with the definition of a picture. These comments, which do not affect the picture's appearance, may be used to provide additional information about the picture when it's played back. This is especially valuable when pictures are transmitted from one application to another. There are two standard types of comment which, like

parentheses, serve to group drawing commands together (such as all the commands that draw a particular part of a picture):

```
const  picLParen = 0;  
       picRParen = 1;
```

The application defining the picture can use these standard comments as well as comments of its own design.

To include a comment in the definition of a picture, the application calls a QuickDraw procedure that specifies the comment with three parameters: the comment kind, which identifies the type of comment; a handle to additional data if desired; and the size of the additional data, if any. When playing back a picture, QuickDraw passes any comments in the picture's definition to a low-level procedure accessed indirectly through the `grafProcs` field of the `grafPort` (see Section E.10, Customizing QuickDraw Operations for more information). To process comments, the application must include a procedure to do the processing and store a pointer to it in the data structure pointed to by the `grafProcs` field.

NOTE

The standard low-level procedure for processing picture comments simply ignores all comments.

E.8.2 Polygons

Polygons are similar to pictures in that you define them by a sequence of calls to QuickDraw routines. They are also similar to other shapes that QuickDraw knows about, since there is a set of procedures for performing graphic operations and calculations on them.

A polygon is simply any sequence of connected lines (see Figure E-17). You define a polygon by moving to the starting point of the polygon and drawing lines from there to the next point, from that point to the next, and so on.

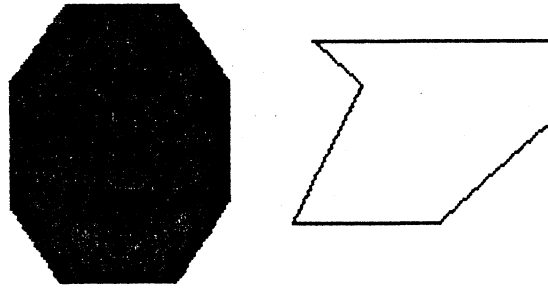


Figure E-17
Polygons

The data structure for a polygon is a variable-length entity. It consists of two fixed fields followed by a variable-length array:

```

type Polygon = record
    polySize: integer;
    polyBBox: Rect;
    polyPoints: array [0..0] of Point
end;
```

The `polySize` field contains the size, in bytes, of the polygon variable. The `polyBBox` field is a rectangle which just encloses the entire polygon. The `polyPoints` array expands as necessary to contain the points of the polygon-- the starting point followed by each successive point to which a line is drawn.

Like pictures and regions, polygons are accessed through handles.

```

type PolyPtr = ^Polygon;
    PolyHandle = ^PolyPtr;
```

To define a polygon, you call a QuickDraw function that returns a `polyHandle` and then form the polygon by calling procedures that draw lines. You call a procedure when you've finished defining the polygon, and another when you're done with the polygon altogether.

Just as for other shapes that QuickDraw knows about, there is a set of graphic operations on polygons to draw them on the screen. QuickDraw draws a polygon by moving to the starting point and then drawing lines to the remaining points in succession, just as when the routines were called to define the polygon. In this sense it "plays back" those routine calls. As a result, polygons are not treated exactly the same as other QuickDraw shapes. For example, the procedure that frames a polygon draws outside the actual boundary of the polygon, because

QuickDraw line-drawing routines draw below and to the right of the pen location. The procedures that fill a polygon with a pattern, however, stay within the boundary of the polygon; they also add an additional line between the ending point and the starting point if those points are not the same, to complete the shape.

There is also a difference in the way QuickDraw scales a polygon and a similarly-shaped region if it's being drawn as part of a picture: when stretched, a slanted line is drawn more smoothly if it's part of a polygon rather than a region. You may find it helpful to keep in mind the conceptual difference between polygons and regions: a polygon is treated more as a continuous shape, a region more as a set of bits.

E.9 QuickDraw Routines

This section describes all the procedures and functions in QuickDraw, their parameters, and their operation. They are presented in their Pascal form; for information on using them from assembly language, see Section E.11, Using QuickDraw from Assembly Language.

E.9.1 GrafPort Routines

Procedure `InitGraf` (`globalPtr: QDPtr`);

Call `InitGraf` once and only once at the beginning of your program to initialize QuickDraw. It initializes the QuickDraw global variables listed below.

<u>Variable</u>	<u>Type</u>	<u>Initial setting</u>
<code>thePort</code>	<code>GrafPtr</code>	NIL
<code>white</code>	<code>Pattern</code>	all-white pattern
<code>black</code>	<code>Pattern</code>	all-black pattern
<code>gray</code>	<code>Pattern</code>	50% gray pattern
<code>ltGray</code>	<code>Pattern</code>	25% gray pattern
<code>dkGray</code>	<code>Pattern</code>	75% gray pattern
<code>arrow</code>	<code>Cursor</code>	pointing arrow cursor
<code>screenBit</code>	<code>BitMap</code>	Lisa screen, (0,0,720,364)
<code>randSeed</code>	<code>longint</code>	1

The `globalPtr` parameter tells QuickDraw where to store its global variables, beginning with `thePort`. From Pascal programs, this parameter should always be set to `@thePort`; assembly language programmers may choose any location, as long as it can accommodate the number of bytes specified by `GRAFSIZE` in `GRAFTYPES.TEXT` (see Section E.11, Using QuickDraw from Assembly Language).

NOTE

To initialize the cursor, call `InitCursor` (described in Section E.9.2, `Cursor-Handling Routines`).

Procedure `OpenPort` (`gp: GrafPtr`);

`OpenPort` allocates space for the given `grafPort`'s `visRgn` and `clipRgn`, initializes the fields of the `grafPort` as indicated below, and makes the `grafPort` the current port (see `SetPort`). You must call `OpenPort` before using any `grafPort`; first perform a `NEW` to create a `grafPtr` and then use that `grafPtr` in the `OpenPort` call.

<u>Field</u>	<u>Type</u>	<u>Initial setting</u>
<code>device</code>	integer	0 (Lisa screen)
<code>portBits</code>	BitMap	<code>screenBits</code> (see <code>InitGraf</code>)
<code>portRect</code>	Rect	<code>screenBits.bounds</code> (0,0,720,364)
<code>visRgn</code>	RgnHandle	handle to the rectangular region (0,0,720,364)
<code>clipRgn</code>	RgnHandle	handle to the rectangular region (-30000,-30000,30000,30000)
<code>bkPat</code>	Pattern	white
<code>fillPat</code>	Pattern	black
<code>pnLoc</code>	Point	(0,0)
<code>pnSize</code>	Point	(1,1)
<code>pnMode</code>	integer	<code>patCopy</code>
<code>pnPat</code>	Pattern	black
<code>pnVis</code>	integer	0 (visible)
<code>txFont</code>	integer	0 (system font)
<code>txFace</code>	Style	normal
<code>txMode</code>	integer	<code>srcOr</code>
<code>txSize</code>	integer	0 (QuickDraw decides)
<code>spExtra</code>	integer	0
<code>fgColor</code>	longint	<code>blackColor</code>
<code>bkColor</code>	longint	<code>whiteColor</code>
<code>colrBit</code>	integer	0
<code>patStretch</code>	integer	0
<code>picSave</code>	QDHandle	NIL
<code>rgnSave</code>	QDHandle	NIL
<code>polySave</code>	QDHandle	NIL
<code>grafProcs</code>	QDProcsPtr	NIL

Procedure InitPort (gp: GrafPtr);

Given a pointer to a grafPort that has been opened with OpenPort, InitPort reinitializes the fields of the grafPort and makes it the current port (if it's not already).

NOTE

InitPort does everything OpenPort does except allocate space for the visRgn and clipRgn.

Procedure ClosePort (gp: GrafPtr);

ClosePort deallocates the space occupied by the given grafPort's visRgn and clipRgn. When you are completely through with a grafPort, call this procedure and then dispose of the grafPort (with a DISPOSE of the grafPtr).

WARNING

If you do not call ClosePort before disposing of the grafPort, the memory used by the visRgn and clipRgn will be unrecoverable.

WARNING

After calling ClosePort, be sure not to use any copies of the visRgn or clipRgn handles that you may have made.

Procedure SetPort (gp: GrafPtr);

SetPort sets the grafPort indicated by gp to be the current port. The global pointer thePort always points to the current port. All QuickDraw drawing routines affect the bitMap thePort^.portBits and use the local coordinate system of thePort^. Note that OpenPort and InitPort do a SetPort to the given port.

WARNING

Never do a SetPort to a port that has not been opened with OpenPort.

Each port possesses its own pen and text characteristics which remain unchanged when the port is not selected as the current port.

Procedure GetPort (var gp: GrafPtr);

GetPort returns a pointer to the current grafPort. If you have a program that draws into more than one grafPort, it's extremely useful to have each procedure

save the current grafPort (with GetPort), set its own grafPort, do drawing or calculations, and then restore the previous grafPort (with SetPort). The pointer to the current grafPort is also available through the global pointer thePort, but you may prefer to use GetPort for better readability of your program text. For example, a procedure could do a GetPort(savePort) before setting its own grafPort and a SetPort(savePort) afterwards to restore the previous port.

Procedure GrafDevice (device: integer);

GrafDevice sets thePort^.device to the given number, which identifies the logical output device for this grafPort. QuickDraw uses this information. The initial device number is 0, which represents the Lisa screen.

Procedure SetPortBits (bm: BitMap);

SetPortBits sets thePort^.portBits to any previously defined bitMap. This allows you to perform all normal drawing and calculations on a buffer other than the Lisa screen--for example, a 640-by-8 output buffer for a C. Itoh printer, or a small off-screen image for later "stamping" onto the screen.

Remember to prepare all fields of the bitMap before you call SetPortBits.

Procedure PortSize (width,height: integer);

PortSize changes the size of the current grafPort's portRect. This does not affect the screen; it merely changes the size of the "active area" of the grafPort.

The top left corner of the portRect remains at its same location; the width and height of the portRect are set to the given width and height. In other words, PortSize moves the bottom right corner of the portRect to a position relative to the top left corner.

PortSize does not change the clipRgn or the visRgn, nor does it affect the local coordinate system of the grafPort: it changes only the portRect's width and height. Remember that all drawing occurs only in the intersection of the portBits.bounds and the portRect, clipped to the visRgn and the clipRgn.

Procedure MovePortTo (leftGlobal,topGlobal: integer);

MovePortTo changes the position of the current grafPort's portRect. This does not affect the screen; it merely changes the location at which subsequent drawing inside the port will appear.

The `leftGlobal` and `topGlobal` parameters set the distance between the top left corner of `portBits.bounds` and the top left corner of the new `portRect`. For example,

```
MovePortTo(360,182);
```

will move the top left corner of the `portRect` to the center of the screen (if `portBits` is the Lisa screen) regardless of the local coordinate system.

Like `PortSize`, `MovePortTo` does not change the `clipRgn` or the `visRgn`, nor does it affect the local coordinate system of the `grafPort`.

Procedure `SetOrigin` (`h,v`: integer);

`SetOrigin` changes the local coordinate system of the current `grafPort`. This does not affect the screen; it does, however, affect where subsequent drawing and calculation will appear in the `grafPort`. `SetOrigin` updates the coordinates of the `portBits.bounds`, the `portRect`, and the `visRgn`. All subsequent drawing and calculation routines will use the new coordinate system.

The `h` and `v` parameters set the coordinates of the top left corner of the `portRect`. All other coordinates are calculated from this point. All relative distances among any elements in the port will remain the same; only their absolute local coordinates will change.

NOTE

`SetOrigin` does not update the coordinates of the `clipRgn` or the pen; these items stick to the coordinate system (unlike the port's structure, which sticks to the screen).

`SetOrigin` is useful for adjusting the coordinate system after a scrolling operation. (See `ScrollRect` in Section E.9.13, Bit Transfer Operations.)

Procedure `SetClip` (`rgn`: `RgnHandle`);

`SetClip` changes the clipping region of the current `grafPort` to a region equivalent to the given region. Note that this does not change the region handle, but affects the clipping region itself. Since `SetClip` makes a copy of the given region, any subsequent changes you make to that region will not affect the clipping region of the port.

You can set the clipping region to any arbitrary region, to aid you in drawing inside the `grafPort`. The initial `clipRgn` is an arbitrarily large rectangle.

Procedure GetClip (rgn: RgnHandle);

GetClip changes the given region to a region equivalent to the clipping region of the current grafPort. This is the reverse of what SetClip does. Like SetClip, it does not change the region handle.

Procedure ClipRect (r: Rect);

ClipRect changes the clipping region of the current grafPort to a rectangle equivalent to given rectangle. Note that this does not change the region handle, but affects the region itself.

Procedure BackPat (pat: Pattern);

BackPat sets the background pattern of the current grafPort to the given pattern. The background pattern is used in ScrollRect and in all QuickDraw routines that perform an "erase" operation.

E.9.2 Cursor-Handling Routines

Additional information on cursor handling can be found in Appendix F, Hardware Interface.

Procedure InitCursor;

InitCursor sets the current cursor to the predefined arrow cursor, an arrow pointing north-northwest, and sets the cursor level to 0, making the cursor visible. The cursor level, which is initialized to 0 when the system is booted, keeps track of the number of times the cursor has been hidden to compensate for nested calls to HideCursor and ShowCursor (below).

Before you call InitCursor, the cursor is undefined (or, if set by a previous process, it's whatever that process set it to).

Procedure SetCursor (crsr: Cursor);

SetCursor sets the current cursor to the 16-by-16-bit image in crsr. If the cursor is hidden, it remains hidden and will attain the new appearance when it's uncovered; if the cursor is already visible, it changes to the new appearance immediately.

The cursor image is initialized by InitCursor to a north-northwest arrow, visible on the screen. There is no way to retrieve the current cursor image.

Procedure HideCursor;

HideCursor removes the cursor from the screen, restoring the bits under it, and decrements the cursor level (which InitCursor initialized to 0). Every call to HideCursor should be balanced by a subsequent call to ShowCursor.

Procedure ShowCursor;

ShowCursor increments the cursor level, which may have been decremented by HideCursor, and displays the cursor on the screen if the level becomes 0. A call to ShowCursor should balance each previous call to HideCursor. The level is not incremented beyond 0, so extra calls to ShowCursor don't hurt.

If the cursor has been changed (with SetCursor) while hidden, ShowCursor presents the new cursor.

The cursor is initialized by InitCursor to a north-northwest arrow, not hidden.

Procedure ObscureCursor;

ObscureCursor hides the cursor until the next time the mouse is moved. Unlike HideCursor, it has no effect on the cursor level and must not be balanced by a call to ShowCursor.

E.9.3 Pen and Line-Drawing Routines

The pen and line-drawing routines all depend on the coordinate system of the current grafPort. Remember that each grafPort has its own pen; if you draw in one grafPort, change to another, and return to the first, the pen will have remained in the same location.

Procedure HidePen;

HidePen decrements the current grafPort's pnVis field, which is initialized to 0 by OpenPort; whenever pnVis is negative, the pen does not draw on the screen. PnVis keeps track of the number of times the pen has been hidden to compensate for nested calls to HidePen and ShowPen (below). HidePen is called by OpenRgn, OpenPicture, and OpenPoly so that you can define regions, pictures, and polygons without drawing on the screen.

Procedure ShowPen;

ShowPen increments the current grafPort's pnVis field, which may have been decremented by HidePen; if pnVis becomes 0, QuickDraw resumes drawing on the screen. Extra calls to ShowPen will increment pnVis beyond 0, so every call to ShowPen should be balanced by a subsequent call to HidePen. ShowPen is called by CloseRgn, ClosePicture, and ClosePoly.

Procedure GetPen (var pt: Point);

GetPen returns the current pen location, in the local coordinates of the current grafPort.

Procedure GetPenState (var pnState: PenState);

GetPenState saves the pen location, size, pattern, and mode in a storage variable, to be restored later with SetPenState (below). This is useful when calling short subroutines that operate in the current port but must change the graphics pen: each such procedure can save the pen's state when it's called, do whatever it needs to do, and restore the previous pen state immediately before returning.

The PenState data type is not useful for anything except saving the pen's state.

Procedure SetPenState (pnState: PenState);

SetPenState sets the pen location, size, pattern, and mode in the current grafPort to the values stored in pnState. This is usually called at the end of a procedure that has altered the pen parameters and wants to restore them to their state at the beginning of the procedure. (See GetPenState, above.)

Procedure PenSize (width,height: integer);

PenSize sets the dimensions of the graphics pen in the current grafPort. All subsequent calls to Line, LineTo, and the procedures that draw framed shapes in the current grafPort will use the new pen dimensions.

The pen dimensions can be accessed in the variable thePort[^].pnSize, which is of type Point. If either of the pen dimensions is set to a negative value, the pen assumes the dimensions (0,0) and no drawing is performed. For a discussion of how the pen draws, see the "General Discussion of Drawing" earlier in this manual.

Procedure PenMode (mode: integer);

PenMode sets the transfer mode through which the pnPat is transferred onto the bitMap when lines or shapes are drawn. The mode may be any one of the pattern transfer modes:

patCopy	patXor	notPatCopy	notPatXor
patOr	patBic	notPatOr	notPatBic

If the mode is one of the source transfer modes (or negative), no drawing is performed. The current pen mode can be obtained in the variable

thePort[^].pnMode. The initial pen mode is patCopy, in which the pen pattern is copied directly to the bitMap.

Procedure PenPat (pat: Pattern);

PenPat sets the pattern that is used by the pen in the current grafPort. The standard patterns white, black, gray, ltGray, and dkGray are predefined; the initial pnPat is black. The current pen pattern can be obtained in the variable thePort[^].pnPat, and this value can be assigned (but not compared!) to any other variable of type Pattern.

Procedure PenNormal;

PenNormal resets the initial state of the pen in the current grafPort, as follows:

<u>Field</u>	<u>Setting</u>
pnSize	(1,1)
pnMode	patCopy
pnPat	black

The pen location is not changed.

Procedure MoveTo (h,v: integer);

MoveTo moves the pen to location (h,v) in the local coordinates of the current grafPort. No drawing is performed.

Procedure Move (dh,dv: integer);

This procedure moves the pen a distance of dh horizontally and dv vertically from its current location; it calls MoveTo(h+dh,v+dv), where (h,v) is the current location. The positive directions are to the right and down. No drawing is performed.

Procedure LineTo (h,v: integer);

LineTo draws a line from the current pen location to the location specified (in local coordinates) by h and v. The new pen location is (h,v) after the line is drawn. See the general discussion of drawing.

If a region or polygon is open and being formed, its outline is infinitely thin and is not affected by the pnSize, pnMode, or pnPat. (See OpenRgn and OpenPoly.)

Procedure Line (dh,dv: integer);

This procedure draws a line to the location that is a distance of dh horizontally and dv vertically from the current pen location; it calls LineTo(h+dh,v+dv), where

(h,v) is the current location. The positive directions are to the right and down. The pen location becomes the coordinates of the end of the line after the line is drawn. See the general discussion of drawing.

If a region or polygon is open and being formed, its outline is infinitely thin and is not affected by the pnSize, pnMode, or pnPat. (See OpenRgn and OpenPoly.)

E.9.4 Text-Drawing Routines

Each grafPort has its own text characteristics, and all these procedures deal with those of the current port.

Procedure TextFont (font: integer);

TextFont sets the current grafPort's font (thePort[^].txFont) to the given font number. The initial font number is 0, which represents the system font.

Procedure TextFace (face: Style);

TextFace sets the current grafPort's character style (thePort[^].txFace). The Style data type allows you to specify a set of one or more of the following predefined constants: bold, italic, underline, outline, shadow, condense, and extend. For example:

TextFace({bold});	{bold}
TextFace({bold,italic});	{bold and italic}
TextFace(thePort [^] .txFace+{bold});	{whatever it was plus bold}
TextFace(thePort [^] .txFace-[bold]);	{whatever it was but not bold}
TextFace(0);	{normal}

Procedure TextMode (mode: integer);

TextMode sets the current grafPort's transfer mode for drawing text (thePort[^].txMode). The mode should be srcOr, srcXor, or srcBic. The initial transfer mode for drawing text is srcOr.

Procedure TextSize (size: integer);

TextSize sets the current grafPort's type size (thePort[^].txSize) to the given number of points. Any size may be specified, but the result will look best if QuickDraw has the font in that size (otherwise it will scale a size it does have). The next best result will occur if the given size is an even multiple of a size available for the font. If 0 is specified, QuickDraw will choose one of the available sizes--whichever is closest to the system font size. The initial txSize setting is 0.

Procedure SpaceExtra (extra: integer);

SpaceExtra sets the current grafPort's spExtra field, which specifies the number of pixels by which to widen each space in a line of text. This is useful when text is being fully justified (that is, aligned with both a left and a right margin). Consider, for example, a line that contains three spaces; if there would normally be six pixels between the end of the line and the right margin, you would call SpaceExtra(2) to print the line with full justification. The initial spExtra setting is 0.

NOTE

SpaceExtra will also take a negative argument, but be careful not to narrow spaces so much that the text is unreadable.

Procedure DrawChar (ch: char);

DrawChar places the given character to the right of the pen location, with the left end of its base line at the pen's location, and advances the pen accordingly. If the character is not in the font, the font's missing symbol is drawn.

Procedure DrawString (s: Str255);

DrawString performs consecutive calls to DrawChar for each character in the supplied string; the string is placed beginning at the current pen location and extending right. No formatting (carriage returns, line feeds, etc.) is performed by QuickDraw. The pen location ends up to the right of the last character in the string.

Procedure DrawText (textBuf: QDPtr; firstByte,byteCount: integer);

DrawText draws text from an arbitrary structure in memory specified by textBuf, starting firstByte bytes into the structure and continuing for byteCount bytes. The string of text is placed beginning at the current pen location and extending right. No formatting (carriage returns, line feeds, etc.) is performed by QuickDraw. The pen location ends up to the right of the last character in the string.

Function CharWidth (ch: char): integer;

CharWidth returns the value that will be added to the pen horizontal coordinate if the specified character is drawn. CharWidth includes the effects of the stylistic variations set with TextFace; if you change these after determining the character width but before actually drawing the character, the predetermined

width may not be correct. If the character is a space, CharWidth also includes the effect of SpaceExtra.

Function StringWidth(s: Str255): integer;

StringWidth returns the width of the given text string, which it calculates by adding the CharWidths of all the characters in the string (see above). This value will be added to the pen horizontal coordinate if the specified string is drawn.

Function TextWidth (textBuf: QDPtr; firstByte,byteCount: integer) : integer;

TextWidth returns the width of the text stored in the arbitrary structure in memory specified by textBuf, starting firstByte bytes into the structure and continuing for byteCount bytes. It calculates the width by adding the CharWidths of all the characters in the text. (See CharWidth, above.)

Procedure GetFontInfo (var info: FontInfo);

GetFontInfo returns the following information about the current grafPort's character font, taking into consideration the style and size in which the characters will be drawn: the ascent, descent, maximum character width (the greatest distance the pen will move when a character is drawn), and leading (the vertical distance between the descent line and the ascent line below it), all in pixels. The FontInfo data structure is defined as:

```
type FontInfo = record
    ascent: integer;
    descent: integer;
    widMax: integer;
    leading: integer
end;
```

E.9.5 Drawing in Color

These routines will enable applications to do color drawing in the future when Apple supports color output devices for the Lisa. All nonwhite colors will appear as black on black-and-white output devices.

Procedure ForeColor (color: longint);

ForeColor sets the foreground color for all drawing in the current grafPort (thePort^.fgColor) to the given color. The following standard colors are predefined: blackColor, whiteColor, redColor, greenColor, blueColor, cyanColor, magentaColor, and yellowColor. The initial foreground color is blackColor.

Procedure BackColor (color: longint);

BackColor sets the background color for all drawing in the current grafPort (thePort^.bkColor) to the given color. Eight standard colors are predefined (see ForeColor above). The initial background color is whiteColor.

Procedure ColorBit (whichBit: integer);

ColorBit is called by printing software for a color printer, or other color-imaging software, to set the current grafPort's colrBit field to whichBit; this tells QuickDraw which plane of the color picture to draw into. QuickDraw will draw into the plane corresponding to bit number whichBit. Since QuickDraw can support output devices that have up to 32 bits of color information per pixel, the possible range of values for whichBit is 0 through 31. The initial value of the colrBit field is 0.

E.9.6 Calculations with Rectangles

Calculation routines are independent of the current coordinate system; a calculation will operate the same regardless of which grafPort is active.

NOTE

Remember that if the parameters to one of the calculation routines were defined in different grafPorts, you must first adjust them to be in the same coordinate system. If you do not adjust them, the result returned by the routine may be different from what you see on the screen. To adjust to a common coordinate system, see LocalToGlobal and GlobalToLocal in Section E.9.17, Calculations with Points.

Procedure SetRect (var r: Rect; left,top,right,bottom: integer);

SetRect assigns the four boundary coordinates to the rectangle. The result is a rectangle with coordinates (left,top,right,bottom).

This procedure is supplied as a utility to help you shorten your program text. If you want a more readable text at the expense of length, you can assign integers (or points) directly into the rectangle's fields. There is no significant code size or execution speed advantage to either method; one's just easier to write, and the other's easier to read.

Procedure OffsetRect (var r: Rect; dh,dv: integer);

OffsetRect moves the rectangle by adding dh to each horizontal coordinate and dv to each vertical coordinate. If dh and dv are positive, the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The rectangle retains its shape and size; it's merely moved

on the coordinate plane. This does not affect the screen unless you subsequently call a routine to draw within the rectangle.

Procedure `InsetRect` (var r: Rect; dh,dv: integer);

`InsetRect` shrinks or expands the rectangle. The left and right sides are moved in by the amount specified by `dh`; the top and bottom are moved toward the center by the amount specified by `dv`. If `dh` or `dv` is negative, the appropriate pair of sides is moved outward instead of inward. The effect is to alter the size by $2*dh$ horizontally and $2*dv$ vertically, with the rectangle remaining centered in the same place on the coordinate plane.

If the resulting width or height becomes less than 1, the rectangle is set to the empty rectangle (0,0,0,0).

Function `SectRect` (srcRectA,srcRectB: Rect; var dstRect: Rect) : boolean;

`SectRect` calculates the rectangle that is the intersection of the two input rectangles, and returns TRUE if they indeed intersect or FALSE if they do not. Rectangles that "touch" at a line or a point are not considered intersecting, because their intersection rectangle (really, in this case, an intersection line or point) does not enclose any bits on the bitMap.

If the rectangles do not intersect, the destination rectangle is set to (0,0,0,0). `SectRect` works correctly even if one of the source rectangles is also the destination.

Procedure `UnionRect` (srcRectA,srcRectB: Rect; var dstRect: Rect);

`UnionRect` calculates the smallest rectangle which encloses both input rectangles. It works correctly even if one of the source rectangles is also the destination.

Function `PtInRect` (pt: Point; r: Rect) : boolean;

`PtInRect` determines whether the pixel below and to the right of the given coordinate point is enclosed in the specified rectangle, and returns TRUE if so or FALSE if not.

Procedure `Pt2Rect` (ptA,ptB: Point; var dstRect: Rect);

`Pt2Rect` returns the smallest rectangle which encloses the two input points.

Procedure PtToAngle (r: Rect; pt: Point; var angle: integer);

PtToAngle calculates an integer angle between a line from the center of the rectangle to the given point and a line from the center of the rectangle pointing straight up (12 o'clock high). The angle is in degrees from 0 to 359, measured clockwise from 12 o'clock, with 90 degrees at 3 o'clock, 180 at 6 o'clock, and 270 at 9 o'clock. Other angles are measured relative to the rectangle: If the line to the given point goes through the top right corner of the rectangle, the angle returned is 45 degrees, even if the rectangle is not square; if it goes through the bottom right corner, the angle is 135 degrees, and so on (see Figure E-18).

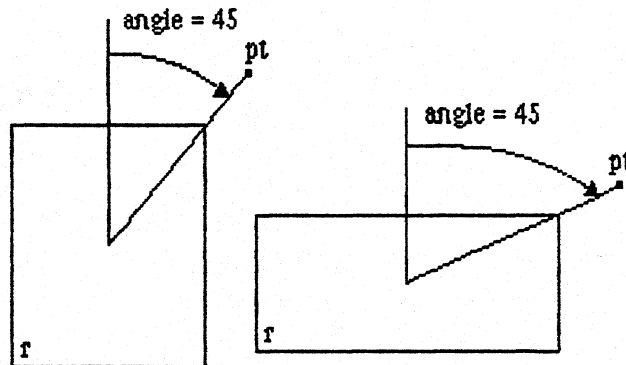


Figure E-18
PtToAngle

The angle returned might be used as input to one of the procedures that manipulate arcs and wedges, as described in Section E.9.10, Graphic Operations on Arcs and Wedges.

Function EqualRect (rectA, rectB: Rect): boolean;

EqualRect compares the two rectangles and returns TRUE if they are equal or FALSE if not. The two rectangles must have identical boundary coordinates to be considered equal.

Function EmptyRect (r: Rect): boolean;

EmptyRect returns TRUE if the given rectangle is an empty rectangle or FALSE if not. A rectangle is considered empty if the bottom coordinate is equal to or less than the top or the right coordinate is equal to or less than the left.

E.9.7 Graphic Operations on Rectangles

These procedures perform graphic operations on rectangles. See also ScrollRect in Section E.9.13, Bit Transfer Operations.

Procedure FrameRect (r: Rect);

FrameRect draws an outline just inside the specified rectangle, using the current grafPort's pen pattern, mode, and size. The outline is as wide as the pen width and as tall as the pen height. It is drawn with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

If a region is open and being formed, the outside outline of the new rectangle is mathematically added to the region's boundary.

Procedure PaintRect (r: Rect);

PaintRect paints the specified rectangle with the current grafPort's pen pattern and mode. The rectangle on the bitMap is filled with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

Procedure EraseRect (r: Rect);

EraseRect paints the specified rectangle with the current grafPort's background pattern bkPat (in patCopy mode). The grafPort's pnPat and pnMode are ignored; the pen location is not changed.

Procedure InvertRect (r: Rect);

InvertRect inverts the pixels enclosed by the specified rectangle: every white pixel becomes black and every black pixel becomes white. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

Procedure FillRect (r: Rect; pat: Pattern);

FillRect fills the specified rectangle with the given pattern (in patCopy mode). The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

E.9.8 Graphic Operations on Ovals

Ovals are drawn inside rectangles that you specify. If the rectangle you specify is square, QuickDraw draws a circle.

Procedure FrameOval (r: Rect);

FrameOval draws an outline just inside the oval that fits inside the specified rectangle, using the current grafPort's pen pattern, mode, and size. The outline is as wide as the pen width and as tall as the pen height. It is drawn with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

If a region is open and being formed, the outside outline of the new oval is mathematically added to the region's boundary.

Procedure PaintOval (r: Rect);

PaintOval paints an oval just inside the specified rectangle with the current grafPort's pen pattern and mode. The oval on the bitMap is filled with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

Procedure EraseOval (r: Rect);

EraseOval paints an oval just inside the specified rectangle with the current grafPort's background pattern bkPat (in patCopy mode). The grafPort's pnPat and pnMode are ignored; the pen location is not changed.

Procedure InvertOval (r: Rect);

InvertOval inverts the pixels enclosed by an oval just inside the specified rectangle: every white pixel becomes black and every black pixel becomes white. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

Procedure FillOval (r: Rect; pat: Pattern);

FillOval fills an oval just inside the specified rectangle with the given pattern (in patCopy mode). The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

E.9.9 Graphic Operations on Rounded-Corner Rectangles**Procedure FrameRoundRect (r: Rect; ovalWidth, ovalHeight: integer);**

FrameRoundRect draws a hollow outline just inside the specified rounded-corner rectangle, using the current grafPort's pen pattern, mode, and size. OvalWidth and ovalHeight specify the diameters of curvature for the corners (see Figure E-19). The outline is as wide as the pen width and as tall as the pen height. It is drawn with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

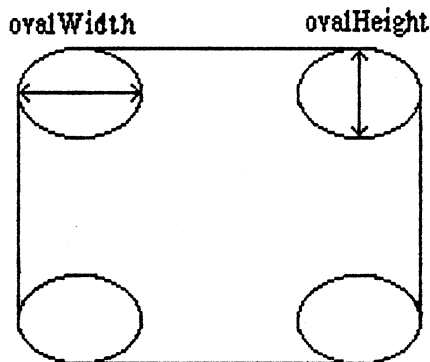


Figure E-19
Rounded-Corner Rectangle

If a region is open and being formed, the outside outline of the new rounded-corner rectangle is mathematically added to the region's boundary.

Procedure `PaintRoundRect` (`r: Rect; ovalWidth, ovalHeight: integer`);

`PaintRoundRect` paints the specified rounded-corner rectangle with the current `grafPort`'s pen pattern and mode. `OvalWidth` and `ovalHeight` specify the diameters of curvature for the corners. The rounded-corner rectangle on the `bitMap` is filled with the `pnPat`, according to the pattern transfer mode specified by `pnMode`. The pen location is not changed by this procedure.

Procedure `EraseRoundRect` (`r: Rect; ovalWidth, ovalHeight: integer`);

`EraseRoundRect` paints the specified rounded-corner rectangle with the current `grafPort`'s background pattern `bkPat` (in `patCopy` mode). `OvalWidth` and `ovalHeight` specify the diameters of curvature for the corners. The `grafPort`'s `pnPat` and `pnMode` are ignored; the pen location is not changed.

Procedure `InvertRoundRect` (`r: Rect; ovalWidth, ovalHeight: integer`);

`InvertRoundRect` inverts the pixels enclosed by the specified rounded-corner rectangle: every white pixel becomes black and every black pixel becomes white. `OvalWidth` and `ovalHeight` specify the diameters of curvature for the corners. The `grafPort`'s `pnPat`, `pnMode`, and `bkPat` are all ignored; the pen location is not changed.

Procedure FillRoundRect (r: Rect; ovalWidth, ovalHeight: integer; pat: Pattern);

FillRoundRect fills the specified rounded-corner rectangle with the given pattern (in patCopy mode). OvalWidth and ovalHeight specify the diameters of curvature for the corners. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

E.9.10 Graphic Operations on Arcs and Wedges

These procedures perform graphic operations on arcs and wedge-shaped sections of ovals. See also PtToAngle in Section E.9.6, Calculations with Rectangles.

Procedure FrameArc (r: Rect; startAngle, arcAngle: integer);

FrameArc draws an arc of the oval that fits inside the specified rectangle, using the current grafPort's pen pattern, mode, and size. StartAngle indicates where the arc begins and is treated mod 360. ArcAngle defines the extent of the arc. The angles are given in positive or negative degrees; a positive angle goes clockwise, while a negative angle goes counterclockwise. Zero degrees is at 12 o'clock high, 90 (or -270) is at 3 o'clock, 180 (or -180) is at 6 o'clock, and 270 (or -90) is at 9 o'clock. Other angles are measured relative to the enclosing rectangle: a line from the center of the rectangle through its top right corner is at 45 degrees, even if the rectangle is not square; a line through the bottom right corner is at 135 degrees, and so on (see Figure E-20).

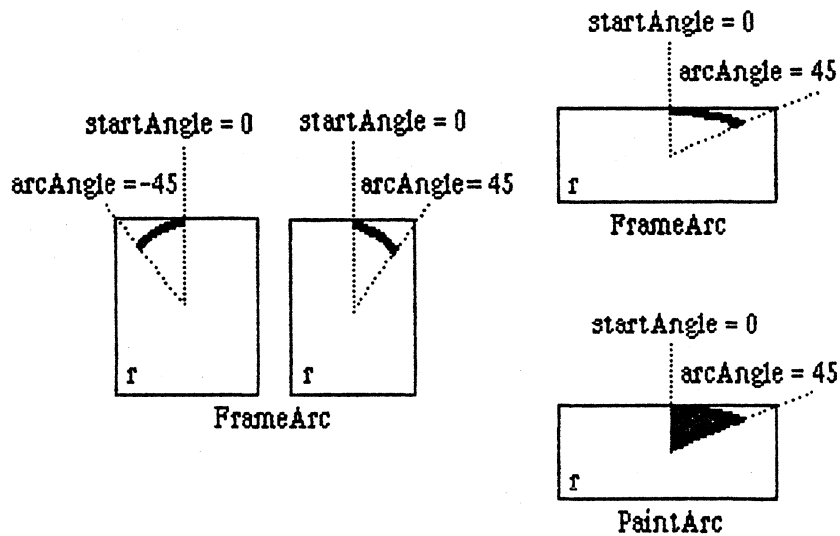


Figure E-20
Operations on Arcs and Wedges

The arc is as wide as the pen width and as tall as the pen height. It is drawn with the `pnPat`, according to the pattern transfer mode specified by `pnMode`. The pen location is not changed by this procedure.

WARNING

`FrameArc` differs from other QuickDraw procedures that frame shapes in that the arc is not mathematically added to the boundary of a region that is open and being formed.

Procedure `PaintArc` (`r: Rect; startAngle, arcAngle: integer`);

`PaintArc` paints a wedge of the oval just inside the specified rectangle with the current `grafPort`'s pen pattern and mode. `startAngle` and `arcAngle` define the arc of the wedge as in `FrameArc`. The wedge on the `bitMap` is filled with the `pnPat`, according to the pattern transfer mode specified by `pnMode`. The pen location is not changed by this procedure.

Procedure EraseArc (r: Rect; startAngle, arcAngle: integer);

EraseArc paints a wedge of the oval just inside the specified rectangle with the current grafPort's background pattern bkPat (in patCopy mode). StartAngle and arcAngle define the arc of the wedge as in FrameArc. The grafPort's pnPat and pnMode are ignored; the pen location is not changed.

Procedure InvertArc (r: Rect; startAngle, arcAngle: integer);

InvertArc inverts the pixels enclosed by a wedge of the oval just inside the specified rectangle: every white pixel becomes black and every black pixel becomes white. StartAngle and arcAngle define the arc of the wedge as in FrameArc. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

Procedure FillArc (r: Rect; startAngle, arcAngle: integer; pat: Pattern);

FillArc fills a wedge of the oval just inside the specified rectangle with the given pattern (in patCopy mode). StartAngle and arcAngle define the arc of the wedge as in FrameArc. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

E.9.11 Calculations with Regions

NOTE

Remember that if the parameters to one of the calculation routines were defined in different grafPorts, you must first adjust them to be in the same coordinate system. If you do not adjust them, the result returned by the routine may be different from what you see on the screen. To adjust to a common coordinate system, see LocaltoGlobal and GlobalToLocal in Section E.9.17, Calculations with Points.

Function NewRgn : RgnHandle;

NewRgn allocates space for a new, dynamic, variable-size region, initializes it to the empty region (0,0,0,0), and returns a handle to the new region. Only this function creates new regions; all other procedures just alter the size and shape of regions you create. OpenPort calls NewRgn to allocate space for the port's visRgn and clipRgn.

WARNING

Except when using `visRgn` or `clipRgn`, you **MUST** call `NewRgn` before specifying a region's handle in any drawing or calculation procedure.

WARNING

Never refer to a region without using its handle.

Procedure `DisposeRgn` (`rgn: RgnHandle`);

`DisposeRgn` deallocates space for the region whose handle is supplied, and returns the memory used by the region to the free memory pool. Use this only after you are completely through with a temporary region.

WARNING

Never use a region once you have deallocated it, or you will risk being hung by dangling pointers!

Procedure `CopyRgn` (`srcRgn,dstRgn: RgnHandle`);

`CopyRgn` copies the mathematical structure of `srcRgn` into `dstRgn`; that is, it makes a duplicate copy of `srcRgn`. Once this is done, `srcRgn` may be altered (or even disposed of) without affecting `dstRgn`. `CopyRgn` does not create the destination region: you must use `NewRgn` to create the `dstRgn` before you call `CopyRgn`.

Procedure `SetEmptyRgn` (`rgn: RgnHandle`);

`SetEmptyRgn` destroys the previous structure of the given region, then sets the new structure to the empty region (0,0,0,0).

Procedure `SetRectRgn` (`rgn: RgnHandle`; `left,top,right,bottom: integer`);

`SetRectRgn` destroys the previous structure of the given region, then sets the new structure to the rectangle specified by `left`, `top`, `right`, and `bottom`.

If the specified rectangle is empty (i.e., `left>=right` or `top>=bottom`), the region is set to the empty region (0,0,0,0).

Procedure RectRgn (rgn: RgnHandle; r: Rect);

RectRgn destroys the previous structure of the given region, then sets the new structure to the rectangle specified by r. This is operationally synonymous with SetRectRgn, except the input rectangle is defined by a rectangle rather than by four boundary coordinates.

Procedure OpenRgn;

OpenRgn tells QuickDraw to allocate temporary space and start saving lines and framed shapes for later processing as a region definition. While a region is open, all calls to Line, LineTo, and the procedures that draw framed shapes (except arcs) affect the outline of the region. Only the line endpoints and shape boundaries affect the region definition; the pen mode, pattern, and size do not affect it. In fact, OpenRgn calls HidePen, so no drawing occurs on the screen while the region is open (unless you called ShowPen just after OpenRgn, or you called ShowPen previously without balancing it by a call to HidePen). Since the pen hangs below and to the right of the pen location, drawing lines with even the smallest pen will change bits that lie outside the region you define.

The outline of a region is mathematically defined and infinitely thin, and separates the bitMap into two groups of bits: those within the region and those outside it. A region should consist of one or more closed loops. Each framed shape itself constitutes a loop. Any lines drawn with Line or LineTo should connect with each other or with a framed shape. Even though the on-screen presentation of a region is clipped, the definition of a region is not; you can define a region anywhere on the coordinate plane with complete disregard for the location of various grafPort entities on that plane.

When a region is open, the current grafPort's rgnSave field contains a handle to information related to the region definition. If you want to temporarily disable the collection of lines and shapes, you can save the current value of this field, set the field to NIL, and later restore the saved value to resume the region definition.

WARNING

Do not call OpenRgn while another region is already open. All open regions but the most recent will behave strangely.

Procedure CloseRgn (dstRgn: RgnHandle);

CloseRgn stops the collection of lines and framed shapes, organizes them into a region definition, and saves the resulting region into the region indicated by

dstRgn. You should perform one and only one CloseRgn for every OpenRgn. CloseRgn calls ShowPen, balancing the HidePen call made by OpenRgn.

Here's an example of how to create and open a region, define a barbell shape, close the region, and draw it:

```

barbell := NewRgn;           {make a new region}
OpenRgn;                    {begin collecting stuff}
  SetRect(tempRect,20,20,30,50); {form the left weight}
  FrameOval(tempRect);
  SetRect(tempRect,30,30,80,40); {form the bar}
  FrameRect(tempRect);
  SetRect(tempRect,80,20,90,50); {form the right weight}
  FrameOval(tempRect);
CloseRgn(barbell);         {we're done; save in barbell}
FillRgn(barbell,black);   {draw it on the screen}
DisposeRgn(barbell);      {we don't need you anymore...}

```

Procedure OffsetRgn (rgn: RgnHandle; dh,dv: integer);

OffsetRgn moves the region on the coordinate plane, a distance of dh horizontally and dv vertically. This does not affect the screen unless you subsequently call a routine to draw the region. If dh and dv are positive, the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The region retains its size and shape.

NOTE

OffsetRgn is an especially efficient operation, because most of the data defining a region is stored relative to rgnBBox and so isn't actually changed by OffsetRgn.

Procedure InsetRgn (rgn: RgnHandle; dh,dv: integer);

InsetRgn shrinks or expands the region. All points on the region boundary are moved inwards a distance of dv vertically and dh horizontally; if dh or dv is negative, the points are moved outwards in that direction. InsetRgn leaves the region "centered" at the same position, but moves the outline in (for positive values of dh and dv) or out (for negative values of dh and dv). InsetRgn of a rectangular region works just like InsetRect.

Procedure SectRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);

SectRgn calculates the intersection of two regions and places the intersection in a third region. This does not create the destination region: you must use

NewRgn to create the dstRgn before you call SectRgn. The dstRgn can be one of the source regions, if desired.

If the regions do not intersect, or one of the regions is empty, the destination is set to the empty region (0,0,0,0).

Procedure UnionRgn(srcRgnA,srcRgnB,dstRgn: RgnHandle);

UnionRgn calculates the union of two regions and places the union in a third region. This does not create the destination region: you must use NewRgn to create the dstRgn before you call UnionRgn. The dstRgn can be one of the source regions, if desired.

If both regions are empty, the destination is set to the empty region (0,0,0,0).

Procedure DiffRgn(srcRgnA,srcRgnB,dstRgn: RgnHandle);

DiffRgn subtracts srcRgnB from srcRgnA and places the difference in a third region. This does not create the destination region: you must use NewRgn to create the dstRgn before you call DiffRgn. The dstRgn can be one of the source regions, if desired.

If the first source region is empty, the destination is set to the empty region (0,0,0,0).

Procedure XorRgn(srcRgnA,srcRgnB,dstRgn: RgnHandle);

XorRgn calculates the difference between the union and the intersection of two regions and places the result in a third region. This does not create the destination region: you must use NewRgn to create the dstRgn before you call XorRgn. The dstRgn can be one of the source regions, if desired.

If the regions are coincident, the destination is set to the empty region (0,0,0,0).

Function PtInRgn(pt: Point; rgn: RgnHandle): boolean;

PtInRgn checks whether the pixel below and to the right of the given coordinate point is within the specified region, and returns TRUE if so or FALSE if not.

Function RectInRgn(r: Rect; rgn: RgnHandle): boolean;

RectInRgn checks whether the given rectangle intersects the specified region, and returns TRUE if the intersection encloses at least one bit or FALSE if not.

Function `EqualRgn (rgnA,rgnB: rgnHandle)`: boolean;

`EqualRgn` compares the two regions and returns TRUE if they are equal or FALSE if not. The two regions must have identical sizes, shapes, and locations to be considered equal. Any two empty regions are always equal.

Function `EmptyRgn (rgn: RgnHandle)`: boolean;

`EmptyRgn` returns TRUE if the region is an empty region or FALSE if not. Some of the circumstances in which an empty region can be created are: a `NewRgn` call; a `CopyRgn` of an empty region; a `SetRectRgn` or `RectRgn` with an empty rectangle as an argument; `CloseRgn` without a previous `OpenRgn` or with no drawing after an `OpenRgn`; `OffsetRgn` of an empty region; `InsetRgn` with an empty region or too large an inset; `SectRgn` of nonintersecting regions; `UnionRgn` of two empty regions; and `DiffRgn` or `XorRgn` of two identical or nonintersecting regions.

E.9.12 Graphic Operations on Regions

These routines all depend on the coordinate system of the current `grafPort`. If a region is drawn in a different `grafPort` than the one in which it was defined, it may not appear in the proper position inside the port.

Procedure `FrameRgn (rgn: RgnHandle)`;

`FrameRgn` draws a hollow outline just inside the specified region, using the current `grafPort`'s pen pattern, mode, and size. The outline is as wide as the pen width and as tall as the pen height; under no circumstances will the frame go outside the region boundary. The pen location is not changed by this procedure.

If a region is open and being formed, the outside outline of the region being framed is mathematically added to that region's boundary.

Procedure `PaintRgn (rgn: RgnHandle)`;

`PaintRgn` paints the specified region with the current `grafPort`'s pen pattern and pen mode. The region on the `bitMap` is filled with the `pnPat`, according to the pattern transfer mode specified by `pnMode`. The pen location is not changed by this procedure.

Procedure `EraseRgn (rgn: RgnHandle)`;

`EraseRgn` paints the specified region with the current `grafPort`'s background pattern `bkPat` (in `patCopy` mode). The `grafPort`'s `pnPat` and `pnMode` are ignored; the pen location is not changed.

Procedure `InvertRgn` (rgn: RgnHandle);

`InvertRgn` inverts the pixels enclosed by the specified region: every white pixel becomes black and every black pixel becomes white. The grafPort's `pnPat`, `pnMode`, and `bkPat` are all ignored; the pen location is not changed.

Procedure `FillRgn` (rgn: RgnHandle; pat: Pattern);

`FillRgn` fills the specified region with the given pattern (in `patCopy` mode). The grafPort's `pnPat`, `pnMode`, and `bkPat` are all ignored; the pen location is not changed.

E.9.13 Bit Transfer Operations

Procedure `ScrollRect` (r: Rect; dh,dv: integer; updateRgn: RgnHandle);

`ScrollRect` shifts ("scrolls") those bits inside the intersection of the specified rectangle, `visRgn`, `clipRgn`, `portRect`, and `portBits.bounds`. The bits are shifted a distance of `dh` horizontally and `dv` vertically. The positive directions are to the right and down. No other bits are affected. Bits that are shifted out of the scroll area are lost; they are neither placed outside the area nor saved. The grafPort's background pattern `bkPat` fills the space created by the scroll. In addition, `updateRgn` is changed to the area filled with `bkPat` (see Figure E-21).

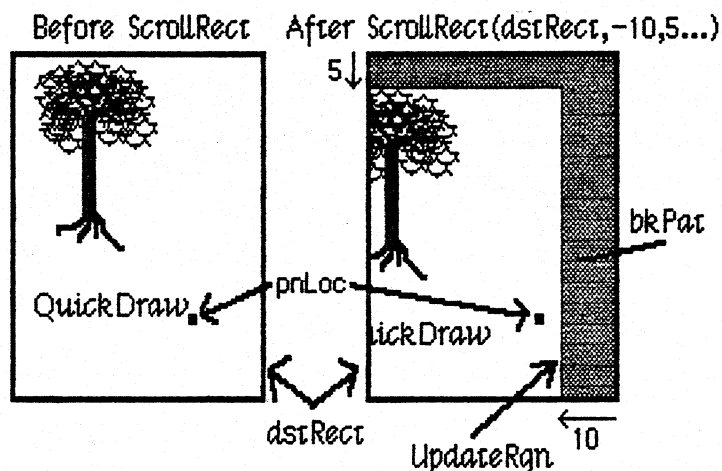


Figure E-21
Scrolling

Figure E-21 shows that the pen location after a `ScrollRect` is in a different position relative to what was scrolled in the rectangle. The entire scrolled item has been moved to different coordinates. To restore it to its coordinates before the `ScrollRect`, you can use the `SetOrigin` procedure. For example, suppose the `dstRect` here is the `portRect` of the grafPort and its top left corner is at (95,120). `SetOrigin(105,115)` will offset the coordinate system to compensate for the

scroll. Since the clipRgn and pen location are not offset, they move down and to the left.

Procedure CopyBits (srcBits,dstBits: BitMap; srcRect,dstRect: Rect; mode: integer; maskRgn: RgnHandle);

CopyBits transfers a bit image between any two bitMaps and clips the result to the area specified by the maskRgn parameter. The transfer may be performed in any of the eight source transfer modes. The result is always clipped to the maskRgn and the boundary rectangle of the destination bitMap; if the destination bitMap is the current grafPort's portBits, it is also clipped to the intersection of the grafPort's clipRgn and visRgn. If you do not want to clip to a maskRgn, just pass NIL for the maskRgn parameter.

The dstRect and maskRgn coordinates are in terms of the dstBits.bounds coordinate system, and the srcRect coordinates are in terms of the srcBits.bounds coordinates.

The bits enclosed by the source rectangle are transferred into the destination rectangle according to the rules of the chosen mode. The source transfer modes are as follows:

srcCopy	srcXor	notSrcCopy	notSrcXor
srcOr	srcBic	notSrcOr	notSrcBic

The source rectangle is completely aligned with the destination rectangle; if the rectangles are of different sizes, the bit image is expanded or shrunk as necessary to fit the destination rectangle. For example, if the bit image is a circle in a square source rectangle, and the destination rectangle is not square, the bit image appears as an oval in the destination (see Figure E-22).

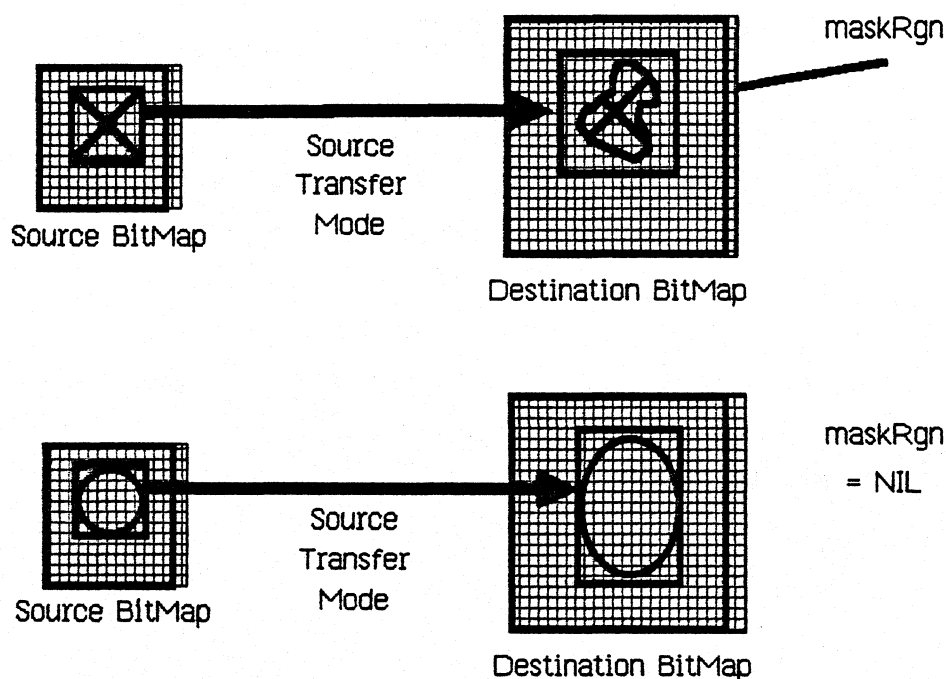


Figure E-22
Operation of CopyBits

E.9.14 Pictures

Function `OpenPicture` (`picFrame: Rect`): `PicHandle`;

`OpenPicture` returns a handle to a new picture which has the given rectangle as its picture frame, and tells QuickDraw to start saving as the picture definition all calls to drawing routines and all picture comments (if any).

`OpenPicture` calls `HidePen`, so no drawing occurs on the screen while the picture is open (unless you call `ShowPen` just after `OpenPicture`, or you called `ShowPen` previously without balancing it by a call to `HidePen`).

When a picture is open, the current `grafPort`'s `picSave` field contains a handle to information related to the picture definition. If you want to temporarily disable the collection of routine calls and picture comments, you can save the current value of this field, set the field to `NIL`, and later restore the saved value to resume the picture definition.

WARNING

Do not call `OpenPicture` while another picture is already open.

Procedure `ClosePicture`;

`ClosePicture` tells QuickDraw to stop saving routine calls and picture comments as the definition of the currently open picture. You should perform one and only one `ClosePicture` for every `OpenPicture`. `ClosePicture` calls `ShowPen`, balancing the `HidePen` call made by `OpenPicture`.

Procedure `PicComment` (`kind`, `dataSize`: integer; `dataHandle`: QDHandle);

`PicComment` inserts the specified comment into the definition of the currently open picture. `Kind` identifies the type of comment. `DataHandle` is a handle to additional data if desired, and `dataSize` is the size of that data in bytes. If there is no additional data for the comment, `dataHandle` should be `NIL` and `dataSize` should be 0. The application that processes the comment must include a procedure to do the processing and store a pointer to the procedure in the data structure pointed to by the `grafProcs` field of the `grafPort` (see Section E.10, Customizing QuickDraw Operations).

Procedure `DrawPicture` (`myPicture`: PicHandle; `dstRect`: Rect);

`DrawPicture` draws the given picture to scale in `dstRect`, expanding or shrinking it as necessary to align the borders of the picture frame with `dstRect`. `DrawPicture` passes any picture comments to the procedure accessed indirectly through the `grafProcs` field of the `grafPort` (see `PicComment` above).

Procedure `KillPicture` (`myPicture`: PicHandle);

`KillPicture` deallocates space for the picture whose handle is supplied, and returns the memory used by the picture to the free memory pool. Use this only when you are completely through with a picture.

E.9.15 Calculations with Polygons**Function `OpenPoly` : PolyHandle;**

`OpenPoly` returns a handle to a new polygon and tells QuickDraw to start saving the polygon definition as specified by calls to line-drawing routines. While a polygon is open, all calls to `Line` and `LineTo` affect the outline of the polygon. Only the line endpoints affect the polygon definition; the pen mode, pattern, and size do not affect it. In fact, `OpenPoly` calls `HidePen`, so no drawing occurs on the

screen while the polygon is open (unless you call ShowPen just after OpenPoly, or you called ShowPen previously without balancing it by a call to HidePen).

A polygon should consist of a sequence of connected lines. Even though the on-screen presentation of a polygon is clipped, the definition of a polygon is not; you can define a polygon anywhere on the coordinate plane with complete disregard for the location of various grafPort entities on that plane.

When a polygon is open, the current grafPort's polySave field contains a handle to information related to the polygon definition. If you want to temporarily disable the polygon definition, you can save the current value of this field, set the field to NIL, and later restore the saved value to resume the polygon definition.

WARNING

Do not call OpenPoly while another polygon is already open.

Procedure ClosePoly;

ClosePoly tells QuickDraw to stop saving the definition of the currently open polygon and computes the polyBBox rectangle. You should perform one and only one ClosePoly for every OpenPoly. ClosePoly calls ShowPen, balancing the HidePen call made by OpenPoly.

Here's an example of how to open a polygon, define it as a triangle, close it, and draw it:

triPoly := OpenPoly;	{save handle and begin collecting stuff}
MoveTo(300,100);	{ move to first point and }
LineTo(400,200);	{ form }
LineTo(200,200);	{ the }
LineTo(300,100);	{ triangle }
ClosePoly;	{ stop collecting stuff }
FillPoly(triPoly,gray);	{ draw it on the screen }
KillPoly(triPoly);	{ we're all done }

Procedure KillPoly (poly: PolyHandle);

KillPoly deallocates space for the polygon whose handle is supplied, and returns the memory used by the polygon to the free memory pool. Use this only after you are completely through with a polygon.

Procedure OffsetPoly (poly: PolyHandle; dh, dv: integer);

OffsetPoly moves the polygon on the coordinate plane, a distance of dh horizontally and dv vertically. This does not affect the screen unless you subsequently call a routine to draw the polygon. If dh and dv are positive, the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The polygon retains its shape and size.

NOTE

OffsetPoly is an especially efficient operation, because the data defining a polygon is stored relative to polyStart and so isn't actually changed by OffsetPoly.

E.9.16 Graphic Operations on Polygons

Procedure FramePoly (poly: PolyHandle);

FramePoly plays back the line-drawing routine calls that define the given polygon, using the current grafPort's pen pattern, mode, and size. The pen will hang below and to the right of each point on the boundary of the polygon; thus, the polygon drawn will extend beyond the right and bottom edges of poly^.polyBBox by the pen width and pen height, respectively. All other graphic operations occur strictly within the boundary of the polygon, as for other shapes. You can see this difference in Figure E-23, where each of the polygons is shown with its polyBBox.

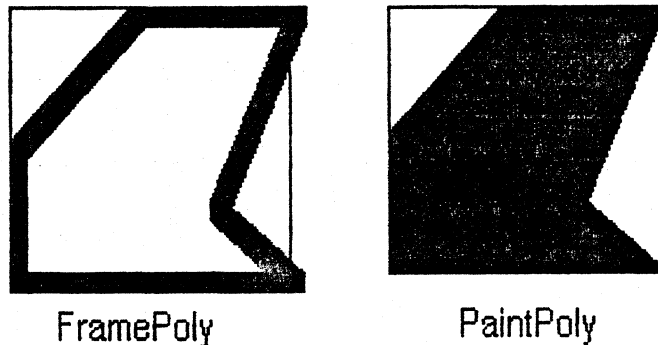


Figure E-23
Drawing Polygons

If a polygon is open and being formed, FramePoly affects the outline of the polygon just as if the line-drawing routines themselves had been called. If a

region is open and being formed, the outside outline of the polygon being framed is mathematically added to the region's boundary.

Procedure PaintPoly (poly: PolyHandle);

PaintPoly paints the specified polygon with the current grafPort's pen pattern and pen mode. The polygon on the bitMap is filled with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

Procedure ErasePoly (poly: PolyHandle);

ErasePoly paints the specified polygon with the current grafPort's background pattern bkPat (in patCopy mode). The pnPat and pnMode are ignored; the pen location is not changed.

Procedure InvertPoly (poly: PolyHandle);

InvertPoly inverts the pixels enclosed by the specified polygon: every white pixel becomes black and every black pixel becomes white. The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

Procedure FillPoly (poly: PolyHandle; pat: Pattern);

FillPoly fills the specified polygon with the given pattern (in patCopy mode). The grafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

E.9.17 Calculations with Points

Procedure AddPt (srcPt: Point; var dstPt: Point);

AddPt adds the coordinates of srcPt to the coordinates of dstPt, and returns the result in dstPt.

Procedure SubPt (srcPt: Point; var dstPt: Point);

SubPt subtracts the coordinates of srcPt from the coordinates of dstPt, and returns the result in dstPt.

Procedure SetPt (var pt: Point; h,v: integer);

SetPt assigns two integer coordinates to a variable of type Point.

Function EqualPt (ptA,ptB: Point): boolean;

EqualPt compares the two points and returns TRUE if they are equal or FALSE if not.

Procedure LocalToGlobal (var pt: Point);

LocalToGlobal converts the given point from the current grafPort's local coordinate system into a global coordinate system with the origin (0,0) at the top left corner of the port's bit image (such as the screen). This global point can then be compared to other global points, or be changed into the local coordinates of another grafPort.

Since a rectangle is defined by two points, you can convert a rectangle into global coordinates by performing two LocalToGlobal calls. You can also convert a rectangle, region, or polygon into global coordinates by calling OffsetRect, OffsetRgn, or OffsetPoly. For examples, see GlobalToLocal below.

Procedure GlobalToLocal (var pt: Point);

GlobalToLocal takes a point expressed in global coordinates (with the top left corner of the bitMap as coordinate (0,0)) and converts it into the local coordinates of the current grafPort. The global point can be obtained with the LocalToGlobal call (see above). For example, suppose a game draws a "ball" within a rectangle named ballRect, defined in the grafPort named gamePort (as illustrated below in Figure E-24). If you want to draw that ball in the grafPort named selectPort, you can calculate the ball's selectPort coordinates like this:

```

SetPort(gamePort);           {start in origin port}
selectBall := ballRect;     {make a copy to be moved}
LocalToGlobal(selectBall.topLeft); {put both corners into }
LocalToGlobal(selectBall.botRight); { global coordinates  }

SetPort(selectPort);        {switch to destination port}
GlobalToLocal(selectBall.topLeft); {put both corners into }
GlobalToLocal(selectBall.botRight); { these local coordinates }
FillOval(selectBall,ballColor); {now you have the ball!}

```

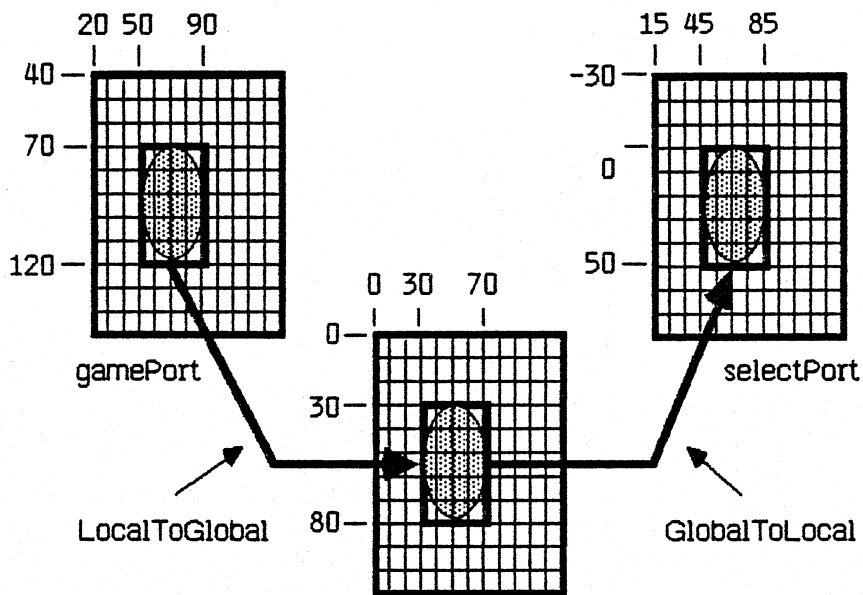


Figure E-24
Converting between Coordinate Systems

You can see from Figure E-24 that `LocalToGlobal` and `GlobalToLocal` simply offset the coordinates of the rectangle by the coordinates of the top left corner of the local `grafPort`'s boundary rectangle. You could also do this with `OffsetRect`. In fact, the way to convert regions and polygons from one coordinate system to another is with `OffsetRgn` or `OffsetPoly` rather than `LocalToGlobal` and `GlobalToLocal`. For example, if `myRgn` were a region enclosed by a rectangle having the same coordinates as `ballRect` in `gamePort`, you could convert the region to global coordinates with

```
OffsetRgn(myRgn, -20, -40);
```

and then convert it to the coordinates of the `selectPort` `grafPort` with

```
OffsetRgn(myRgn, 15, -30);
```

E.9.18 Miscellaneous Utilities

Function `Random`: integer;

This function returns an integer, uniformly distributed pseudo-random, in the range from -32768 through 32767. The value returned depends on the global

variable `randSeed`, which `InitGraf` initializes to 1; you can start the sequence over again from where it began by resetting `randSeed` to 1.

Function `GetPixel(h,v: integer): boolean;`

`GetPixel` looks at the pixel associated with the given coordinate point and returns `TRUE` if it is black or `FALSE` if it is white. The selected pixel is immediately below and to the right of the point whose coordinates are given in `h` and `v`, in the local coordinates of the current `grafPort`. There is no guarantee that the specified pixel actually belongs to the port, however; it may have been drawn by a port overlapping the current one. To see if the point indeed belongs to the current port, perform a `PtInRgn(pt,thePort^.visRgn)`.

Procedure `StuffHex(thingPtr: QDPtr; s: Str255);`

`StuffHex` pokes bits (expressed as a string of hexadecimal digits) into any data structure. This is a good way to create cursors, patterns, or bit images to be "stamped" onto the screen with `CopyBits`. For example,

```
StuffHex(@stripes,'0102040810204080')
```

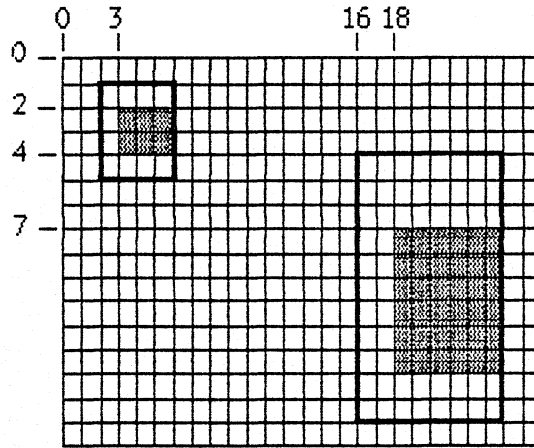
places a striped pattern into the pattern variable `stripes`.

WARNING

There is no range checking on the size of the destination variable. It's easy to overrun the variable and destroy something if you don't know what you're doing.

Procedure `ScalePt(var pt: Point; srcRect,dstRect: Rect);`

A width and height are passed in `pt`; the horizontal component of `pt` is the width, and the vertical component of `pt` is the height. `ScalePt` scales these measurements as follows and returns the result in `pt`: it multiplies the given width by the ratio of `dstRect`'s width to `srcRect`'s width, and multiplies the given height by the ratio of `dstRect`'s height to `srcRect`'s height. In Figure E-25, where `dstRect`'s width is twice `srcRect`'s width and its height is three times `srcRect`'s height, the pen width is scaled from 3 to 6 and the pen height is scaled from 2 to 6.



ScalePt scales pen size (3,3) to (6,6)

MapPt maps point (3,2) to (18,7)

Figure E-25
ScalePt and MapPt

Procedure MapPt (var pt: Point; srcRect, dstRect: Rect);

Given a point within srcRect, MapPt maps it to a similarly located point within dstRect (that is, to where it would fall if it were part of a drawing being expanded or shrunk to fit dstRect). The result is returned in pt. A corner point of srcRect would be mapped to the corresponding corner point of dstRect, and the center of srcRect to the center of dstRect. In Figure E-25 above, the point (3,2) in srcRect is mapped to (18,7) in dstRect. FromRect and dstRect may overlap, and pt need not actually be within srcRect.

WARNING

Remember, if you are going to draw inside the rectangle in dstRect, you will probably also want to scale the pen size accordingly with ScalePt.

Procedure MapRect (var r: Rect; srcRect,dstRect: Rect);

Given a rectangle within srcRect, MapRect maps it to a similarly located rectangle within dstRect by calling MapPt to map the top left and bottom right corners of the rectangle. The result is returned in r.

Procedure MapRgn (rgn: RgnHandle; srcRect,dstRect: Rect);

Given a region within srcRect, MapRgn maps it to a similarly located region within dstRect by calling MapPt to map all the points in the region.

Procedure MapPoly (poly: PolyHandle; srcRect,dstRect: Rect);

Given a polygon within srcRect, MapPoly maps it to a similarly located polygon within dstRect by calling MapPt to map all the points that define the polygon.

E.10 Customizing QuickDraw Operations

For each shape that QuickDraw knows how to draw, there are procedures that perform these basic graphic operations on the shape: frame, paint, erase, invert, and fill. Those procedures in turn call a low-level drawing routine for the shape. For example, the FrameOval, PaintOval, EraseOval, InvertOval, and FillOval procedures all call a low-level routine that draws the oval. For each type of object QuickDraw can draw, including text and lines, there is a pointer to such a routine. By changing these pointers, you can install your own routines, and either completely override the standard ones or call them after your routines have modified parameters as necessary.

Other low-level routines that you can install in this way are:

- The procedure that does bit transfer and is called by CopyBits.
- The function that measures the width of text and is called by CharWidth, StringWidth, and TextWidth.
- The procedure that processes picture comments and is called by DrawPicture. The standard such procedure ignores picture comments.
- The procedure that saves drawing commands as the definition of a picture, and the one that retrieves them. This enables the application to draw on remote devices, print to the disk, get picture input from the disk, and support large pictures.

The grafProcs field of a grafPort determines which low-level routines are called; if it contains NIL, the standard routines are called, so that all operations in that grafPort are done in the standard ways described in this manual. You can set the grafProcs field to point to a record of pointers to routines. The data type of grafProcs is QDProcsPtr:

```

type QDProcsPtr = ^QDProcs;
    QDProcs = record
        textProc:    QDPtr; {text drawing}
        lineProc:    QDPtr; {line drawing}
        rectProc:    QDPtr; {rectangle drawing}
        rRectProc:   QDPtr; {roundRect drawing}
        ovalProc:    QDPtr; {oval drawing}
        arcProc:     QDPtr; {arc/wedge drawing}
        polyProc:    QDPtr; {polygon drawing}
        rgnProc:     QDPtr; {region drawing}
        bitsProc:    QDPtr; {bit transfer}
        commentProc: QDPtr; {picture comment
                             processing}
        txMeasProc:  QDPtr; {text width measurement}
        getPicProc:  QDPtr; {picture retrieval}
        putPicProc:  QDPtr; {picture saving}
    end;

```

To assist you in setting up a QDProcs record, QuickDraw provides the following procedure:

Procedure SetStdProcs (var procs: QDProcs);

This procedure sets all the fields of the given QDProcs record to point to the standard low-level routines. You can then change the ones you wish to point to your own routines. For example, if your procedure that processes picture comments is named MyComments, you will store @MyComments in the commentProc field of the QDProcs record.

The routines you install must of course have the same calling sequences as the standard routines, which are described below. The standard drawing routines tell which graphic operation to perform from a parameter of type GrafVerb.

type GrafVerb = (frame, paint, erase, invert, fill);

When the grafVerb is fill, the pattern to use when filling is passed in the fillPat field of the grafPort.

Procedure StdText (byteCount: integer; textBuf: QDPtr; numer,denom: integer);

StdText is the standard low-level routine for drawing text. It draws text from the arbitrary structure in memory specified by textBuf, starting from the first byte and continuing for byteCount bytes. Numer and denom specify the scaling, if any: numer.v over denom.v gives the vertical scaling, and numer.h over denom.h gives the horizontal scaling.

Procedure StdLine (newPt: Point);

StdLine is the standard low-level routine for drawing a line. It draws a line from the current pen location to the location specified (in local coordinates) by newPt.

Procedure StdRect (verb: GrafVerb; r: Rect);

StdRect is the standard low-level routine for drawing a rectangle. It draws the given rectangle according to the specified grafVerb.

Procedure StdRRect (verb: GrafVerb; r: Rect; ovalwidth, ovalHeight: integer);

StdRRect is the standard low-level routine for drawing a rounded-corner rectangle. It draws the given rounded-corner rectangle according to the specified grafVerb. OvalWidth and ovalHeight specify the diameters of curvature for the corners.

Procedure StdOval (verb: GrafVerb; r: Rect);

StdOval is the standard low-level routine for drawing an oval. It draws an oval inside the given rectangle according to the specified grafVerb.

Procedure StdArc (verb: GrafVerb; r: Rect; startAngle, arcAngle: integer);

StdArc is the standard low-level routine for drawing an arc or a wedge. It draws an arc or wedge of the oval that fits inside the given rectangle. The grafVerb specifies the graphic operation; if it's the frame operation, an arc is drawn; otherwise, a wedge is drawn.

Procedure StdPoly (verb: GrafVerb; poly: PolyHandle);

StdPoly is the standard low-level routine for drawing a polygon. It draws the given polygon according to the specified grafVerb.

Procedure StdRgn (verb: GrafVerb; rgn: RgnHandle);

StdRgn is the standard low-level routine for drawing a region. It draws the given region according to the specified grafVerb.

Procedure StdBits (var srcBits: BitMap; var srcRect, dstRect: Rect; mode: integer; maskRgn: RgnHandle);

StdBits is the standard low-level routine for doing bit transfer. It transfers a bit image between the given bitMap and thePort ^ .portBits, just as if CopyBits were

called with the same parameters and with a destination bitMap equal to thePort^.portBits.

Procedure StdComment (kind,dataSize: integer; dataHandle: QDHandle);

StdComment is the standard low-level routine for processing a picture comment. Kind identifies the type of comment. DataHandle is a handle to additional data, and dataSize is the size of that data in bytes. If there is no additional data for the command, dataHandle will be NIL and dataSize will be 0. StdComment simply ignores the comment.

Function StdTxMeas (byteCount: integer; textBuf: QDPtr; var numer,denom: Point; var info: FontInfo) : integer;

StdTxMeas is the standard low-level routine for measuring text width. It returns the width of the text stored in the arbitrary structure in memory specified by textBuf, starting with the first byte and continuing for byteCount bytes. Numer and denom specify the scaling as in the StdText procedure; note that StdTxMeas may change them.

Procedure StdGetPic (dataPtr: QDPtr; byteCount: integer);

StdGetPic is the standard low-level routine for retrieving information from the definition of a picture. It retrieves the next byteCount bytes from the definition of the currently open picture and stores them in the data structure pointed to by dataPtr.

Procedure StdPutPic (dataPtr: QDPtr; byteCount: integer);

StdPutPic is the standard low-level routine for saving information as the definition of a picture. It saves as the definition of the currently open picture the drawing commands stored in the data structure pointed to by dataPtr, starting with the first byte and continuing for the next byteCount bytes.

E.11 Using QuickDraw from Assembly Language

All QuickDraw routines can be called from assembly language programs as well as from Pascal. When you write an assembly language program to use these routines, though, you must emulate Pascal's parameter passing and variable transfer protocols.

This section discusses how to use the QuickDraw constants, global variables, data types, procedures, and functions from assembly language.

The primary aid to assembly language programmers is a file named GRAFTYPES.TEXT. If you use .INCLUDE to include this file when you assemble your program, all the QuickDraw constants, offsets to locations of global

variables, and offsets into the fields of structured types will be available in symbolic form.

E.11.1 Constants

QuickDraw constants are stored in the GRAFTYPES.TEXT file, and you can use the constant values symbolically. For example, if you've loaded the effective address of the thePort^.txMode field into address register A2, you can set that field to the srcXor mode with this statement:

```
MOVE.W #SRCXOR(A2)
```

To refer to the number of bytes occupied by the QuickDraw global variables, you can use the constant GRAFSIZE. When you call the InitGraf procedure, you must pass a pointer to an area at least that large.

E.11.2 Data Types

Pascal's strong typing ability lets you write Pascal programs without really considering the size of a variable. But in assembly language, you must keep track of the size of every variable. The sizes of the standard Pascal data types are as follows:

Type	Size
integer	Word (2 bytes)
longint	Long (4 bytes)
boolean	Word (2 bytes)
char	Word (2 bytes)
real	Long (4 bytes)

Integers and longints are in two's complement form; booleans have their boolean value in bit 8 of the word (the low-order bit of the byte at the same location); chars are stored in the high-order byte of the word; and reals are in the KCS standard format.

The QuickDraw simple data types listed below are constructed out of these fundamental types.

Type	Size
QDPtr	Long (4 bytes)
QDHandle	Long (4 bytes)
Word	Long (4 bytes)
Str255	Page (256 bytes)
Pattern	8 bytes
Bits16	32 bytes

Other data types are constructed as records of variables of the above types. The size of such a type is the sum of the sizes of all the fields in the record; the fields appear in the variable with the first field in the lowest address. For example, consider the data type BitMap, which is defined as follows:

```

type BitMap = record
    baseAddr: QDPtr;
    rowBytes: integer;
    bounds: Rect
end;

```

This data type would be arranged in memory as seven words: a long for the baseAddr, a word for the rowBytes, and four words for the top, left, right, and bottom parts of the bounds rectangle. To assist you in referring to the fields inside a variable that has a structure like this, the GRAFTYPES.TEXT file defines constants that you can use as offsets into the fields of a structured variable. For example, to move a bitMap's rowBytes value into D3, you would execute the following instruction:

```
MOVE.W MYBITMAP+ROWBYTES,D3
```

Displacements are given in the GRAFTYPES.TEXT file for all fields of all data types defined by QuickDraw.

To do double indirection, you perform an LEA indirectly to obtain the effective address from the handle. For example, to get at the top coordinate of a region's enclosing rectangle:

```

MOVE.L MYHANDLE,A1           ; Load handle into A1
MOVE.L (A1),A1               ; Use handle to get pointer
MOVE.W RGNBBOX+TOP(A1),D3    ; Load value using pointer

```

WARNING

For regions (and all other variable-length structures with handles), you must not move the pointer into a register once and just continue to use that pointer; you must do the double indirection each time. Every QuickDraw call you make can possibly trigger a heap compaction that renders all pointers to movable heap items (like regions) invalid. The handles will remain valid, but pointers you've obtained through handles can be rendered invalid at any subroutine call or trap in your program.

E.11.3 Global Variables

Register A5 always points to the section of memory where global variables are stored. The GRAFTYPES.TEXT file defines a constant GRAFGLOB that points to the beginning of the QuickDraw variables in this space, and other constants that point to the individual variables. To access one of the variables, put GRAFGLOB in an address register, sum the constants, and index off of that register. For example, if you want to know the horizontal coordinate of the pen

location for the current grafPort, which the global variable thePort points to, you can give the following instructions:

```

MOVE.L GRAFGLOB(A5),A0 ; Point to QuickDraw globals
MOVE.L THEPORT(A0),A1  ; Get current grafPort
MOVE.W PNLOC+H(A1),D0  ; Get thePort .pnLoc.h

```

E.11.4 Procedures and Functions

To call a QuickDraw procedure or function, you must push all parameters to it on the stack, then JSR to the function or procedure. When you link your program with QuickDraw, these JSRs are adjusted to refer to QuickDraw's jump table, so that a JSR into the table redirects you to the actual location of the procedure or function.

The only difficult part about calling QuickDraw procedures and functions is stacking the parameters. You must follow some strict rules:

- Save all registers you wish to preserve before you begin pushing parameters. Any QuickDraw procedure or function can destroy the contents of the registers A0, A1, D0, D1, and D2, but the others are never altered.
- Push the parameters in the order that they appear in the Pascal procedural interface.
- For booleans, push a byte; for integers and characters, push a word; for pointers, handles, long integers, and reals, push a long.
- For any structured variable longer than four (4) bytes, push a pointer to the variable.
- For all var parameters, regardless of size, push a pointer to the variable.
- When calling a function, first push a null entry equal to the size of the function result, then push all other parameters. The result will be left on the stack after the function returns to you.

This makes for a lengthy interface, but it also guarantees that you can mock up a Pascal version of your program, and later translate it into assembly code that works the same. For example, the Pascal statement

```
blackness := GetPixel(50,mousePos.v);
```

would be written in assembly language like this:

```

CLR.W    -(SP)           ; Save space for boolean result
MOVE.W   #50,-(SP)       ; Push constant 50 (decimal)
MOVE.W   MOUSEPOS+V,-(SP) ; Push the value of mousePos.v
JSR      GETPIXEL        ; Call routine
MOVE.W   (SP)+,BLACKNESS ; Fetch result from stack

```

This is a simple example, pushing and pulling word-long constants. Normally, you'll be pushing more pointers, using the PEA (Push Effective Address) instruction:

```
FillRoundRect(myRect,1,thePort^.pnSize.v,white);

PEA    MYRECT           ; Push pointer to myRect
MOVE.W #1,-(SP)        ; Push constant 1
MOVE.L GRAFGLDB(A5),A0 ; Point to QuickDraw globals
MOVE.L THEPORT(A0),A1  ; Get current grafPort
MOVE.W PNSIZE+V(A1),-(SP) ; Push value of thePort^.pnSize.v
PEA    WHITE(A0)       ; Push pointer to global variable white
JSR    FILLROUNDRECT   ; Call the subroutine
```

To call the TextFace procedure, push a word in which each of seven bits represents a stylistic variation: set bit 0 for bold, bit 1 for italic, bit 2 for underline, bit 3 for outline, bit 4 for shadow, bit 5 for condense, and bit 6 for extend.

E.12 Summary of QuickDraw

```
UNIT QuickDraw;
```

```
{ Copyright 1983 Apple Computer Inc. }
```

```
INTERFACE
```

```
CONST srcCopy      = 0; { the 16 transfer modes }
      srcOr        = 1;
      srcXor       = 2;
      srcBic       = 3;
      notSrcCopy   = 4;
      notSrcOr     = 5;
      notSrcXor    = 6;
      notSrcBic    = 7;
      patCopy      = 8;
      patOr        = 9;
      patXor       = 10;
      patBic       = 11;
      notPatCopy   = 12;
      notPatOr     = 13;
      notPatXor    = 14;
      notPatBic    = 15;
```

```
{ QuickDraw color separation constants }
```

```
normalBit   = 0;      { normal screen mapping }
inverseBit  = 1;      { inverse screen mapping }
redBit      = 4;      { RGB additive mapping }
greenBit    = 3;
blueBit     = 2;
cyanBit     = 8;      { CMYBk subtractive mapping }
magentaBit  = 7;
yellowBit   = 6;
blackBit    = 5;
```

```
blackColor  = 33;     { colors expressed in these mappings }
whiteColor  = 30;
redColor    = 205;
greenColor  = 341;
blueColor   = 409;
cyanColor   = 273;
magentaColor = 137;
yellowColor = 69;
```

```
picLParen  = 0;      { standard picture comments }
picRParen  = 1;
```

```
TYPE QDByte   = -128..127;
     QDPtr    = ^QDByte;          { blind pointer }
     QDHandle = ^QDPtr;          { blind handle }
     Str255   = String[255];
     Pattern  = PACKED ARRAY[0..7] OF 0..255;
     Bits16   = ARRAY[0..15] OF INTEGER;
     VHSelect = (v,h);
     GrafVerb = (frame,paint,erase,invert,fill);
     StyleItem = (bold,italic,underline,outline,shadow,condense,extend);
     Style     = SET OF StyleItem;

     FontInfo = RECORD
         ascent: INTEGER;
         descent: INTEGER;
         widMax: INTEGER;
         leading: INTEGER;
     END;
```

```
Point = RECORD CASE INTEGER OF
    0: (v: INTEGER;
        h: INTEGER);
    1: (vh: ARRAY[VHSelect] OF INTEGER);
END;
```

```
Rect = RECORD CASE INTEGER OF
    0: (top:    INTEGER;
        left:   INTEGER;
        bottom: INTEGER;
        right:  INTEGER);
    1: (topLeft: Point;
        botRight: Point);
END;
```

```
Bitmap = RECORD
    baseAddr: QDPtr;
    rowBytes: INTEGER;
    bounds:   Rect;
END;
```

```
Cursor = RECORD
    data:   Bits16;
    mask:   Bits16;
    hotSpot: Point;
END;
```

```
PenState = RECORD
    pnLoc:   Point;
    pnSize:  Point;
    pnMode:  INTEGER;
    pnPat:   Pattern;
END;
```



```
PolyHandle = ^PolyPtr;  
PolyPtr    = ^Polygon;  
Polygon    = RECORD  
    polySize:  INTEGER;  
    polyBBox:  Rect;  
    polyPoints: ARRAY[0..0] OF Point;  
END;
```

```
RgnHandle = ^RgnPtr;  
RgnPtr    = ^Region;  
Region    = RECORD  
    rgnSize:  INTEGER; { rgnSize = 10 for rectangular }  
    rgnBBox:  Rect;  
    { plus more data if not rectangular }  
END;
```

```
PicHandle = ^PicPtr;  
PicPtr    = ^Picture;  
Picture   = RECORD  
    picSize:  INTEGER;  
    picFrame: Rect;  
    { plus byte codes for picture content }  
END;
```

```
QDProcsPtr = ^QDProcs;  
QDProcs    = RECORD  
    textProc:  QDPtr;  
    lineProc:  QDPtr;  
    rectProc:  QDPtr;  
    rRectProc: QDPtr;  
    ovalProc:  QDPtr;  
    arcProc:   QDPtr;  
    polyProc:  QDPtr;  
    rgnProc:   QDPtr;  
    bitsProc:  QDPtr;  
    commentProc: QDPtr;  
    txMeasProc: QDPtr;  
    getPicProc: QDPtr;  
    putPicProc: QDPtr;  
END;
```

```
GrafPtr = ^GrafPort;
GrafPort = RECORD
    device:      INTEGER;
    portBits:    BitMap;
    portRect:    Rect;
    visRgn:      RgnHandle;
    clipRgn:     RgnHandle;
    bkPat:       Pattern;
    fillPat:     Pattern;
    pnLoc:       Point;
    pnSize:      Point;
    pnMode:      INTEGER;
    pnPat:       Pattern;
    pnVis:       INTEGER;
    txFont:      INTEGER;
    txFace:      Style;
    txMode:      INTEGER;
    txSize:      INTEGER;
    spExtra:     INTEGER;
    fgColor:     LongInt;
    bkColor:     LongInt;
    colrBit:     INTEGER;
    patStretch:  INTEGER;
    picSave:     QDHandle;
    rgnSave:     QDHandle;
    polySave:    QDHandle;
    grafProcs:  QDProcsPtr;
END;
```

```
VAR thePort: GrafPtr;
    white: Pattern;
    black: Pattern;
    gray: Pattern;
    ltGray: Pattern;
    dkGray: Pattern;
    arrow: Cursor;
    screenBits: BitMap;
    randSeed: LongInt;
```

{ GrafPort Routines }

```
PROCEDURE InitGraf (globalPtr: QDPtr);
PROCEDURE OpenPort (port: GrafPtr);
PROCEDURE InitPort (port: GrafPtr);
PROCEDURE ClosePort (port: GrafPtr);
PROCEDURE SetPort (port: GrafPtr);
PROCEDURE GetPort (VAR port: GrafPtr);
PROCEDURE GrafDevice (device: INTEGER);
PROCEDURE SetPortBits (bm: BitMap);
PROCEDURE PortSize (width,height: INTEGER);
PROCEDURE MovePortTo (leftGlobal,topGlobal: INTEGER);
PROCEDURE SetOrigin (h,v: INTEGER);
PROCEDURE SetClip (rgn: RgnHandle);
PROCEDURE GetClip (rgn: RgnHandle);
PROCEDURE ClipRect (r: Rect);
PROCEDURE BackPat (pat: Pattern);
```

{ Cursor Routines }

```
PROCEDURE InitCursor;
PROCEDURE SetCursor (crsr: Cursor);
PROCEDURE HideCursor;
PROCEDURE ShowCursor;
PROCEDURE ObscureCursor;
```

{ Line Routines }

```
PROCEDURE HidePen;
PROCEDURE ShowPen;
PROCEDURE GetPen (VAR pt: Point);
PROCEDURE GetPenState (VAR pnState: PenState);
PROCEDURE SetPenState (pnState: PenState);
PROCEDURE PenSize (width,height: INTEGER);
PROCEDURE PenMode (mode: INTEGER);
PROCEDURE PenPat (pat: Pattern);
PROCEDURE PenNormal;
PROCEDURE MoveTo (h,v: INTEGER);
PROCEDURE Move (dh,dv: INTEGER);
PROCEDURE LineTo (h,v: INTEGER);
PROCEDURE Line (dh,dv: INTEGER);
```

{ Text Routines }

```

PROCEDURE TextFont      (font: INTEGER);
PROCEDURE TextFace     (face: Style);
PROCEDURE TextMode     (mode: INTEGER);
PROCEDURE TextSize     (size: INTEGER);
PROCEDURE SpaceExtra   (extra: INTEGER);
PROCEDURE DrawChar     (ch: char);
PROCEDURE DrawString   (s: Str255);
PROCEDURE DrawText     (textBuf: QDPtr; firstByte,byteCount: INTEGER);
FUNCTION CharWidth     (ch: CHAR): INTEGER;
FUNCTION StringWidth   (s: Str255): INTEGER;
FUNCTION TextWidth     (textBuf: QDPtr; firstByte,byteCount: INTEGER):
    INTEGER;
PROCEDURE GetFontInfo  (VAR info: FontInfo);

```

{ Point Calculations }

```

PROCEDURE AddPt        (src: Point; VAR dst: Point);
PROCEDURE SubPt        (src: Point; VAR dst: Point);
PROCEDURE SetPt        (VAR pt: Point; h,v: INTEGER);
FUNCTION EqualPt       (pt1,pt2: Point): BOOLEAN;
PROCEDURE ScalePt      (VAR pt: Point;   fromRect,toRect: Rect);
PROCEDURE MapPt        (VAR pt: Point;   fromRect,toRect: Rect);
PROCEDURE LocalToGlobal (VAR pt: Point);
PROCEDURE GlobalToLocal (VAR pt: Point);

```

{ Rectangle Calculations }

```

PROCEDURE SetRect      (VAR r: Rect; left,top,right,bottom: INTEGER);
FUNCTION EqualRect     (rect1,rect2: Rect): BOOLEAN;
FUNCTION EmptyRect     (r: Rect): BOOLEAN;
PROCEDURE OffsetRect   (VAR r: Rect; dh,dv: INTEGER);
PROCEDURE MapRect      (VAR r: Rect;   fromRect,toRect: Rect);
PROCEDURE InsetRect    (VAR r: Rect; dh,dv: INTEGER);
FUNCTION SectRect      (src1,src2: Rect; VAR dstRect: Rect): BOOLEAN;
PROCEDURE UnionRect    (src1,src2: Rect; VAR dstRect: Rect);
FUNCTION PtInRect      (pt: Point; r: Rect): BOOLEAN;
PROCEDURE Pt2Rect      (pt1,pt2: Point; VAR dstRect: Rect);

```

{ Graphical Operations on Rectangles }

```
PROCEDURE FrameRect (r: Rect);
PROCEDURE PaintRect (r: Rect);
PROCEDURE EraseRect (r: Rect);
PROCEDURE InvertRect (r: Rect);
PROCEDURE FillRect (r: Rect; pat: Pattern);
```

{ RoundRect Routines }

```
PROCEDURE FrameRoundRect (r: Rect; ovWd, ovHt: INTEGER);
PROCEDURE PaintRoundRect (r: Rect; ovWd, ovHt: INTEGER);
PROCEDURE EraseRoundRect (r: Rect; ovWd, ovHt: INTEGER);
PROCEDURE InvertRoundRect (r: Rect; ovWd, ovHt: INTEGER);
PROCEDURE FillRoundRect (r: Rect; ovWd, ovHt: INTEGER; pat: Pattern);
```

{ Oval Routines }

```
PROCEDURE FrameOval (r: Rect);
PROCEDURE PaintOval (r: Rect);
PROCEDURE EraseOval (r: Rect);
PROCEDURE InvertOval (r: Rect);
PROCEDURE FillOval (r: Rect; pat: Pattern);
```

{ Arc Routines }

```
PROCEDURE FrameArc (r: Rect; startAngle, arcAngle: INTEGER);
PROCEDURE PaintArc (r: Rect; startAngle, arcAngle: INTEGER);
PROCEDURE EraseArc (r: Rect; startAngle, arcAngle: INTEGER);
PROCEDURE InvertArc (r: Rect; startAngle, arcAngle: INTEGER);
PROCEDURE FillArc (r: Rect; startAngle, arcAngle: INTEGER; pat:
Pattern);
PROCEDURE PtToAngle (r: Rect; pt: Point; VAR angle: INTEGER);
```

{ Polygon Routines }

```

FUNCTION OpenPoly: PolyHandle;
PROCEDURE ClosePoly;
PROCEDURE KillPoly (poly: PolyHandle);
PROCEDURE OffsetPoly (poly: PolyHandle; dh, dv: INTEGER);
PROCEDURE MapPoly (poly: PolyHandle; fromRect, toRect: Rect);
PROCEDURE FramePoly (poly: PolyHandle);
PROCEDURE PaintPoly (poly: PolyHandle);
PROCEDURE ErasePoly (poly: PolyHandle);
PROCEDURE InvertPoly (poly: PolyHandle);
PROCEDURE FillPoly (poly: PolyHandle; pat: Pattern);

```

{ Region Calculations }

```

FUNCTION NewRgn: RgnHandle;
PROCEDURE DisposeRgn(rgn: RgnHandle);
PROCEDURE CopyRgn (srcRgn, dstRgn: RgnHandle);
PROCEDURE SetEmptyRgn(rgn: RgnHandle);
PROCEDURE SetRectRgn(rgn: RgnHandle; left, top, right, bottom: INTEGER);
PROCEDURE RectRgn (rgn: RgnHandle; r: Rect);
PROCEDURE OpenRgn;
PROCEDURE CloseRgn (dstRgn: RgnHandle);
PROCEDURE OffsetRgn (rgn: RgnHandle; dh, dv: INTEGER);
PROCEDURE MapRgn (rgn: RgnHandle; fromRect, toRect: Rect);
PROCEDURE InsetRgn (rgn: RgnHandle; dh, dv: INTEGER);
PROCEDURE SectRgn (srcRgnA, srcRgnB, dstRgn: RgnHandle);
PROCEDURE UnionRgn (srcRgnA, srcRgnB, dstRgn: RgnHandle);
PROCEDURE DiffRgn (srcRgnA, srcRgnB, dstRgn: RgnHandle);
PROCEDURE XorRgn (srcRgnA, srcRgnB, dstRgn: RgnHandle);
FUNCTION EqualRgn (rgnA, rgnB: RgnHandle): BOOLEAN;
FUNCTION EmptyRgn (rgn: RgnHandle): BOOLEAN;
FUNCTION PtInRgn (pt: Point; rgn: RgnHandle): BOOLEAN;
FUNCTION RectInRgn (r: Rect; rgn: RgnHandle): BOOLEAN;

```

{ Graphical Operations on Regions }

```

PROCEDURE FrameRgn (rgn: RgnHandle);
PROCEDURE PaintRgn (rgn: RgnHandle);
PROCEDURE EraseRgn (rgn: RgnHandle);
PROCEDURE InvertRgn (rgn: RgnHandle);
PROCEDURE FillRgn (rgn: RgnHandle; pat: Pattern);

```

```
{ Graphical Operations on BitMaps }
```

```
PROCEDURE ScrollRect(dstRect: Rect; dh,dv: INTEGER; updateRgn:
    RgnHandle);
PROCEDURE CopyBits (srcBits,dstBits: BitMap;
    srcRect,dstRect: Rect;
    mode:          INTEGER;
    maskRgn:      RgnHandle);
```

```
{ Picture Routines }
```

```
FUNCTION OpenPicture(picFrame: Rect): PicHandle;
PROCEDURE ClosePicture;
PROCEDURE DrawPicture(myPicture: PicHandle; dstRect: Rect);
PROCEDURE PicComment(kind,dataSize: INTEGER; dataHandle: QDHandle);
PROCEDURE KillPicture(myPicture: PicHandle);
```

```
{ The Bottleneck Interface: }
```

```
PROCEDURE SetStdProcs(VAR procs: QDProcs);
PROCEDURE StdText (count: INTEGER; textAddr: QDPtr; numer,denom:
    Point);
PROCEDURE StdLine (newPt: Point);
PROCEDURE StdRect (verb: GrafVerb; r: Rect);
PROCEDURE StdRRect (verb: GrafVerb; r: Rect; ovWd,ovHt: INTEGER);
PROCEDURE StdOval (verb: GrafVerb; r: Rect);
PROCEDURE StdArc (verb: GrafVerb; r: Rect; startAngle,arcAngle:
    INTEGER);
PROCEDURE StdPoly (verb: GrafVerb; poly: PolyHandle);
PROCEDURE StdRgn (verb: GrafVerb; rgn: RgnHandle);
PROCEDURE StdBits (VAR srcBits: BitMap; VAR srcRect,dstRect: Rect;
    mode: INTEGER; maskRgn: RgnHandle);
PROCEDURE StdComment (kind,dataSize: INTEGER; dataHandle: QDHandle);
FUNCTION StdTxMeas (count: INTEGER; textAddr: QDPtr; VAR numer,denom:
    Point; VAR info: FontInfo): INTEGER;
PROCEDURE StdGetPic (dataPtr: QDPtr; byteCount: INTEGER);
PROCEDURE StdPutPic (dataPtr: QDPtr; byteCount: INTEGER);
```

```
{ Misc Utility Routines }
```

```
FUNCTION GetPixel (h,v: INTEGER): BOOLEAN;
FUNCTION Random: INTEGER;
PROCEDURE StuffHex (thingptr: QDPtr; s:Str255);
PROCEDURE ForeColor (color: LongInt);
PROCEDURE BackColor (color: LongInt);
PROCEDURE ColorBit (whichBit: INTEGER);
```

```
IMPLEMENTATION
```

```
{ $I QuickDraw2.text }
```

```
{ QuickDraw2.text: Implementation part of QuickDraw }
```

```
{ $S Graf }
```

```
TYPE FMOutPtr = ^FMOutRec;
      FMOutrec = PACKED RECORD
          errNum:    INTEGER;    { used only for GrafError
          fontHandle: QDHandle;  { handle to font
          bold:      0..255;     { how much to smear horiz
          italic:    0..255;     { how much to shear
          ulOffset:  0..255;     { pixels below baseline
          ulShadow:  0..255;     { how big is the halo
          ulThick:   0..255;     { how thick is the underline
          shadow:    0..255;     { 0,1,2, or 3 only
          extra:     -128..127;   { extra white dots each char
          ascent:    0..255;     { ascent measure for font
          descent:   0..255;     { descent measure for font
          widMax:    0..255;     { width of widest char
          leading:   -128..127;   { leading between lines
          unused:    0..255;
          numer:     Point;      { use this modified scale to
          denom:     Point;      { draw or measure text with
      END;
```



```

VAR wideOpen: RgnHandle; { a dummy rectangular region, read-only }
    wideMaster: RgnPtr;
    wideData: Region;
    rgnBuf: QDHandle; { point saving buffer for OpenRgn }
    rgnIndex: INTEGER; { current bytes used in rgnBuf }
    rgnMax: INTEGER; { max bytes allocated so far to rgnBuf }
    playPic: PicHandle; { used by StdGetPic }
    playIndex: INTEGER; { used by StdGetPic }
    thePoly: PolyHandle; { the current polygon being defined }
    polyMax: INTEGER; { max bytes allocated so far to thePoly }
    patAlign: Point; { to align pattern during DrawPicture }
    fontPtr: FMOutPtr; { the last font used, used by DrawText }
    fontData: FMOutRec;

```

```
{ grafPort routines }
```

```

PROCEDURE InitGraf;           EXTERNAL;
PROCEDURE OpenPort;          EXTERNAL;
PROCEDURE InitPort;          EXTERNAL;
PROCEDURE ClosePort;         EXTERNAL;
PROCEDURE GrafDevice;        EXTERNAL;
PROCEDURE SetPort;           EXTERNAL;
PROCEDURE GetPort;           EXTERNAL;
PROCEDURE SetPortBits;       EXTERNAL;
PROCEDURE PortSize;          EXTERNAL;
PROCEDURE MovePortTo;        EXTERNAL;
PROCEDURE SetOrigin;         EXTERNAL;
PROCEDURE SetClip;           EXTERNAL;
PROCEDURE GetClip;           EXTERNAL;
PROCEDURE ClipRect;          EXTERNAL;
PROCEDURE BackPat;           EXTERNAL;

```

```
{ cursor routines }
```

```

PROCEDURE InitCursor;        EXTERNAL;
PROCEDURE SetCursor;         EXTERNAL;
PROCEDURE HideCursor;        EXTERNAL;
PROCEDURE ShowCursor;        EXTERNAL;
PROCEDURE ObscureCursor;     EXTERNAL;

```

{ text routines }

PROCEDURE TextFont;	EXTERNAL;
PROCEDURE TextFace;	EXTERNAL;
PROCEDURE TextMode;	EXTERNAL;
PROCEDURE TextSize;	EXTERNAL;
PROCEDURE SpaceExtra;	EXTERNAL;
PROCEDURE DrawChar;	EXTERNAL;
PROCEDURE DrawString;	EXTERNAL;
PROCEDURE DrawText;	EXTERNAL;
FUNCTION CharWidth;	EXTERNAL;
FUNCTION StringWidth;	EXTERNAL;
FUNCTION TextWidth;	EXTERNAL;
PROCEDURE GetFontInfo;	EXTERNAL;

{ line routines }

PROCEDURE HidePen;	EXTERNAL;
PROCEDURE ShowPen;	EXTERNAL;
PROCEDURE GetPen;	EXTERNAL;
PROCEDURE GetPenState;	EXTERNAL;
PROCEDURE SetPenState;	EXTERNAL;
PROCEDURE PenSize;	EXTERNAL;
PROCEDURE PenMode;	EXTERNAL;
PROCEDURE PenPat;	EXTERNAL;
PROCEDURE PenNormal;	EXTERNAL;
PROCEDURE MoveTo;	EXTERNAL;
PROCEDURE Move;	EXTERNAL;
PROCEDURE LineTo;	EXTERNAL;
PROCEDURE Line;	EXTERNAL;

{ rectangle calculations }

PROCEDURE SetRect;	EXTERNAL;
FUNCTION EqualRect;	EXTERNAL;
FUNCTION EmptyRect;	EXTERNAL;
PROCEDURE OffsetRect;	EXTERNAL;
PROCEDURE MapRect;	EXTERNAL;
PROCEDURE InsetRect;	EXTERNAL;
FUNCTION SectRect;	EXTERNAL;
PROCEDURE UnionRect;	EXTERNAL;
FUNCTION PtInRect;	EXTERNAL;
PROCEDURE Pt2Rect;	EXTERNAL;

{ graphical operations on rectangles }

PROCEDURE FrameRect;	EXTERNAL;
PROCEDURE PaintRect;	EXTERNAL;
PROCEDURE EraseRect;	EXTERNAL;
PROCEDURE InvertRect;	EXTERNAL;
PROCEDURE FillRect;	EXTERNAL;

{ graphical operations on RoundRects }

PROCEDURE FrameRoundRect;	EXTERNAL;
PROCEDURE PaintRoundRect;	EXTERNAL;
PROCEDURE EraseRoundRect;	EXTERNAL;
PROCEDURE InvertRoundRect;	EXTERNAL;
PROCEDURE FillRoundRect;	EXTERNAL;

{ graphical operations on Ovals }

PROCEDURE FrameOval;	EXTERNAL;
PROCEDURE PaintOval;	EXTERNAL;
PROCEDURE EraseOval;	EXTERNAL;
PROCEDURE InvertOval;	EXTERNAL;
PROCEDURE FillOval;	EXTERNAL;

{ Arc routines }

PROCEDURE FrameArc;	EXTERNAL;
PROCEDURE PaintArc;	EXTERNAL;
PROCEDURE EraseArc;	EXTERNAL;
PROCEDURE InvertArc;	EXTERNAL;
PROCEDURE FillArc;	EXTERNAL;
PROCEDURE PtToAngle;	EXTERNAL;

{ polygon routines }

FUNCTION	OpenPoly;	EXTERNAL;
PROCEDURE	ClosePoly;	EXTERNAL;
PROCEDURE	KillPoly;	EXTERNAL;
PROCEDURE	OffsetPoly;	EXTERNAL;
PROCEDURE	MapPoly;	EXTERNAL;

PROCEDURE	FramePoly;	EXTERNAL;
PROCEDURE	PaintPoly;	EXTERNAL;
PROCEDURE	ErasePoly;	EXTERNAL;
PROCEDURE	InvertPoly;	EXTERNAL;
PROCEDURE	FillPoly;	EXTERNAL;

{ region calculations }

FUNCTION	NewRgn;	EXTERNAL;
PROCEDURE	DisposeRgn;	EXTERNAL;
PROCEDURE	OpenRgn;	EXTERNAL;
PROCEDURE	CloseRgn;	EXTERNAL;
PROCEDURE	OffsetRgn;	EXTERNAL;
PROCEDURE	MapRgn;	EXTERNAL;
PROCEDURE	InsetRgn;	EXTERNAL;
PROCEDURE	SectRgn;	EXTERNAL;
PROCEDURE	CopyRgn;	EXTERNAL;
PROCEDURE	SetEmptyRgn;	EXTERNAL;
PROCEDURE	SetRectRgn;	EXTERNAL;
PROCEDURE	RectRgn;	EXTERNAL;
PROCEDURE	UnionRgn;	EXTERNAL;
PROCEDURE	DiffRgn;	EXTERNAL;
PROCEDURE	XorRgn;	EXTERNAL;
FUNCTION	EqualRgn;	EXTERNAL;
FUNCTION	EmptyRgn;	EXTERNAL;
FUNCTION	PtInRgn;	EXTERNAL;
FUNCTION	RectInRgn;	EXTERNAL;

{ graphical operations on Regions }

PROCEDURE	FrameRgn;	EXTERNAL;
PROCEDURE	PaintRgn;	EXTERNAL;
PROCEDURE	EraseRgn;	EXTERNAL;
PROCEDURE	InvertRgn;	EXTERNAL;
PROCEDURE	FillRgn;	EXTERNAL;

{ BitMap routines }

PROCEDURE CopyBits; EXTERNAL;
PROCEDURE ScrollRect; EXTERNAL;

{ Picture routines }

FUNCTION OpenPicture; EXTERNAL;
PROCEDURE ClosePicture; EXTERNAL;
PROCEDURE KillPicture; EXTERNAL;
PROCEDURE DrawPicture; EXTERNAL;
PROCEDURE PicComment; EXTERNAL;

{ BottleNeck routines }

PROCEDURE StdText; EXTERNAL;
PROCEDURE StdLine; EXTERNAL;
PROCEDURE StdRect; EXTERNAL;
PROCEDURE StdRRect; EXTERNAL;
PROCEDURE StdOval; EXTERNAL;
PROCEDURE StdArc; EXTERNAL;
PROCEDURE StdPoly; EXTERNAL;
PROCEDURE StdRgn; EXTERNAL;
PROCEDURE StdBits; EXTERNAL;
PROCEDURE StdComment; EXTERNAL;
FUNCTION StdTxMeas; EXTERNAL;
PROCEDURE StdGetPic; EXTERNAL;
PROCEDURE StdPutPic; EXTERNAL;

{ misc utility routines }

FUNCTION GetPixel; EXTERNAL;
FUNCTION Random; EXTERNAL;
PROCEDURE AddPt; EXTERNAL;
PROCEDURE SubPt; EXTERNAL;
PROCEDURE SetPt; EXTERNAL;
FUNCTION EqualPt; EXTERNAL;
PROCEDURE StuffHex; EXTERNAL;
PROCEDURE LocalToGlobal; EXTERNAL;
PROCEDURE GlobalToLocal; EXTERNAL;

```
PROCEDURE ScalePt;           EXTERNAL;  
PROCEDURE MapPt;            EXTERNAL;  
PROCEDURE ForeColor;       EXTERNAL;  
PROCEDURE BackColor;      EXTERNAL;  
PROCEDURE ColorBit;       EXTERNAL;  
PROCEDURE SetStdProcs;    EXTERNAL;
```

```
END. { of UNIT }
```

E.13 QuickDraw Sample Program

```
{QDSSample
```

```
▣
```

```
gQDSSample
```

```
x#5:iulinker
```

```
-P
```

```
obj:HWIntL
```

```
obj:Unit68K
```

```
obj:UnitStd
```

```
obj:UnitHz
```

```
obj:FontMgr
```

```
obj:FM68K
```

```
obj:Storage
```

```
obj:QuickDraw
```

```
obj:GrafLib
```

```
obj:QDSupport
```

```
QDSSample
```

```
*iospaslib
```

```
QDSSample
```

```
{{ the above junk is an exec file to compile and run this program }
```

```
PROGRAM QDSSample;
```

```
{ $U - }
```

```
USES { $U obj:QuickDraw } QuickDraw,  
     { $U obj:QDSupport } QDSupport;
```

```
TYPE IconData = ARRAY[0..95] OF INTEGER;
```

```
VAR heapBuf:    ARRAY[0..10000] OF INTEGER;
    myPort:    GrafPort;
    icons:     ARRAY[0..5] OF IconData;
    errNum:    INTEGER;
```

```
FUNCTION HeapFull(hz: QDPtr; bytesNeeded: INTEGER): INTEGER;
{ this function will be called if the heapZone runs out of space }
BEGIN
  WRITELN('The heap is full. User Croak !! ');
  Halt;
END;
```

```
PROCEDURE InitIcons;
{ Manually stuff some icons. Normally we would read them from a file }
BEGIN
```

```
  { Lisa }
```

```
  StuffHex(@icons[0, 0], '000000000000000000000000000000001FFFFFFFFC');
  StuffHex(@icons[0, 12], '006000000006018000000000B0600000000130FFFFFFFFFA3');
  StuffHex(@icons[0, 24], '18000000004311FFFFFF00023120000080F231200000BF923');
  StuffHex(@icons[0, 36], '120000080F23120000080023120000080023120000080F23');
  StuffHex(@icons[0, 48], '1200000BF923120000080F2312000008002311FFFFFF00023');
  StuffHex(@icons[0, 60], '08000000004307FFFFFFFA30100000000260FFFFFFFFFE2C');
  StuffHex(@icons[0, 72], '18000000013832AAAAA8A9F0655555515380C2AAAA82A580');
  StuffHex(@icons[0, 84], '800000000980FFFFFFFF300800000000160FFFFFFFFFC00');
```

```
  { Printer }
```

```
  StuffHex(@icons[1, 0], '0000000000000000000000000000000000000000');
  StuffHex(@icons[1, 12], '00000000000000007FFFFFF0000080000280000111514440');
  StuffHex(@icons[1, 24], '0002000008400004454510400004000017C00004A5151000');
  StuffHex(@icons[1, 36], '0004000010000004A54510000004000017FE00F4A5151003');
  StuffHex(@icons[1, 48], '0184000013870327FFFFFF10F0640000002180CFFFFFFFFC37');
  StuffHex(@icons[1, 60], '18000000006B3000000000D77FFFFFFFFFABC00000000356');
  StuffHex(@icons[1, 72], '8000000001AC87F000000158841000CCC1B087F000CCC160');
  StuffHex(@icons[1, 84], '8000000001C0C000000003807FFFFFFFFF0007800001E000');
```

```

{ Trash Can }
StuffHex(@icons[2, 0], '000001FC000000000E0600000000300300000000C0918000');
StuffHex(@icons[2, 12], '00013849800000026C4980000004C0930000000861260000');
StuffHex(@icons[2, 24], '0010064FE0000031199830000020E6301800002418E00800');
StuffHex(@icons[2, 36], '0033E3801C0000180E002C00000FF801CC0000047FFE0C00');
StuffHex(@icons[2, 48], '000500004C000005259A4C000005250A4C00000525FA4C00');
StuffHex(@icons[2, 60], '000524024C00000524924C00600524924C0090E524924C7C');
StuffHex(@icons[2, 72], '932524924C82A44524924D01C88524924CF10C4524924C09');
StuffHex(@icons[2, 84], '0784249258E70003049233100000E000E40800001FFFC3F0');

{ tray }
StuffHex(@icons[3, 0], '00000000000000000000000000000000000000000000000000000000');
StuffHex(@icons[3, 12], '0000000000000000000000000000000000000000000000007FFFFFF0');
StuffHex(@icons[3, 24], '000E00000018001A00000038003600000078006A0000000D8');
StuffHex(@icons[3, 36], '0007FFFFFFB801AC0000035803580000068807FC000FFD58');
StuffHex(@icons[3, 48], '040600180AB80403FFF00D58040000000AB8040000000D58');
StuffHex(@icons[3, 60], '040000000AB807FFFFFFD5806AC00000AB8055800000D58');
StuffHex(@icons[3, 72], '068000000AB807FC000FFD70040600180AE00403FFF00DC');
StuffHex(@icons[3, 84], '040000000B80040000000F00040000000E0007FFFFFFFC00');

{ File Cabinet }
StuffHex(@icons[4, 0], '0007FFFFFFC00000800000C00001000001C00002000003400');
StuffHex(@icons[4, 12], '004000006C0000FFFFFFD40000800000AC0000BFFFFFFD400');
StuffHex(@icons[4, 24], '00A00002AC0000A07F02D40000A04102AC0000A07F02D400');
StuffHex(@icons[4, 36], '00A00002AC0000A08082D40000A0FF82AC0000A00002D400');
StuffHex(@icons[4, 48], '00A00002AC0000BFFFFFFD40000800000AC0000BFFFFFFD400');
StuffHex(@icons[4, 60], '00A00002AC0000A07F02D40000A04102AC0000A07F02D400');
StuffHex(@icons[4, 72], '00A00002AC0000A08082D40000A0FF82AC0000A00002D800');
StuffHex(@icons[4, 84], '00A00002B00000BFFFFFFE00000800000C00000FFFFFF8000');

{ drawer }
StuffHex(@icons[5, 0], '00000000000000000000000000000000000000000000000000000000');
StuffHex(@icons[5, 12], '00000000000000000000000000000000000000000000000000000000');
StuffHex(@icons[5, 24], '00000000000000000000000000000000000000000000000000000000');
StuffHex(@icons[5, 36], '0000000000000000000000000000000000000000000000001FFFFFF0');
StuffHex(@icons[5, 48], '0000380000300000680000700000D80000D0003FFFFFF180');
StuffHex(@icons[5, 60], '00200000135000200000168000201FE01D50002010201A80');
StuffHex(@icons[5, 72], '00201FE01560002000001AC0002000001580002020101800');
StuffHex(@icons[5, 84], '00203FF01600002000001C00002000001800003FFFFFF000');

```

END;


```
PROCEDURE DrawIcon(whichIcon, h, v: INTEGER);
VAR srcBits: BitMap;
    srcRect, dstRect: Rect;
BEGIN
  srcBits.baseAddr:=@icons[whichIcon];
  srcBits.rowBytes:=6;
  SetRect(srcBits.bounds, 0, 0, 48, 32);
  srcRect:=srcBits.bounds;
  dstRect:=srcRect;
  OffsetRect(dstRect, h, v);
  CopyBits(srcBits, thePort^.portBits, srcRect, dstRect, srcOr, Nil);
END;
```

```
PROCEDURE DrawStuff;
```

```
VAR i: INTEGER;
    tempRect: Rect;
    myPoly: PolyHandle;
    myRgn: RgnHandle;
    myPattern: Pattern;
```

```
BEGIN
```

```
  StuffHex(@myPattern, '8040200002040800');
```

```
  tempRect := thePort^.portRect;
  ClipRect(tempRect);
  EraseRoundRect(tempRect, 30, 20);
  FrameRoundRect(tempRect, 30, 20);
```

```
  { draw two horizontal lines across the top }
```

```
  MoveTo(0, 18);
  LineTo(719, 18);
  MoveTo(0, 20);
  LineTo(719, 20);
```

```
  { draw divider lines }
```

```
  MoveTo(0, 134);
  LineTo(719, 134);
  MoveTo(0, 248);
  LineTo(719, 248);
  MoveTo(240, 21);
  LineTo(240, 363);
  MoveTo(480, 21);
  LineTo(480, 363);
```

```
{ draw title }
TextFont(0);
MoveTo(210,14);
DrawString('Look what you can draw with QuickDraw');
```

```
{----- draw text samples ----- }
```

```
MoveTo(80,34); DrawString('Text');
```

```
TextFace([bold]);
MoveTo(70,55); DrawString('Bold');
```

```
TextFace([italic]);
MoveTo(70,70); DrawString('Italic');
```

```
TextFace([underline]);
MoveTo(70,85); DrawString('Underline');
```

```
TextFace([outline]);
MoveTo(70,100); DrawString('Outline');
```

```
TextFace([shadow]);
MoveTo(70,115); DrawString('Shadow');
```

```
TextFace([]); { restore to normal }
```

```
{ ----- draw line samples ----- }
```

```
MoveTo(330,34); DrawString('Lines');
```

```
MoveTo(280,25); Line(160,40);
```

```
PenSize(3,2);
MoveTo(280,35); Line(160,40);
```

```
PenSize(6,4);
MoveTo(280,46); Line(160,40);
```

```
PenSize(12, 8);
PenPat(gray);
MoveTo(280, 61); Line(160, 40);

PenSize(15, 10);
PenPat(myPattern);
MoveTo(280, 80); Line(160, 40);
PenNormal;

{ ----- draw rectangle samples ----- }

MoveTo(560, 34); DrawString('Rectangles');

SetRect(tempRect, 510, 40, 570, 70);
FrameRect(tempRect);

OffsetRect(tempRect, 25, 15);
PenSize(3, 2);
EraseRect(tempRect);
FrameRect(tempRect);

OffsetRect(tempRect, 25, 15);
PaintRect(tempRect);

OffsetRect(tempRect, 25, 15);
PenNormal;
FillRect(tempRect, gray);
FrameRect(tempRect);

OffsetRect(tempRect, 25, 15);
FillRect(tempRect, myPattern);
FrameRect(tempRect);

{ ----- draw roundRect samples ----- }

MoveTo(70, 148); DrawString('RoundRects');

SetRect(tempRect, 30, 150, 90, 180);
FrameRoundRect(tempRect, 30, 20);
```

```
OffsetRect(tempRect, 25, 15);
PenSize(3, 2);
EraseRoundRect(tempRect, 30, 20);
FrameRoundRect(tempRect, 30, 20);

OffsetRect(tempRect, 25, 15);
PaintRoundRect(tempRect, 30, 20);

OffsetRect(tempRect, 25, 15);
PenNormal;
FillRoundRect(tempRect, 30, 20, gray);
FrameRoundRect(tempRect, 30, 20);

OffsetRect(tempRect, 25, 15);
FillRoundRect(tempRect, 30, 20, myPattern);
FrameRoundRect(tempRect, 30, 20);

{ ----- draw bitmap samples ----- }

MoveTo(320, 148); DrawString('BitMaps');

DrawIcon(0, 266, 156);
DrawIcon(1, 336, 156);
DrawIcon(2, 406, 156);
DrawIcon(3, 266, 196);
DrawIcon(4, 336, 196);
DrawIcon(5, 406, 196);

{ ----- draw ARC samples ----- }

MoveTo(570, 148); DrawString('Arcs');

SetRect(tempRect, 520, 153, 655, 243);
FillArc(tempRect, 135, 65, dkGray);
FillArc(tempRect, 200, 130, myPattern);
FillArc(tempRect, 330, 75, gray);
FrameArc(tempRect, 135, 270);
OffsetRect(tempRect, 20, 0);
PaintArc(tempRect, 45, 90);
```

```
{ ----- draw polygon samples ----- }  
MoveTo(80,262); DrawString('Polygons');  
  
myPoly:=OpenPoly;  
  MoveTo(30,290);  
  LineTo(30,280);  
  LineTo(50,265);  
  LineTo(90,265);  
  LineTo(80,280);  
  LineTo(95,290);  
  LineTo(30,290);  
ClosePoly;      { end of definition }  
  
FramePoly(myPoly);  
  
OffsetPoly(myPoly,25,15);  
PenSize(3,2);  
ErasePoly(myPoly);  
FramePoly(myPoly);  
  
OffsetPoly(myPoly,25,15);  
PaintPoly(myPoly);  
  
OffsetPoly(myPoly,25,15);  
PenNormal;  
FillPoly(myPoly,gray);  
FramePoly(myPoly);  
  
OffsetPoly(myPoly,25,15);  
FillPoly(myPoly,myPattern);  
FramePoly(myPoly);  
  
KillPoly(myPoly);
```

```
{ ----- demonstrate region clipping ----- }
MoveTo(320,262); DrawString('Regions');

myRgn:=NewRgn;
OpenRgn;
  ShowPen;

  SetRect(tempRect,260,270,460,350);
  FrameRoundRect(tempRect,24,16);

  MoveTo(275,335); { define triangular hole }
  LineTo(325,285);
  LineTo(375,335);
  LineTo(275,335);

  SetRect(tempRect,365,277,445,325); { oval hole }
  FrameOval(tempRect);

  HidePen;
  CloseRgn(myRgn);      { end of definition }

SetClip(myRgn);

FOR i:=0 TO 6 DO { draw stuff inside the clip region }
  BEGIN
    MoveTo(260,280+12*i);
    DrawString('Arbitrary Clipping Regions');
  END;

ClipRect(thePort^.portRect);
DisposeRgn(myRgn);

{ ----- draw oval samples ----- }
MoveTo(580,262); DrawString('Ovals');

SetRect(tempRect,510,264,570,294);
FrameOval(tempRect);
```

```
OffsetRect(tempRect, 25, 15);
PenSize(3, 2);
EraseOval(tempRect);
FrameOval(tempRect);

OffsetRect(tempRect, 25, 15);
PaintOval(tempRect);

OffsetRect(tempRect, 25, 15);
PenNormal;
FillOval(tempRect, gray);
FrameOval(tempRect);

OffsetRect(tempRect, 25, 15);
FillOval(tempRect, myPattern);
FrameOval(tempRect);
END; { DrawStuff }

BEGIN { main program }
  InitHeap(@heapBuf, @heapBuf[10000], @HeapFull);
  InitGraf(@thePort);
  InitCursor;
  HideCursor;
  FMInit(errNum);
  OpenPort(@myPort);
  PaintRect(thePort^.portRect);
  InitIcons;
  DrawStuff;
END.
```

E.14 Glossary

bit image: A collection of bits in memory which have a rectilinear representation. The Lisa screen is a visible bit image.

bitMap: A pointer to a bit image, the row width of that image, and its boundary rectangle.

boundary rectangle: A rectangle defined as part of a bitMap, which encloses the active area of the bit image and imposes a coordinate system on it. Its top left corner is always aligned around the first bit in the bit image.

character style: A set of stylistic variations, such as bold, italic, and underline. The empty set indicates normal text (no stylistic variations).

clipping: Limiting drawing to within the bounds of a particular area.

clipping region: Same as clipRgn.

clipRgn: The region to which an application limits drawing in a grafPort.

coordinate plane: A two-dimensional grid. In QuickDraw, the grid coordinates are integers ranging from -32768 to +32767, and all grid lines are infinitely thin.

cursor: A 16-by-16-bit image that appears on the screen and is controlled by the mouse.

cursor level: A value, initialized to 0 when the system is booted, that keeps track of the number of times the cursor has been hidden.

empty: Containing no bits, as a shape defined by only one point.

font: The complete set of characters of one typeface, such as Century.

frame: To draw a shape by drawing an outline of it.

global coordinate system: The coordinate system based on the top left corner of the bit image being at (0,0).

grafPort: A complete drawing environment, including such elements as a bitMap, a subset of it in which to draw, a character font, patterns for drawing and erasing, and other pen characteristics.

grafPtr: A pointer to a grafPort.

handle: A pointer to one master pointer to a dynamic, relocatable data structure (such as a region).

hotSpot: The point in a cursor that is aligned with the mouse position.

kern: To stretch part of a character back under the previous character.

local coordinate system: The coordinate system local to a grafPort, imposed by the boundary rectangle defined in its bitMap.

missing symbol: A character to be drawn in case of a request to draw a character that is missing from a particular font.

pattern: An 8-by-8-bit image, used to define a repeating design (such as stripes) or tone (such as gray).

pattern transfer mode: One of eight transfer modes for drawing lines or shapes with a pattern.

picture: A saved sequence of QuickDraw drawing commands (and, optionally, picture comments) that you can play back later with a single procedure call; also, the image resulting from these commands.

picture comments: Data stored in the definition of a picture which does not affect the picture's appearance but may be used to provide additional information about the picture when it's played back.

picture frame: A rectangle, defined as part of a picture, which surrounds the picture and gives a frame of reference for scaling when the picture is drawn.

pixel: The visual representation of a bit on the screen (white if the bit is 0, black if it's 1).

point: The intersection of a horizontal grid line and a vertical grid line on the coordinate plane, defined by a horizontal and a vertical coordinate.

polygon: A sequence of connected lines, defined by QuickDraw line-drawing commands.

port: Same as grafPort.

portBits: The bitMap of a grafPort.

portBits.bounds: The boundary rectangle of a grafPort's bitMap.

portRect: A rectangle, defined as part of a grafPort, which encloses a subset of the bitMap for use by the grafPort.

region: An arbitrary area or set of areas on the coordinate plane. The outline of a region should be one or more closed loops.

row width: The number of bytes in each row of a bit image.

solid: Filled in with any pattern.

source transfer mode: One of eight transfer modes for drawing text or transferring any bit image between two bitMaps.

style: See character style.

thePort: A global variable that points to the current grafPort.

transfer mode: A specification of which boolean operation QuickDraw should perform when drawing or when transferring a bet image from one bitMap to another.

visRgn: The region of a grafPort which is actually visible on the screen.

Appendix F HARDWARE INTERFACE

F.1	The Mouse	F-3
F.1.1	Mouse Location	F-3
F.1.2	Mouse Update Frequency	F-3
F.1.3	Mouse Scaling	F-3
F.1.4	Mouse Odometer	F-4
F.2	The Cursor	F-4
F.2.1	Cursor/Mouse Tracking	F-5
F.2.2	The Busy Cursor	F-5
F.3	The Display Screen	F-6
F.3.1	Screen Contrast	F-6
F.3.2	Automatic Screen Fading	F-6
F.4	The Speaker	F-7
F.5	The Keyboard	F-7
F.5.1	Keyboard Identification	F-9
F.5.2	Keyboard State	F-10
F.5.3	Keyboard Events	F-10
F.5.4	Dead Key Diacriticals	F-12
F.5.5	Repeats	F-12
F.6	The Microsecond Timer	F-13
F.7	The Millisecond Timer	F-13
F.8	Date and Time	F-14
F.9	Time Stamp	F-14
F.10	Summary of the Hardware Interface	F-15

HARDWARE INTERFACE

The hardware interface software provides an interface for accessing and controlling several parts of the Lisa hardware. The hardware/software capabilities addressed include the mouse, the cursor, the display, the contrast control, the speaker, both undecoded and decoded keyboard access, the microsecond and millisecond timers and the hardware clock/calendar.

The documentation below contains interleaved text descriptions and Pascal function and procedure declarations. Pascal type declarations and a summary of the function and procedure declarations can be found in Section F.10, Summary of the Hardware Interface.

F.1 The Mouse

F.1.1 Mouse Location

The mouse is a pointing device used to indicate screen locations. Procedure `MouseLocation` returns the location of the mouse. The X-coordinate can range from 0 to 719, and the Y-coordinate from 0 to 363. The initial mouse location is 0,0.

```
procedure MouseLocation (var x: Pixels; var y: Pixels);
```

F.1.2 Mouse Update Frequency

Software knowledge of the mouse location is updated periodically, rather than continuously. The frequency of these updates can be set by calling procedure `MouseUpdates`. The time between updates can range from 0 milliseconds (continuous updating) to 28 milliseconds, in intervals of 4 milliseconds. The initial setting is 16 milliseconds.

```
procedure MouseUpdates (delay: MilliSeconds);
```

F.1.3 Mouse Scaling

The relationship between physical mouse movements and logical mouse movements is not necessarily a fixed linear mapping. Three alternatives are available: 1) unscaled, 2) scaled for fine movement and 3) scaled for coarse movement. Initially mouse movements are unscaled.

When mouse movement is unscaled, a horizontal mouse movement of x units yields a change in the mouse X-coordinate of x pixels. Similarly, a vertical movement of y units yields a change in the mouse Y-coordinate of y pixels. These rules apply independent of the speed of the mouse movement.

When mouse movement is scaled, horizontal movements are magnified by $3/2$ relative to vertical movements. This is intended to compensate for the $2/3$ aspect ratio of pixels on the screen. When scaling is in effect, a distinction is made between fine (small) movements and coarse (large) movements. Fine movements are slightly reduced, while coarse movements are magnified. For

scaled fine movements, a horizontal mouse movement of x units yields a change in the X-coordinate of x pixels, but a vertical movement of y units yields a change of $(2/3)*y$ pixels. For scaled coarse movements, a horizontal movement a x units yields a change of $(3/2)*x$ pixels, while a vertical movements of y units yields a change of y pixels.

The distinction between fine movements and coarse movements is determined by the sum of the x and y movements each time the mouse location is updated. If this sum is at or below the 'threshold', the movement is considered to be a fine movement. Values of the threshold range from 0 (which yields all coarse movements) to 256 (which yields all fine movements). Given the default mouse updating frequency, a threshold of about 8 (threshold's initial setting) gives a comfortable transition between fine and coarse movements.

Procedure `MouseScaling` enables and disables mouse scaling. Procedure `MouseThresh` sets the threshold between fine and coarse movements.

```
procedure MouseScaling (scale:Boolean);
procedure MouseThresh (threshold: Pixels);
```

F.1.4 Mouse Odometer

In order to properly specify, design and test mice, it's important to estimate how far a mouse moves during its lifetime. Function `MouseOdometer` returns the sum of the X and Y movements of the mouse since boot time. The value returned is in (unscaled) pixels. There are 180 pixels per inch of mouse movement.

```
function MouseOdometer: ManyPixels;
```

F.2 The Cursor

The cursor is a small image that is displayed on the screen. Its shape is specified by two bitmaps, called 'data' and 'mask'. These bitmaps are 16 bits wide and from 0 to 32 bits high. The rule used to combine the bits already on the screen with the data and mask is

```
screen <- (screen and (not mask)) xor data.
```

The effect is that white areas of the screen are replaced with the cursor data. Black areas of the screen are replaced with (not mask) xor data. If the data and mask bitmaps are identical, the effect is to 'or' the data onto the screen.

The cursor has both a location and a hotspot. The location is a position on the screen, with X-coordinates of 0 to 719 and Y-coordinates of 0 to 363. The hotspot is a position within the cursor bitmaps, with X- and Y-coordinates ranging from 0 to 16. The cursor is displayed on the screen with its hotspot at its location. If the cursor's location is near an edge of the screen, the cursor image may be partially or completely off the screen.

Most cursor operations can be performed by calling procedures `SetCursor`, `HideCursor`, `ShowCursor`, and `ObscureCursor` defined by `QuickDraw` (see Section

E.9.2, Cursor-Handling Routines). Additional capabilities are provided by the Hardware Interface routines described below.

Procedure `CursorImage` is used to specify the data bitmap, mask bitmap, height and hotspot of the cursor. Initially the cursor data and mask bitmaps contain all zeros, which yields a blank (invisible) cursor. The initial hotspot is 0,0.

```
procedure CursorImage (hotX: Pixels; hotY: Pixels; height:
  CursorHeight; proceduredata: CursorPtr; mask: CursorPtr);
```

F.2.1 Cursor/Mouse Tracking

Procedure `CursorTracking` enables and disables cursor tracking of the mouse. When tracking is enabled the cursor location is changed to the mouse location each time the mouse moves. Setting the cursor location will have no effect, the cursor sticks with the mouse.

When tracking is disabled, the mouse location and cursor location are independent. Setting the cursor location will move the cursor; moving the mouse will not.

When tracking is first enabled (i.e. on each transition from disabled to enabled) the mouse location is modified to equal the cursor location. Therefore, enabling tracking does not move the cursor; it does modify the mouse location. Initially tracking is enabled.

```
procedure CursorTracking (track: Boolean);
```

F.2.2 The Busy Cursor

Applications may desire to display a busy cursor (e.g. an hourglass) when an operation in progress requires more than a few seconds to complete. Procedure `BusyImage` is used to specify the data bitmap, mask bitmap, height and hotspot of the busy cursor.

A call to procedure `BusyDelay` specifies that the normal cursor should currently be displayed, and that display of the busy cursor should be delayed for the specified number of milliseconds. Subsequent calls to `BusyDelay` override previous calls, postponing display of the busy cursor. If no calls to `BusyDelay` occur for the specified number of milliseconds, the busy cursor will be displayed until the next call to `BusyCursor`.

Initially the busy cursor data and mask bitmaps contain all zeros, which yields a blank (invisible) cursor. The initial hotspot is 0,0. The initial busy delay is infinite, that is, the busy cursor will not be displayed until `BusyDelay` is called.

```
procedure BusyImage (hotX: Pixels; hotY: Pixels; height: CursorHeight;
  data: CursorPtr; mask: CursorPtr);
```

```
procedure BusyDelay (delay: Milliseconds);
```

F.3 The Display Screen

The display screen is a bit mapped display; that is, each pixel on the screen is controlled by a bit in main memory. The display has 720 pixels horizontally and 364 lines vertically, and therefore requires 32,760 bytes of main memory. The screen size may be determined by calling procedure `ScreenSize`.

```
procedure ScreenSize (var x: Pixels; var y: Pixels);
```

The screen is redisplayed about 60 times per second. A frame counter is incremented between screen updates, at the vertical retrace interrupt. The frame counter is an unsigned 32-bit integer which is reset to 0 each time the machine is booted. Function `FrameCounter` returns this value. An application can synchronize with the vertical retraces by watching for changes in the value of this counter. The frame counter should not be used as a timer, use the millisecond and microsecond timers instead.

```
function FrameCounter: Frames;
```

F.3.1 Screen Contrast

The display's contrast level is under program control. Contrast values range from 0 to 255 (\$FF), with 0 as maximum contrast and 255 as minimum. Function `Contrast` returns the contrast setting; procedure `SetContrast` sets the screen contrast. The low order two bits of the contrast value are ignored. The initial contrast value is 128 (\$80).

```
function Contrast: ScreenContrast;
```

```
procedure SetContrast (contrast: ScreenContrast);
```

A sudden change in the contrast level can be jarring to the user. Procedure `RampContrast` gradually changes the contrast to the new setting over a period of approximately a second. `RampContrast` returns immediately, then ramps the contrast using interrupt driven processing.

```
procedure RampContrast (contrast: ScreenContrast);
```

F.3.2 Automatic Screen Fading

The screen contrast level is automatically dimmed if no user activity is noted over a specified period (usually several minutes). This is done in order to preserve the screen phosphor. Function `DimContrast` returns the contrast value to which the screen is dimmed; procedure `SetDimContrast` sets this value. The initial dim contrast setting is 176 (\$B0).

```
function DimContrast: ScreenContrast;
```

```
procedure SetDimContrast (contrast: ScreenContrast);
```

The delay between the last user activity and dimming of the screen is under software control. Function `FadeDelay` returns the fade delay; procedure `SetFadeDelay` sets it. The actual delay will range from the specified delay to twice the specified delay. The initial delay period is five minutes.

When the screen is dimmed, user interaction will cause the screen contrast to return to its normal bright level (determined by routines Contrast and SetContrast defined above). Moving the mouse or pressing a key on the keyboard (e.g. SHIFT) is enough to trigger the screen brightening. Calling procedures CursorLocation and/or SetFadeDelay also indicates user activity.

```
function FadeDelay: MilliSeconds;
procedure SetFadeDelay (delay: MilliSeconds);
```

F.4 The Speaker

The routines below provide square wave output from the Lisa speaker. The speaker volume can be set to values in the range 0 (soft) to 7 (loud). Function Volume reads the volume setting; procedure SetVolume sets it. The initial volume setting is 4.

Procedure Noise produces a square wave of approximately the specified wavelength. Procedure Silence shuts off the square wave. The minimum wavelength is about 8 microseconds, which corresponds to a frequency of 125,000 cycles per second, well above the audible range. The maximum wavelength is 8,191 microseconds, which corresponds to about 122 cycles per second.

Procedures Noise and Silence are called in pairs to start and stop square wave output. In contrast, procedure Beep starts square wave output which will automatically stop after the specified period of time. The effects of Noise, Silence and Beep are overridden by subsequent calls.

```
function Volume: SpeakerVolume;
procedure SetVolume (volume: SpeakerVolume);
procedure Noise (waveLength: MicroSeconds);
procedure Silence;
procedure Beep (waveLength: MicroSeconds; duration: MilliSeconds);
```

F.5 The Keyboard

The routines below provide an interface to the keyboard, the keypad, the mouse button and plug, the diskette buttons and insertion switches, and the power switch. Two interfaces are provided, a pollable keyboard state and a queue of keyboard events.

Three physical keyboard layouts are defined, the "Old US Layout" (with 73 keys on the main keyboard and numeric keypad), the "Final US Layout" (76 keys) and the "European Layout" (77 keys). Each key has been assigned a keycode, which uniquely identifies the key. Keycode values range from 0 to 127. The matrix below defines the keycodes for the "Final US Layout", using the legends from the US Keyboard. The "Old US Layout" has three less keys. \backslash , Alpha Enter, and Right Option are not on the old keyboard. The "European Layout" has one additional key, $\><$, with a keycode of \$43.

Two keys on the "Old US Layout" generate keycodes different from the corresponding keys on the "Final US Layout". To aid in compatibility, software changes the keycode for ~ from \$7C to \$68, and the keycode for Right Option from \$68 to \$4E.

high	000	001	010	011	100	101	110	111
	0	1	2	3	4	5	6	7
low								
0000	0		Clear		_ -	(9	E	A
0001	1	Disk 1 Inserted	-		+ =) 0	^ 6	@ 2
0010	2	Disk 1 Button	+ Left		\	U	& 7	# 3
0011	3	Disk 2 Inserted	* Right			I	* 8	\$ 4
0100	4	Disk 2 Button	7		P	J	% 5	! 1
0101	5	Parallel Port	8		Backsp	K	R	Q
0110	6	Mouse Button	9		Alpha Enter	{ [T	S
0111	7	Mouse Plug	/ Up			}]	Y	W
1000	8	Power Button	4		Return	M	~ `	Tab
1001	9		5		0	L	F	Z
1010	A		6			: ;	G	X
1011	B		, Down			" '	H	D
1100	C		.		? /	Space	V	Left Option
1101	D		2		1	< ,	C	Caps Lock
1110	E		3		Right Option	> .	B	Shift
1111	F		Enter			0	N	⌘

F.5.1 Keyboard Identification

The Lisa supports a host of different keyboards. Each keyboard has three major attributes: manufacturer, physical layout, and keycap legends. The chart below describes how these three attributes are combined to form a keyboard identification number. The keyboards self identify when the machine is turned on and when a new keyboard is attached. Function Keyboard returns the identification number of the keyboard currently attached. Function Legends and procedure SetLegends provide a means of pretending to have different legends, without physically replacing the keyboard.

Keyboard identification numbers:

7	6	5	4	3	2	1	0
manufacturer	Layout	Keycap Legends					

Manufacturer:

00 -- APD (i.e. TKC)
 01 --
 10 -- Keytronics

Layout:

00 -- Old US (73 keys)
 01 --
 10 -- European (77 keys)
 11 -- Final US (76 keys)

Layout/Legends:

0F -- Old US
 26 -- Swiss-German (proposed)
 27 -- Swiss-French (proposed)
 29 -- Portuguese (proposed)
 29 -- Spanish (proposed)
 2A -- Danish (proposed)
 2B -- Swedish
 2C -- Italian
 2D -- French
 2E -- German
 2F -- UK

```

3C -- APL          (proposed)
3D -- Canadian    (proposed)
3E -- US-Dvorak
3F -- Final US

```

```
function Keyboard: KeybdId;
```

```
function Legends: KeybdId;
```

```
procedure SetLegends (id: KeybdId);
```

F.5.2 Keyboard State

Low level access to the keyboard is provided through a pollable keyboard state. This state information is based on the physical keycodes defined above. Function `KeyIsDown` returns the position of a single specified key. Procedure `KeyMap` returns an 128-bit map, one bit for each key. A zero indicates the key is up, a one indicates down. For the mouse plug, a zero indicates unplugged, a one indicates plugged in. Certain keys are not pollable; the corresponding bits will always be zero. These keys are the diskette insertion switches and buttons, parallel port, and power switch. The parallel port and mouse plug keys are unreliable across reboots on older hardware.

```
function KeyIsDown (key: KeyCap): Boolean;
```

```
procedure KeyMap (var keys: KeyCapSet);
```

F.5.3 Keyboard Events

The hardware interface provides a queue of keyboard events. The events in the input queue are generally key down transitions. Each event contains the following information:

```

keycode  -- physical key
ascii    -- ASCII interpretation of this key
state    -- caps-lock, shift, option, ⌘, mouse button and repeat
mouseX   -- X-coordinate of the mouse when the key was pressed
mouseY   -- Y-coordinate of the mouse when the key was pressed
time     -- value of the millisecond timer when the key was pressed

```

Keycode -- Keycodes are defined in the chart above.

Ascii -- The ASCII interpretation of keys depends on the state of the caps-lock, shift and option keys. Six interpretations are associated with each different keyboard layout:

```

normal
caps-lock
shift or (shift and caps-lock)

alternate
alternate with caps-lock
alternate with shift or (shift and caps-lock)

```

In most cases the ASCII value returned is obvious. The table below lists the cases that aren't so obvious.

\$00 (NUL)	Disk 1 Inserted
\$00 (NUL)	Disk 1 Button
\$00 (NUL)	Disk 2 Inserted
\$00 (NUL)	Disk 1 Button
\$00 (NUL)	Power Button
\$00 (NUL)	Mouse Button (down)
\$00 (NUL)	Mouse Plug (in)
\$01 (SOH)	Mouse Button (up)
\$01 (SOH)	Mouse Plug (out)
\$03 (ETX)	Enter
\$08 (BS)	BackSpace
\$09 (HT)	Tab
\$0D (CR)	Return
\$1B (ESC)	Clear
\$1C (FS)	Left
\$1D (GS)	Right
\$1E (RS)	Up
\$1F (US)	Down
\$20 (SP)	Space

State -- A 16-bit word is used to return the state of several keys with each event. Each bit represents one or more keys, a zero indicates that all of the keys are up, a one indicates that at least one of the keys is down. An additional bit indicates, if it is a one, that the event was generated by repeating the previous event. The following bits of state are currently assigned:

- bit 0: caps-lock
- bit 1: left or right shift
- bit 2: left or right option
- bit 3: ⌘
- bit 4: mouse button
- bit 5: this event is a repeat

Certain keys never generate events. These keys are caps-lock, both shift keys, option keys, and the ⌘ key. The mouse button generates events on both the down and up transitions. Down transitions have an ASCII value of 0, up transitions 1. The mouse plug also generates two different events. When the mouse is plugged in an event with an ASCII value of 0 is returned, when it is unplugged a value of 1 is returned.

Function `KeybdPeek` is used to examine events in the keyboard queue, without removing them from the queue. The first input parameter indicates whether repeats are desired. The second parameter is the queue index. The first output parameter indicates whether the specified queue entry contains an event. To

examine an entire queue, first call `KeybdPeek` with a queue index of 1. If an event is returned, call it again with a queue index of 2, etc.

Function `KeybdEvent` is used both to determine if a keyboard event is available, and to return the event if one is available. The event is removed from the queue. `KeybdEvent` returns a boolean result which is true if an event is returned. The first parameter to `KeybdEvent` is used to indicate if the caller will accept repeated events on this call. The second parameter indicates if the functions should wait for an event if one is not immediately available.

```
function KeybdPeek (repeats: Boolean; index: KeybdQIndex; var
    event: KeyEvent): Boolean;

function KeybdEvent (repeats: Boolean; wait: Boolean; var event:
    KeyEvent): Boolean;
```

F.5.4 Dead Key Diacriticals

Many languages employ diacritical marks on certain letters. Several of the required diacritical mark-letter combinations appear on European keyboards, but others do not. The combinations shown in the table below may be typed as a two-key sequence, by first typing the dead key diacritical (which has no immediate effect), and then typing the letter. Dead key diacriticals appear on keyboard legends as the diacritical mark over a dotted square.

circumflex	^	--	â	ê	î	ô	û
grave accent	`	--	à	è	ì	ò	ù
tilde	~	--	ã		ñ	õ	
acute accent	´	--	á	é	í	ó	ú
umlaut	¨	--	ä	ë	ï	ö	ü


A dead key diacritical followed by a letter which appears in the table above yields the corresponding character. The event that is generated contains the keycode, state, mouse location and time that correspond to the letter, but the ASCII value of the letter-diacritical combination. A dead key diacritical followed by a space yields just the diacritical mark. The event contains the keycode, state, mouse location and time corresponding to the space, but the ASCII value of the diacritical mark. Finally, a dead key diacritical followed by any other character (i.e. not a space or defined letter) yields the diacritical mark followed by the other character.

diacritical, defined letter	-->	foreign character
diacritical, space	-->	diacritical
diacritical, other character	-->	diacritical, other character

F.5.5 Repeats

Most keys, if held down for an extended period of time, may generate multiple events (repeats). Several conditions must all be satisfied before a repeat is generated. These conditions are as follows:

1. KeybdPeek or KeybdEvent is called with repeatsDesired true.
2. The keyboard event queue is empty.
3. The key returned in the last event is still down.
4. No down transitions have occurred since the last event.
5. The key is repeatable.
6. Enough time has elapsed.

All keys are repeatable, with the exception of caps-lock, shifts, options, , disk 1 inserted, disk 1 button, disk 2 inserted, disk 2 button, parallel port, mouse button, mouse plug, and the power button.

Repeats generate events with the following attributes:

keycode	-- original keycode
ascii	-- original ASCII interpretation
state	-- original position of the caps-lock, shift, etc.
mouseX	-- revised X-coordinate of the mouse
mouseY	-- revised Y-coordinate of the mouse
time	-- revised value of the millisecond timer

The repeat rates can be read and set by calls to RepeatRate and SetRepeatRate. The rates include an initial delay, which occurs prior to the first repetition, and a subsequent delay, prior to additional repetitions. They are both in units of milliseconds. The default repeat rates are 400 milliseconds initially and 100 milliseconds subsequently.

```
procedure RepeatRate (var initial: MilliSeconds; var subsequent:
  MilliSeconds);
```

```
procedure SetRepeatRate (Initial: MilliSeconds; subsequent:
  MilliSeconds);
```

F.6 The Microsecond Timer

Function MicroTimer simulates a continuously running 32-bit counter which is incremented every microsecond. The timer is reset to 0 each time the machine is booted. The timer changes sign about once every 35 minutes, and rolls over every 70 minutes.

The microsecond timer is designed for performance measurements. It has a resolution of 2 microseconds. Calling MicroTimer from Pascal takes about 135 microseconds. Note that interrupt processing will have a major effect on microsecond timings.

```
function MicroTimer: Microseconds;
```

F.7 The Millisecond Timer

Function Timer simulates a continuously running 32-bit counter which is incremented every millisecond. The timer is reset to 0 each time the machine is booted. The timer changes sign about once every 25 days, and rolls over every 7 weeks.

The millisecond timer is designed for timing user interactions such as mouse clicks and repeat keys. It can also be used for performance measurements, assuming that millisecond resolution is sufficient.

```
function Timer: Milliseconds;
```

F.8 Date and Time

The current date and time is available as a set of 16-bit integers which represent the year, day, hour, minute and second, by calling procedures `DateTime` and `SetDateTime`. The date and time are based on the hardware clock/calendar. This restricts dates to the years 1980-1995. The clock/calendar continues to operate during soft power off, and for brief periods on battery backup if the machine is unplugged. If the clock/calendar hasn't been set since the last loss of battery power, the date and time will be midnight prior to January 1, 1980. Setting the date and time also sets the time stamp described below. Procedure `DateToTime` converts a date and time to a time stamp, defined in the next section.

```
procedure DateTime (var date: DateArray);
```

```
procedure SetDateTime (date: DateArray);
```

```
procedure DateToTime (date: DateArray; var time: Seconds);
```

F.9 Time Stamp

The current date and time is also available as a 32-bit unsigned integer which represents the number of seconds since the midnight prior to 1 January 1901, by calling function `TimeStamp` and procedure `SetTimeStamp`. The time stamp will roll over once every 135 years. Beware - for dates beyond the mid 1960's the sign bit is set. The time stamp is based on the hardware clock/calendar. This clock continues to operate during soft power off, and for brief periods on battery backup if the machine is unplugged. If the clock/calendar hasn't been set since the last loss of battery power, the date and time will be midnight prior to January 1, 1980. Setting the time stamp also sets the date and time described above. Since the date and time is restricted to 1980-1995, the time stamp is also restricted to this range. Procedure `TimeToDate` converts a time stamp to the date and time format defined above.

```
function TimeStamp: Seconds;
```

```
procedure SetTimeStamp (time: Seconds);
```

```
procedure TimeToDate (time: Seconds; var date: DateArray);
```

F.10 Summary of the Hardware Interface

Unit Hardware;

Interface

type

```

Pixels          = Integer;
ManyPixels     = LongInt;
CursorHeight   = Integer;
CursorPtr      = ^Integer;
DateArray      = Record
    year: Integer;
    day: Integer;
    hour: Integer;
    minute: Integer;
    second: Integer;
end;

Frames         = LongInt;
Seconds        = LongInt;
MilliSeconds   = LongInt;
MicroSeconds   = LongInt;
SpeakerVolume  = Integer;
ScreenContrast = Integer;
KeybdQIndex    = 1..1000;
KeybdId        = Integer;
KeyCap         = 0..127;
KeyCapSet      = Set of KeyCap;

KeyEvent       = Packed Record
    key: KeyCap;
    ascii: Char;
    state: Integer;
    mouseX: Pixels;
    mouseY: Pixels;
    time: MilliSeconds;
end;

```

{ Mouse }

```

procedure MouseLocation (var x: Pixels; var y: Pixels);
procedure MouseUpdates (delay: MilliSeconds);
procedure MouseScaling (scale: Boolean);
procedure MouseThresh (threshold: Pixels);
function MouseOdometer: ManyPixels;

```


{ Cursor }

```
procedure CursorLocation (x: Pixels; y: Pixels);
procedure CursorTracking (track: Boolean);
procedure CursorImage (hotX: Pixels; hotY: Pixels; height: CursorHeight;
  data: CursorPtr; mask: CursorPtr);

procedure BusyImage (hotX: Pixels; hotY: Pixels; height: CursorHeight; data:
  CursorPtr; mask: CursorPtr);
procedure BusyDelay (delay: Milliseconds);
```

{ Screen }

```
function FrameCounter: Frames;
procedure ScreenSize (var x: Pixels; var y: Pixels);

function Contrast: ScreenContrast;
procedure SetContrast (contrast: ScreenContrast);
procedure RampContrast (contrast: ScreenContrast);
function DimContrast: ScreenContrast;
procedure SetDimContrast (contrast: ScreenContrast);

function FadeDelay: Milliseconds;
procedure SetFadeDelay (delay: Milliseconds);
```

{ Speaker }

```
function Volume: SpeakerVolume;
procedure SetVolume (volume: SpeakerVolume);
procedure Noise (waveLength: MicroSeconds);
procedure Silence;
procedure Beep (waveLength: MicroSeconds; duration: Milliseconds);
```

{ Keyboard }

```
function Keyboard: KeybdId;
function Legends: KeybdId;
procedure SetLegends (id: KeybdId);
function KeyIsDown (key: KeyCap): Boolean;
procedure KeyMap (var keys: KeyCapSet);
function KeybdPeek (repeats: Boolean; index: KeybdQIndex; var event:
    KeyEvent): Boolean;
function KeybdEvent (repeats: Boolean; wait: Boolean; var event:
    KeyEvent): Boolean;
procedure RepeatRate (var initial: Milliseconds; var subsequent:
    Milliseconds);
procedure SetRepeatRate (initial: Milliseconds; subsequent: Milliseconds);
```

{ Timers }

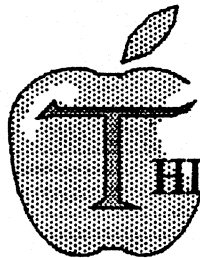
```
function MicroTimer: MicroSeconds;
function Timer: Milliseconds;
```

{ Date and Time }

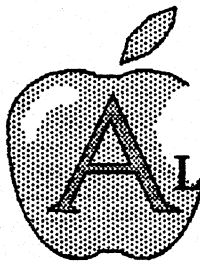
```
procedure DateTime (var date: DateArray);
procedure SetDateTime (date: DateArray);
procedure DateToTime (date: DateArray; var time: Seconds);
```

{ Time Stamp }

```
function TimeStamp: Seconds;
procedure SetTimeStamp (time: Seconds);
procedure TimeToDate (time: Seconds; var date: DateArray);
```

HIS MANUAL was produced using
LisaWrite, LisaDraw, and
LisaList.



ALL PRINTING was done with an
Apple Dot-Matrix Printer.

 Lisa™

...we use it ourselves.



..... FOLD

..... FOLD

FROM:

PLACE
STAMP
HERE



APPLE COMPUTER INC.
POS Publications Department
20525 Mariani Avenue, MS 2-0
Cupertino, California 95014

TAPE OR STAPLE

Apple publications would like to learn about readers and what you think about this manual in order to make better manuals in the future. Please fill out this form, or write all over it, and send it to us. We promise to read it.

Is it quick and easy to find the information you need in this manual?

always often sometimes seldom never

Comments _____

What made this manual easy to use? _____

What made this manual hard to use? _____

How are you using this manual?

learning to use the product reference both reference and learning

other _____

Please comment on, for example, accuracy, level of detail, number and usefulness of examples, length or brevity of explanation, style, use of graphics, usefulness of the index, organization, suitability to your particular needs, readability.

What do you like most about the manual? _____

What do you like least about the manual? _____

In school have you completed?

high school some college BA/BS MA/MS more

Comments _____

What is your job title? _____

How long have you been programming?

0-1 years 1-3 4-7 over 7 not a programmer

Comments _____

What languages do you use on your Lisa? (check each)

Pascal BASIC COBOL other _____

Comments _____

What magazines do you read? _____