OS-9/68000
PASCAL LANGUAGE
USER'S MANUAL

The information contained herein is believed to be accurate as of the date of publication, however, Microware will not be liable for any damages, including indirect or consequential, from use of the OS-9 operating system or reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

Revision A
Publication date: July, 1985

Microware Systems Corporation
1866 N.W. 114th Street
Des Moines, Iowa 50322
(515) - 224-1929

PMN PCL68

OS-9/68000 PASCAL LANGUAGE USER'S MANUAL
TABLE OF CONTENTS

Chapter 1 - INTRODUCTION

About OS-9 Pascal 1-1
OS-9/68000 Pascal Features 1-1
Do You Know Pascal 1-3
Suggested ISO Pascal Source Books 1-4

Chapter 2 - INSTALLING AND RUNNING THE COMPILER

Requirements and Information 2-1
Installing the parts of the Pascal System 2-1
Description of OS-9/68000 Pascal Files 2-3
File Name Suffix Conventions 2-5
Command Lines and the Pascal Executive 2-5
Phase One 2-5
Phase Two 2-6
Phase Three 2-6
Phase Four 2-6
Executive Options 2-7
"-B<number>" 2-7
"-D<number>" 2-7
"-D" 2-7
"-E=<number>" 2-8
"-F" 2-8
"-G" 2-8
"-I" 2-8
"-K" 2-8
"-L" 2-8
"-L=<path>" 2-9
"-M=<number> 2-9
"-N=<name>" 2-8
"-N" 2-9
"-Q" 2-9
"-S" 2-9
"-S<number>" 2-9
"-T" 2-9
"-W<number>" 2-10
Temporary Files 2-11
Example Command Lines 2-11
Steps in Testing a Pascal Program 2-11
The Pascal Program Source File 2-12
Compiler Directives 2-13
Listing Directives 2-13
$TITLE 2-13
$SUBTITLE 2-13
$PAGE 2-13
$INCLUDE 2-13

Chapter 2 - INSTALLING AND RUNNING THE COMPILER (continued)

Compilation Mode Directives 2-14
$SET 2-14
L+, L- 2-14
D+, D- 2-14
S+, S- 2-15
X+, X- 2-15
Default Mode Values 2-16
$PUSH 2-16
$POP 2-16

Example use of Compilation Mode Directives 2-16
A Sample Compilation Listing 2-17
Sample Source Program 2-17
Sample Compilation Listing 2-18
Listing Page Headings 2-20
Error Messages in Program Listings 2-22
The Procedure Statistics Table 2-23
Separate Compilation 2-24
Example Use of Seperate Compilation 2-24

Chapter 3 - A PASCAL LANGUAGE SUMMARY

Synopsis.....3-1
Alternate Character Sets 3-1
Notation Used in Descriptions 3-2
Source Program Format 3-2
Compilation Mode 3-2
Identifier Names 3-2
General Program Organization 3-3
The Program Statement 3-3
Comments and Blank Statements 3-4
Indentation of Lexical Levels 3-4
Label Declarations and GOTO Statements 3-4
Constant Declarations and Constants 3-5
Numeric Constants 3-6
String and Character Constants 3-7
Type and Variable Declarations 3-7
Hierarchy of OS-9/68000 Pascal Data Types .. 3-8
Storage Allocation 3-8
Arrays 3-9
Sets 3-9
Packed Structures 3-10
File Types 3-10
Procedure and Function Declarations 3-10
Assignment Statements and Expressions 3-13
Pascal Operators 3-14
Compound Statements 3-16
Looping and Conditional Statements 3-16
The IF-THEN-ELSE Statement 3-16
The CASE Statement 3-17

Chapter 3 - A PASCAL LANGUAGE SUMMARY (continued)

The REPEAT Statement 3-19
The WHILE-DO Statement 3-19
The FOR Statement 3-20
The WITH-DO Statement 3-21

Chapter 4 - PASCAL PROGRAM EXECUTION

Executing a Pascal Program 4-1
SYSPARAM and SYSPARAMSIZE 4-1
Abortive Errors 4-2
OS9ERR 4-3

Chapter 5 - PASCAL INPUT/OUTPUT ROUTINES

Predefined Standard I/O Files 5-2
Interactive Files 5-2
TEXT File Declaration 5-3
Window Variables 5-4
Random Access 5-5
STANDARD ROUTINES
APPEND 5-7
CLOSE 5-9
EOF 5-10
EOLN 5-11
FILESIZE 5-12
GET 5-13
GETCHAR 5-14
GETINFO 5-16
INTERACTIVE 5-18
IOABORT 5-19
IOREADY 5-21
IORESULT 5-23
IOSYSERR 5-25
LINELENGTH 5-27
OPENED 5-30
OVERPRINT 5-31
PAGE 5-32
POSITION 5-33
PUT 5-35
PUTINFO 5-36
READ 5-37
READLN 5-41
REPOSITION 5-42
RESET 5-43
REWRITE 5-44
SEEKEOF 5-46
SHORTIO 5-47
SYSREPORT 5-50
UPDATE 5-52

Chapter 5 - PASCAL INPUT/OUTPUT ROUTINES (continued)

WRITE 5-54
WRITEEOF 5-47
WRITELN 5-61

Chapter 6 - PASCAL STRING AND ARRAY ROUTINES

String Declaration 6-1
String Assignment Conformability 6-1
Assigning a String to an Array 6-2
Assigning an Array to a String 6-2
Assigning a Character to a String 6-2
The Null String 6-3
Typeless Parameters 6-4
Vector Slice 6-5

STANDARD ROUTINES
CAP 6-6
CONCAT 6-7
COPY 6-8
DELETE 6-9
FILLCHAR 6-10
INSERT 6-11
LENGTH 6-12
MOVELEFT 6-13
MOVERIGHT 6-14
POS 6-15
SCAN 6-16
STR 6-17
STRINGCMP 6-18

Chapter 7 - PASCAL ARITHMETIC ROUTINES

MATHABORT and MATHRESULT 7-1
REALPRECISION 7-2

STANDARD ROUTINES
ABS 7-3
ARCCOS 7-4
ARCSIN 7-5
ARCTAN 7-6
CHR 7-7
CONVERT 7-8
COS 7-10
EXP 7-11
FIELDGET 7-12
FIELDPUT 7-14
FLOAT 7-17
INT 7-18
LN 7-19
LOG 7-20

Chapter 7 - PASCAL ARITHMETIC ROUTINES (continued)

LONG 7-21
LROUND 7-22
LTRUNC 7-23
ODD 7-24
ORD 7-25
PRED 7-26
RANDOM 7-27
ROUND 7-29
SIN 7-30
SQR 7-31
SQRT 7-32
SUCC 7-33
TAN 7-34
TRUNC 7-35

Chapter 8 - PASCAL HEAP ROUTINES

HEAPREQUEST AND HEAP STRUCTURE 8-1

STANDARD ROUTINES
DISPOSE 8-2
MARK 8-3
NEW 8-4
PROGINFO 8-6
RELEASE 8-8
VARNEW 8-9
VARDISPOSE 8-11

Chapter 9 - PASCAL MISCELLANEOUS ROUTINES

STANDARD ROUTINES
ADDRESS 9-1
EXIT 9-2
HALT 9-4
OS9 9-5
PACK 9-8
SIZEOF 9-9
SYSTIME 9-10
UNPACK 9-12

Appendix A - ERROR MESSAGE DESCRIPTIONS A-1

Appendix B - RUN-TIME ERROR MESSAGE DESCRIPTIONS B-1

Appendix C - PASCAL SYNTAX C-1

Appendix D - RESERVED WORDS AND PREDECLARED INDENTIFIERS ... D-1

OS-9/68000 PASCAL LANGUAGE USER'S MANUAL
CHAPTER 1
INTRODUCTION

ABOUT OS-9/68000 PASCAL

In the last few years, the Pascal language has become one of the most popular computer programming languages. Although it was originally developed as an aid in teaching computer science, it has found wide use in almost every imaginable computer application for good reason. The Pascal language gives programmers an almost perfect medium for concise expression of solutions to complex problems.

The internal operation of a language as powerful as Pascal is relatively complex, and running Pascal programs can be quite demanding of the computer. Therefore, microcomputer versions of Pascal traditionally have been quite limited and much slower than their big computer cousins. The gap has been narrowed considerably in OS-9/68000 Pascal because of the 68000 microprocessor, which was specifically designed to efficiently execute high-level languages such as Pascal. OS-9/68000 Pascal is a full implementation of the language according to the ISO Standard 7185.1 Level 0, also if the "-i" ISO only compile option isn't enabled then all of the extensions which are documented in this manual are allowed. Here is a brief summary of some of OS-9/68000 features.

OS-9/68000 Pascal Features

1. Completely 100% ISO7185.1 compatable with "-i" compile option set.

2. Seperate compilation with "-s" compile option and external procedure directive allow a large project to be logically divided into many parts.

3. OS-9/68000 Pascal extensions
   a. OTHERWISE selector in CASE statement.
   b. Identifiers may contain '_' and '$' characters.
   c. External procedure directive.
   d. Integer 16 bit and Linteger 32 bit signed types which are automatically converted from one form to the other as required by the compiler.
   e. String data types with dynamic size from 0 to 255.
   f. Real functions ARCSIN, ARCCOS, TAN, and LOG.
   g. Random access of any pascal file with UPDATE, REPOSITION, GET, PUT, READ, and WRITE.
   h. String procedures DELETE, INSERT, and STR.
   i. String functions LENGTH, COPY, CONCAT, and POS.
   j. Array procedures and functions SCAN, CAP, FILLCHAR, MOVELEFT, and MOVERIGHT.

   k. Programatic control of error conditions.
      1. Math over/under flow control with MATHABORT and MATHRESULT variables.
      2. Individual file error control with IOABORT, IORESULT, and IOSYSERR.
   l. Operating system call with OS9 procedure.
   m. Bit field manipulations with FIELDGET and FIELDPUT routines.
   n. Dynamic heap management with SIZEOF, VARNEW, VARDISPOSE, PROGINFO, and HEAPREQUEST.
   o. Runtime parameter processing with SYSPARAM array and SYSPARAMSIZE.

4. OS-9/68000 Pascal optimizations
   a. Constant folding.
   b. Sparce case statement optimizations.
   c. Enumerations with 256 or less elements are stored in one byte.
   d. Branch optimizations. All branches in short displacement range are coded as short branches.
   e. Peephole optimization is applied to icode and assembly code.
   f. Extention of stack is kept in registers for fast expression execution.

5. OS-9/68000 Pascal error handling
   a. Program walkback when program aborts.
   b. Runtime checks of pointer dereferences, subrange assignments, array index, and parameter passing.
   c. Programatic control of math and file errors.

6. OS-9/68000 Pascal code generation
   a. Generates position-independent, reentrant, ROMable code.
   b. Extremely efficant code generation, producing extremely fast and compact programs.
   c. High compilation speed.

DO YOU KNOW PASCAL?

If you already know Pascal, you will be pleased to discover that OS-9/68000 Pascal is a very thorough implementation of the language according to ISO Standard 7185.1 Level 0 with exceptions and extensions documented in this manual. The compiler behaves as the Wirth and Jensen "bible" says it should. You will discover that a number of very useful additional library functions that are not included in the ISO specification have been added to OS-9/68000 Pascal. Also, some unnecessarilly restrictive Pascal syntax requirements have been relaxed.

If you don't already know Pascal, you have some studying to do. Fortunately, Pascal was originally designed for teaching programming, so it is easy to learn in stages. Unfortunately, a course in Pascal programming is beyond the scope of this manual. The books listed on the next page are recommended as reference and self-study source books. They are generally available at, or through, many larger bookstores, or they can be ordered directly from the publishers.

SUGGESTED ISO PASCAL SOURCE BOOKS

PROGRAMMING IN PASCAL, REVISED EDITION by Peter Grogono
Addison-Wesley Publishing Co., Reading, Mass., 1980

This book presents a good self-study course on Pascal for beginners. It is based on the ISO Pascal Standard which is important for compatibility with OS-9/68000 Pascal. There are a large number of similar volumes on bookstore shelves, including many good self-study courses.

PASCAL USER MANUAL AND REPORT by Kathleen Jensen and Niklaus Wirth
Springer-Verlag, New York, 1974

This book is the "bible", written by the creators of the language itself. The first part of the book is a "user manual" that shows how Pascal programs are constructed. The second part is a "report" giving a concise description of Pascal's syntax. An invaluable reference work. When expert Pascal programmers argue over trivial points, this book is consulted to settle matters.

STANDARD PASCAL USER REFERENCE MANUAL by Doug Cooper
W.W. Norton & Co., New York, 1983

The ISO Pascal Standard document was written by and for computer scientists. This book does an admirable job of translating the ISO specification for those "with only human powers of understanding", as Mr. Cooper puts it. Fortunately, the result is still technically accurate. You may find this to be a better reference than the Wirth/Jensen classic.

REQUIREMENTS AND INFORMATION

Requirements: OS-9/68000 Computer
disk drive(s)
OS-9/68000 Pascal distribution disk(s)

Before using your Pascal, it is important to create a BACKUP of your OS-9/68000 distribution disks. The original disks should then be stored in a safe place. To protect the information that you store on diskette, make backups of your working disks frequently.

The steps taken to create backup disks on a two drive system are as follows.

1. FORMAT a new disk.
   Place a disk in drive 1 and type FORMAT /D1. The format utility will print a ready prompt. Type 'Y' and the disk will be formatted.

2. BACKUP the disk.
   For a two drive backup, place the source disk in drive 0, a formatted disk in drive 1, and then type BACKUP. The backup utility will prompt for any input.

INSTALLING THE PARTS OF THE PASCAL SYSTEM

Your copy of the OS-9/68000 Pascal system will consist of one or more diskettes depending on the paticular disk size ordered. Before you use Pascal, you must copy these files onto your system disk.

The on the following page below lists the files included in the OS-9/68000 Pascal System. Each file must be copied from the distribution disk to the directory specified.

FILE NAME	FUNCTION	DIRECTORY
pp68	Pascal executive code file	CMDS
pc68	Pascal compiler code file	CMDS
pt68	Pascal icode translator code file	CMDS
r68	OS-9/68000 macro assembler code file	CMDS
l68	OS-9/68000 linker code file	CMDS
pc.initfile	Pascal compiler initialization file	CMDS
pc.errors	Pascal compiler error message file	CMDS
p.errors	Pascal run-time error message file	CMDS
math1	Floating point functions trap handler	CMDS
math2	Transcendental functions trap handler	CMDS
pmacros.a	Pascal assembly language macro file	/dd/DEFS
pas.l	Pascal support routines library file	/dd/LIB

If your disk capacity is limited, You should create a special system disk for working with Pascal that has only the minimum set of commonly used OS-9 commands (such as COPY, DEL, DIR, etc.) and omits those commands not frequently used when working with Pascal (such as FORMAT, DCHECK, OS9GEN, etc.).

DESCRIPTION OF OS-9/68000 PASCAL FILES

pp68

"pp68" is the pascal executive program. This program scans parameters passed to it and forks the other pascal programs to complete the four phases of compiling a pascal program. These phases are briefly described below.

PHASE	PROGRAM	RESULT
1) Compilation	pc68	Compile a PASCAL source program into ICODE file.
2) Translation	pt68	Translate ICODE file into assembly language file.
3) Assemble	r68	Assemble 68000 assembly language file into relocatable object file.
4) Link	l68	Link relocatable object file(s) into an OS-9 executable object module.

pc68

"pc68" is the pascal compiler program. This program takes as input a pascal source file and produces an icode file.

pt68

"pt68" is the pascal icode translator program. This program takes as input an icode file and produces a 68000 assembly language file.

r68

"r68" is the OS-9/68000 macro assembler program. This program takes as input a 68000 assembly language file and produces a relocatable object file.

l68

"l68" is the OS-9/68000 linker program. This program takes as input relocatable object files and librarys and produces an OS-9 executable object module.

pc.initfile

"pc.initfile" is an initialization file for the "pc68" program.

pc.errors

"pc.errors" is the compiler error message file which is used by the "pc68" program.

p.errors

"p.errors" is the run-time error message file. All pascal programs use this file to report run-time errors.

math1

"math1" is the floating point functions trap handler module. This module is automaticly loaded when needed by any pascal program.

math2

"math2" is the floating point transcendental functions trap handler. This module is automaticly loaded when needed by any pascal program.

pmacros.a

"pmacros.a" is the pascal macro definition file. This file is used during the assembly phase.

pas.l

"pas.l" is the pascal support library file. This file is used during the link phase.

FILE NAME SUFFIX CONVENTIONS

The executive program "pp68" accepts four types of source files, provided each file has the relevant suffix as shown on the following page.

Suffix	Usage
.p	PASCAL source file
.i	ICODE source file
.a	ASSEMBLY language source file
.r	RELOCATABLE object file

COMMAND LINES AND THE PASCAL EXECUTIVE

The OS-9/68000 Pascal compiler system is managed by a small executive program called "pp68". The executive accepts a single simplified command line and automatically calls the other parts of the compiler system as needed.

The syntax of the executive command line is shown below.

pp68 [option-flags] file {file} [option-flags]

One file at a time can be compiled, or a number of files may be compiled together. You can mix any combination of PASCAL source files, ICODE files, ASSEMBLY language files, or RELOCATABLE object files on the command line. The executive manages the compilation through up to four phases, compilation to icode, translation to assembler code, assemble to relocatable object, and linking to a binary executable code module (in OS-9/68000 memory module format). Complete descriptions of the operation of these four phases follow.

PHASE ONE (Compilation)

Each file on the command line is scanned for a suffix of ".p" which indicates a pascal source file. When a pascal source file is found the program "pc68" is invoked to compile the pascal source file into an icode source file. The icode source file will be in the same directory as the pascal source file but it will have a suffix of ".i" which indicates an icode source file. Then the pascal source file name on the command line will have it's suffix changed to ".i" and it will be flagged as a temporary file.

PHASE TWO (Translation)

Each file on the command line is scanned for a suffix of ".i" which indicates an icode source file. When an icode source file is found the program "pt68" is invoked to translate the icode source file into a 68000 assembly language file. The assembly language file will be in the same directory as the icode source file but it will have a suffix of ".a" which indicates an assembly language file. If the icode source file was flagged as temporary, from PHASE ONE, and the "-k" option isn't specified then the icode source file will be deleted. Also the icode source file name on the command line will have it's suffix changed to ".a" and it will be flagged as a temporary file.

PHASE THREE (Assemble)

Each file on the command line is scanned for a suffix of ".a" which indicates an assembly language source file. When an assembly language source file is found the program "r68" is invoked to assemble the assembly language source file into a relocatable object file. The relocatable object file will be in the same directory as the assembly language source file but it will have a suffix of ".r" which indicates a relocatable object file. If the assembly language source file was flagged as temporary, from PHASE TWO, and the "-k" option isn't specified then the assembly language file will be deleted. Also the assembly language source file name on the command line will have it's suffix changed to ".r" and it will be flagged as a temporary file.

PHASE FOUR (link)

If the "-s" seperate compilation option flag is specified then this phase isn't executed, and the relocatable object files produced by PHASE THREE are kept for future linkage, otherwise the program "l68" is invoked to link, all of the relocatable object files, any pascal library(s) specified with the "-l=<path>" compile option, and the standard librarys PAS.L (pascal support library) and SYS.L (system definition library) to produce an executable OS-9/68000 object module in the system execution directory. The name of the object module will be the name of the first file on the command line minus it's suffix unless a name is given using the '-n=<name>" compile option. All of the relocatable object files flagged as temporary, from PHASE THREE, will be deleted if the "-k" compile option isn't specified.

Note. If multiple source files are specified on the command line and the "-s" (seperate compilation) option isn't specified then the first file on the command line must be a complete pascal program. IF this file is a pascal source file then it needs to have an outer code block else if this file has a suffix other than ".p" then it needs to have been previously compiled without the "-s" option specified. This ensures that the first relocatable object module sent to the linker will be a mainline segment. Also all subsequent files on the command line will be compiled with the "-s" option enabled if it's suffix is ".p", otherwise if it's suffix isn't ".p" then the file has to have been compiled with the "-s" option specified. This ensures that these files won't be a mainline segment since only one is allowed per linkage.

EXECUTIVE OPTIONS

The executive program "pp68" recognizes many command line option flags which modify the compilation process. All flags are recognized before compilation commences so the flags may be placed any where on the command line. Flags must be seperated by either a comma (",") or blank (" ") character. The following table shows the flags recognized by the executive.

FLAG	USE
-B<number>	Used during PHASE ONE (compilation). The program "pc68" will have it's stack area increased by the specified <number> of bytes. This is useful if the program "pc68" aborted because of a stack overflow. The default stack area for the "pc68" program is set to a value such that all but the most extremely nested pascal source program will compile without this option specified.
-D<number>	Used during PHASE ONE (compilation) and PHASE TWO (translation). This option specifies the <number> of print lines per page of output. The default lines per page is 24 if this option isn't specified. Also if <number> is less than 12 or greater than 260 then the approprate limit will be used.
-D	Used during PHASE ONE (compilation). This flag disables debug code generation for the entire program. Refer to page 2-13 for information on debug code generation.

FLAG	USE
-E=<number>	Used during PHASE FOUR (link). This option sets the edition of the OS-9 program module to the <number> given.
-F	Used during PHASE ONE (compilation), PHASE TWO (translation), and PHASE THREE (assemble). This option enables the use of the FORM FEED character ($C) for a page eject. The default, if this option isn't specified, is to use carriage return(s) ($D) for a page eject.
-G	Used during PHASE FOUR (link). This option enables the generation of the symbol table file ".stb" for use with the "debug" program.
-I	Used during PHASE ONE (compilation). This option allows only ISO PASCAL syntax to be compiled. IF any OS-9/68000 Pascal extension is used it will be flagged as an error. This allows pascal programs to be developed under OS-9/68000 Pascal and ported to any other processor with an ISO pascal compiler.
-K	Used during all phases. This option allows all temporary files, icode (".i"), assembly language (".a"), and relocatable object (".r") created by each phase to be kept rather than deleted.
-L	Used during PHASE ONE (compilation). This option enables full compilation output listing. The default is partial list mode if this option isn't specified.

FLAG	USE
-L=<path>	Used during PHASE FOUR (link). This option specifies <path> as a library file. During the link phase these library files will be searched first to resolve any external references, i.e. any external procedures or functions. Multiple "-L=<path>" options are allowed on the command line and the order of searching is from the first "-L=<path>" option specified to the last one specified (left to right on the command line).
-M=<number>	Used during PHASE FOUR (link). This option adds <number> Kbytes to the stack allocation for the pascal object module. This additional stack size is in addition to any default size or any size defined with the "-S" option.
-N=<name>	Used during PHASE FOUR (link). This option naming convention. The executable object module will have its name set to <name>. Used during PHASE ONE (compilation). This option enables the inclusion of pascal source lines in the icode (".i") and assembly language (".a") files. This option is useful, with the "-k" option, such that the pascal source will be included in the assembly language file as comments and can be examined by the programmer.
-Q	Used during PHASE ONE (compilation), PHASE TWO option enables the quiet list mode. No output listing will be produced. Only error(s) will be listed if they occur.

FLAG	USE
-S	Used during PHASE ONE (compilation). This option pascal source files, on the command line, will be compiled to relocatable object file (".r") form. In each source file both the "PROGRAM" statement and the "outer-code-block" is optional. Only the terminating period is needed to terminate a pascal source file. Also all procedures and functions, declared at the global level (lev = 1), will have there first eight characters of there name used to define a global name which can be used by the linker to resolve external references. Refer to page 2-23 for more information on seperate compilation.
-S<number>	Used during PHASE TWO (translation). This option pascal object module. If this option is ommited or <number> equals zero then the stack allocation is set to the sum of all local and extended stacks plus 256, otherwise the stack allocation size is set to sum of the global data stacks (i.e. Procedure 0 local and extend stacks) and this <number>. This option allows the programmer to allocate a larger stack allocation size for highly recursive routines or to specify an allocation size smaller than the default stack size.
-T	Used during PHASE ONE (compilation). This option dump for each procedure or function declared in the Pascal source.
	Used during PHASE ONE (compilation). This option as <number>. All print lines except symbol table dump and program information table lines will be truncated if they exceed this <number>. The default print width is 79 if this option isn't specified. Also if <number> is less than 18 or greater than 260 then the approprate limit will be used.

TEMPORARY FILES

A number of temporary files are created and deleted in the current working data directory during the compilation, translation, and assemble phases. It is important to ensure that enough space is available in this directory.

EXAMPLE COMMAND LINES

```
pp68 sieve.p
```

Compile a pascal source file "sieve.p" in the current working data directory to an executable program module with a name of "sieve" in the current working execution directory.

```
pp68 part1.p part2.p -q -s
```

Compile the pascal source files "part1.p" and "part2.p" in the current working data directory to relocatable object files with the names of "part1.r" and "part2.r" in the current working data directory. Quiet list mode is selected and seperate compilation mode is enabled.

```
pp68 /dd/pascal/project/alpha.p part1.r part2.r -q
```

Compile the pascal source file "alpha.p" in the directory "/dd/pascal/project" and link with the relocatable object files "part1.r" and "part2.r" in the current working data directory to an executable program module with a name of "alpha" in the current working execution directory.

STEPS IN TESTING A PASCAL PROGRAM

The steps in creating and testing a program in OS-9/68000 Pascal are listed on the following page.

1. Create the Pascal source program using a text editor.

2. Compile the program to an OS-9 native code module using the "pp68" executive program.

3. If there are compilation errors, edit the Pascal source file and go back to step 2.

4. Run the program.

5. If there are run-time errors or program errors, edit the Pascal source file and go back to step 2.

THE PASCAL PROGRAM SOURCE FILE

The source file containing the program to be compiled is prepared using the system text editor. The source file generally resides in the "PASCAL" or current working data directory. In preparing your source program, keep the following rules in mind.

Source lines must be less than 200 characters in length.

The program can contain a maximum of 4095 PROCEDURES or FUNCTIONS.

PROCEDURE or FUNCTION declarations can be nested to a maximum depth of 31.

RECORD declarations can be nested to a maximum depth of (47 - "declaration-level-of-current-routine"). For example, if the current procedure declaration level is 1, which is a global procedure, then the record declarations can be nested to a depth of 46. However if the current procedure declaration level is 31, which is the deepest level allowed, then the record declarations can be nested to a depth of 16.

The maximum number of open WITH statement records are the same limits as the RECORD declarations described above.

More than one source file can be compiled together by using the $INCLUDE compiler directive in the source code. Included source files can be nested to a depth of 3.

COMPILER DIRECTIVES

Compiler directives are an OS-9/68000 Pascal extension that allow the programmer to format the compilation listing, include multiple source files, or to set compilation mode flags. A compiler directive must start in column one of a source line and both upper and lower case letters match. Compiler directives are not printed in the listing unless an error occurs on them. When a compiler directive is recognized the entire source line is used solely for the directive, no program text will be recognized on the directive source line.

LISTING DIRECTIVES

$TITLE

The "$TITLE" directive defines a character string which will be printed on the first header line of every listing page. This character string will remain in effect until another "$TITLE" directive is found. The character string starts with the first non blank character which follows the "$TITLE" directive. The appearance of this directive causes a page eject to be queued, that is a page eject is not immediately made but is delayed until the next time a source line is encountered which needs to be printed. This allows combinations of "$TITLE", "$SUBTITLE", and "$PAGE" directives to be grouped together and cause only a single page eject.

$SUBTITLE

The "$SUBTITLE" directive is identical to the "$TITLE" directive described above except that the character string will be printed on the second header line.

$PAGE

The "$PAGE" directive causes a page eject to be queued. Refer to the "$TITLE" directive above for a description of page eject queuing.

$INCLUDE

The "$INCLUDE" directive allows a new source file to be used in place of the "$INCLUDE" line. The new source file path name is specified on the "$INCLUDE" directive line. This new source file is used until it reaches it's end-of-file at which time the original source file will start up at the source line after the "$INCLUDE" directive. The file being included can itself contain "$INCLUDE" directives. In fact, "$INCLUDE" directives can be nested up to three levels. When source lines from an included file are printed they will have a plus ("+") character after the source line number.

COMPILATION MODE DIRECTIVES

$SET

The "$SET" directive allows the programmer to enable or disable various compile time option flags. Any number of options may follow on the "$SET" directive line and each of them must be seperated by a comma (",") or blank (" ") character. Upper and lower case letters are considered equivalent. The options and their uses are described below. Each option remains in effect until changed with another "$SET" directive or by a "$POP" directive.

OPTION	USE
L+	Enable program listing.
L-	Disable program listing.

This option has effect only when the "-L" (enable compilation listing) executive option is specified.

OPTION	USE
D+	Enable debug code generation.
D-	Disable debug code generation.

If the "-D" (disable debug code generation) executive option is specified then debug code generation is always disabled. Otherwise, if the "d+" compilation mode is enabled when a PROCEDURE, FUNCTION, or PROGRAM statement is scanned then debug code will be generated for that routine. This debug code performs the following operations.

All pointer references are checked to be within the allocated heap area.

Array index expressions are checked to be within the declared array bounds.

Any assignment to a subrange variable or passing of an expression to a subrange parameter is checked to be within the bounds of the subrange.

Statement offset table is generated. If a run-time error occurs in this routine then the statement number in error will be reported.

Debug code is generated to check for math over/underflow error for the following operations.

Addition of integer or linteger arguments.

Subtraction of integer or linteger arguments.

Negation of integer or linteger arguments.

chr(argument), check argument range.

multiplication of integer arguments.

check range of conversion from linteger to integer.

The selecting expression of a CASE statement is always checked.

OPTION	USE
S+	Enable stack check code generation.
S-	Disable stack check code generation.

If the "S+" compilation mode is set when a PROCEDURE or FUNCTION statement is scanned then the compiler will generate code to check that enough area is available for the routine's stack frame (i.e. local and extended stacks areas). This option should not be disabled unless the user fully understands the use of the stack by the running pascal program.

OPTION	USE
X+	Enable long branch code generation.
X-	Disable long branch code generation.

It is possible for a pascal routine to be too large such that a 68000 branch instruction, of 16-bit displacment, will be out of range. If this occurs, then the "X+" compilation mode should be enabled for those routines. When this mode is enabled and a PROCEDURE, FUNCTION, or PROGRAM statement is scanned, the compiler will generate, for each branch instruction in the routine, code that has a 32-bit displacment. This option should only be enabled for those routines that need it because each branch with this option enabled will take 10 bytes of code as compared to 2 or 4 bytes of code with the option disabled.

DEFAULT MODE VALUES

At the start of compilation the default compilation mode settings are: D+, L+, S+, and X-.

$PUSH

The "$PUSH" directive saves the current values of the four compilation mode flags (D,L,S,X) on a last-in first-out push down stack. This stack has a depth of 20 entries.

$POP

The "$POP" directive restores the values of the four compilation mode flags (D,L,S,X) from the last entry saved on the push down stack. If the push down stack had no entries then this option has no effect on the compilation flags.

EXAMPLE USE OF COMPILATION MODE DIRECTIVES

The following program shows some example uses of compilation mode directives.

```
$SET d-
PROGRAM dummy; {Debug code isn't generated for outer-block }
$PUSH
{ save current compilation modes }
$set S- d+
FUNCTION getchar : char; { No stack check code, generate debug
                           code for this routine }
   BEGIN
   getchar := input^;
   get(input);
   END;
$Pop
{ restore saved compilation modes }
PROCEDURE putchar(ch : char); { Check stack code, no debug }
   BEGIN
   output^ := ch;
   put(output);
   END;

BEGIN {outer-block}
WHILE NOT eof(input) DO
   putchar(getchar);
END.
```

A SAMPLE COMPILATION LISTING

As an aid in describing how to interpret the compilation listing, a small source program (Figure 1) is shown below, and the corresponding compilation listing (Figure 2) is shown on the next page. The program is designed to demonstrate most of the features of a compilation listing and deliberately includes errors. Refer to these two examples for the discussion that follows.

Figure 1
SAMPLE SOURCE PROGRAM

```
$TITle DumpReal
$SUBTITLE Global Definitions
PROGRAM dumpreal;
VAR
   badvar  : ^anqtherbadvar;
   i       : integer;
   hexc    : PACKED ARRAY [1..16] OF char;
   overdef : RECORD
             CASE boolean OF
          true  : (r         : real);
          false : (c         : PACKED ARRAY [1..8] OF char);
             END;
$subtitle PROCEDURE procwitherrors
PROCEDURE procwitherrors;
   BEGIN
   this demonstrates what error messages
   look like;
   END;
$Subtitle PROCEDURE hexval
PROCEDURE hexval(ch : char);
   BEGIN
   write(hexc[(ord(ch) >> 4) + 1], hexc[ord(ch) & $F]);
   END;
$SUBtitle M A I N L I N E
BEGIN
hexc := '0123456789abcdef';
WHILE true DO
   BEGIN
   write('Enter a real number : ');
   readln(overdef.r);
   FOR i := 1 TO 8 DO
      BEGIN
      hexval(overdef.c[i]);
      write(' ');
      END;
   writeln;
   END;
END.
```

With the program in figure 1 in the current working directory, it was compiled to produce the listing that follows using the command line:

```
pp68 dumpreal.p -l -w75
```

The command line initiates compilation of a pascal source file called "dumpreal.p" in the current working data directory. The list option is enabled and the print width is set to 75.

Figure 2
SAMPLE COMPILATION LISTING

```
Page   1  85/07/01  21:05:44  OS-9/68000 Pascal - release 1.0  DumpReal
 LINE    STMT LEV   Global Definitions
------+-------+--+---------1---------2---------3---------4---------5-------
     3            PROGRAM dumpreal;
     4            VAR
     5      0D       badvar  : ^anotherbadvar;
     6     -4D       i       : integer;
     7     -6D       hexc    : PACKED ARRAY [1..16] OF char;
     8    -22D       overdef : RECORD
     9    -22D  1                CASE boolean OF
    10    -22D  2           true  : (r         : real);
    11    -22D  2           false : (c         : PACKED ARRAY [1..8] OF char);
    12    -22D  2                END;


Page   2  85/07/01  21:05:44  OS-9/68000 Pascal - release 1.0  DumpReal
 LINE    STMT LEV   PROCEDURE procwitherrors
------+-------+--+---------1---------2---------3---------4---------5-------
    14    -30D     PROCEDURE procwitherrors;
       *****                 ^104,117
104: Identifier is not declared.
117: Unsatisfied forward reference.
*ERROR, The following type-id's were assumed to be forward declared.
ANOTHERBADVAR
    15          1     BEGIN
    16       0  2     this demonstrates what error messages
       *****             ^104          ^59  *SEE: 14
 59: Variable expected.
104: Identifier is not declared.
    17       1  2     look like;
       *****                 ^51  *SEE: 16
 51: ':=' expected.
    18       1  2     END;
```

FIGURE 2 (continued)

```
 Page   3  85/07/01  21:05:44  OS-9/68000 Pascal - release 1.0  DumpReal
  LINE    STMT LEV   PROCEDURE hexval
------+-------+--+---------1---------2---------3---------4---------5-------
    20          1 PROCEDURE hexval(ch : char);
    21          1    BEGIN
    22       0  2    write(hexc[(ord(ch) >> 4) + 1], hexc[ord(ch) & $F]);
    23       1  2    END;
```

```
 Page   4  85/07/01  21:05:44  OS-9/68000 Pascal - release 1.0  DumpReal
  LINE    STMT LEV   M A I N L I N E
------+-------+--+---------1---------2---------3---------4---------5-------
    25          1 BEGIN
    26       0  1 hexc := '0123456789abcdef';
    27       1  1 WHILE true DO
    28       2  2    BEGIN
    29       2  2    write('Enter a real number : ');
    30       3  2    readln(overdef.r);
    31       4  2    FOR i := 1 TO 8 DO
    32       5  3       BEGIN
    33       5  3       hexval(overdef.c[i]);
    34       6  3       write(' ');
    35       7  3       END;
    36       7  2    writeln;
    37       8  2    END;
    38       8  1 END.
```

```
 Page   5  85/07/01  21:05:44  OS-9/68000 Pascal - release 1.0  DumpReal
  LINE    STMT LEV   M A I N L I N E
------+-------+--+---------1---------2---------3---------4---------5-------
PROC G NAME          DEF'D      BODY      LOCAL     EXTEND     ICODE     STRING
   0   DUMPREAL          3        25         32         18       214         38
   1 * PROCWITH        14         15          0          0         9          0
   2 * HEXVAL          20         21          0          8       106          0
                                              32         26       329         38

38 Lines of source code compiled with 5 errors found, see: 17


Compilation aborted on file dumpreal.
Error #1:1
```

LISTING PAGE HEADINGS

A three line page header is printed at the top of every listing page. The word "Page" is followed by a page number, which is followed by the current system date and time. The date and time is read only once at the beginning of a compilation, and this single time stamp is used for the heading of each page thereafter. To the right of the time stamp is the release level. Always be sure that the release level corresponds to the user manual or other reference documents that you may be using.

To the right of the release level is the program title string which was specified by a $TITLE directive in the program (see page 2-12). Note how the title string in figure 2 is derived from the title directive in figure 1.

The second line of figure 2 is typical of the form of every page heading. The column titled, 'LINE', refers to the sequence number of the corresponding source line. Note that the first line number is 3, because line numbers 1 and 2 of the source text in figure 1 contain title and subtitle options respectively which are never themselves printed.

The next column titled 'STMT' refers either to the variable (data) location counter or to the statement location counter, depending on whether the corresponding source line is a variable declaration statement or an executable program statement.

Source line number 6, for instance, shows that the data location counter is currently at -4 when the line is being compiled. The number is negative because data storage is assigned backwards from the stack frame pointer. Since the line indicates that the global variable 'i' is being defined, and it is an integer which requires two bytes of storage, 'i' will be assigned to data location -6 in the global stack frame. Likewise, for line 7 of the program, the current data location counter is at -6 from the assignment of variable 'i' on the preceding line. The variable 'hexc' is being defined which is a 16 element character array. Since character type elements require one byte of storage, the 'hexc' variable will require 16 bytes of storage; thus, 'hexc' will be assigned to data location -22. To find where the variable 'overdef' is assigned, look at the listing for source line 14; 'overdef' is assigned to data location -30.

Statement numbers are assigned beginning at location zero for every procedure. The 'write' call in line 22 then begins at statement location 0 for the procedure. Each subsequent executable pascal statement will cause the statement location counter to be incremented. The importance of showing the statement location is for understanding the run-time error messages. When an error occurs during the execution of a program, the procedure which caused the error and, if it is known, the statement location within the procedure is reported. You can then easily find where within your program that the run-time error actually occurred. Numbers under the 'STMT' column, which contain data location counter values, are always followed by the letter 'D', and statement location counter values are always followed by a space character.

The next column titled 'LEV' represents the compiler's 'lexical' level. The lexical level shows the nesting of control structures (loops, etc.) within the program and is helpful for finding unbalanced compound statements and declarations. Generally, those statements which have the same lexical level number are at the save level of nesting during the compilation scan of your program. Since the compiler always prints the image of a line of source program before it begins its scan of that line, the lexical numbers can sometimes lag behind what you might think they should actually be. Due to the number and complexity of conditions which cause the level number to be incremented and decremented, the best advice is to simply look at the numbers on your listings and gain a feel for how they move up and down.

A few spaces to the right of the title 'LEV', on the second header line, is the subtitle string. This is defined by the "$SUBTITLE" directive - see page 2-12.

The third line of each page heading is used to mark column boundaries as a visual aid. After the first 16 characters, which marks the required prefix of every line of source listing, is a scale showing the position of every 10th column of source text. If a parameter is used to define a short line width, then only that part of the scale up to the print line width is shown.

ERROR MESSAGES IN PROGRAM LISTINGS

There are two types of error messages. The first of which is shown in figure 2 after the listing of source line 14. The compiler displays a line beginning with four asterisks followed by an up arrow which indicates within a character position or two where the compiler was scanning when it realized that an error had occurred. Following the up arrow, the compiler prints the error number(s) for the error(s) found. On the following line(s) the compiler will print the full english error message text if it is able to open the file "pc.errors". This particular error also generated the second type of error message. Specifically, the message shown indicates that at this point in scanning the source program, the compiler realized that the identifier called 'ANOTHERBADVAR' was previously declared to be a forward reference to an identifier which was not found by the time it was required to be found.

A more complete example of an error message is shown following the listing of source line 16. Here, two errors were discovered for the source line, and the up arrows and error numbers appear as described above. Also, since this was not the first error message for the program, the text '*SEE: 14' is added as a programmer aid. The 'SEE' tells you that the last error message previous to this current message occurred after the listing of source line 14. If you look at the end of figure 2, you can see a message indicating that 5 errors were found in the source program, and that the last message appeared after the listing of source line 17. The error message after source line 17 points you to the previous error message and that message points you to its preceding error message and so forth. You can quickly find all the error messages in your program without going through the whole listing. Error messages of the second type always occur with error messages of the first type, so you will be able to find all error messages by following the backward list.

Up to 9 error messages will be reported for any one line of source code. If more than 9 errors are found, a message is triggered that indicates that too many errors were found for the line. Sometimes one error will trigger other errors in the program. For example, if a variable declaration is being scanned and the reserved word 'PROCEDURE' is encountered, the compiler may get temporarily confused as to what it should really be scanning and multiple errors may be triggered as the compiler attempts to get on track again. Therefore, correcting one error may get rid of several other error messages.

If the listing of the source program is inhibited by compiler option(s), only source lines having errors and the corresponding error messages will be listed to let you perform "error checking" of long source programs without having to print the entire program.

THE PROCEDURE STATISTICS TABLE

The procedure statistics table is always displayed after a compilation unless the '-Q' executive option was specified. This table is not affected by the line width parameter option. The procedure table gives important information about the program and variable storage requirements of each procedure compiled and is helpful when interpreting any run time error messages. The procedure statistics table looks like the following example.

PROC	G	NAME	DEF'D	BODY	LOCAL	EXTEND	ICODE	STRING
0		DUMPREAL	3	25	32	18	214	38
1	*	PROCWITH	14	15	0	0	9	0
2	*	HEXVAL	20	21	0	8	106	0
					32	26	329	38

The information presented in each column is:

"PROC" is the procedure number assigned by the compiler. When run-time errors occur, they will be identified by this number. Procedure number zero always refers to the "outer-code-block" the main program itself.

"G" is the global level procedure field, if this field has an "*" printed in it then this procedure was declared at the global level.

"NAME" is the first eight characters of the procedure name. When run-time errors occur, they will be identified by this name, also this name will be used to define a global reference if the '-S' (seperate compilation) executive option is specified and the procedure is defined at the global level.

"DEF'D" is the source line number of the procedure's declaration.

"BODY" is the source line number of the procedure's code body.

"LOCAL" and "EXTEND" columns give the number of bytes that the compiler has calculated will be required for each procedure's variable storage (e,g., local and extended stack areas).

"ICODE" gives the number of bytes of ICODE instructions generated for the procedure.

"STRING" is the length (in bytes) of string constants used by the procedure. A string constant is any sequence of two or more characters enclosed within single quote marks.

SEPARATE COMPILATION

Seperate compilation is a feature of OS-9/68000 Pascal that allows for the building of pascal librarys or for the ability to seperately compile the individual parts of a large pascal program.

When the seperate compilation mode is enabled, either by the "-S" executive option or by a pascal source file not being the first file specified to the executive, all routines declared within the program at the global level will have their routine names included in the relocatable (".r") object file as global symbols. These symbols will be used to resolve the external routine references generated by the "EXTERNAL" compiler routine directive. Note: only the first 8 characters of the routine name are used to identify the routine in the relocatable object file so be sure that these names are unique.

EXAMPLE USE OF SEPARATE COMPILATION

The following example programs and command lines show an example use of seperate compilation.

EXAMPLE PROGRAM lib1.p

```
PROGRAM lib1;
{define global routine "ALPHANUM"}
FUNCTION alphanumeric(value : integer) : char;
   BEGIN
   IF (value & $FFF0) <> 0 THEN
      alphanumeric := '?'
   ELSE IF value < 10 THEN
      alphanumeric := chr(ord('0') + value)
   ELSE
      alphanumeric := chr(ord('A') - 10 + value);
   END;
```

EXAMPLE PROGRAM lib2.p

```
PROGRAM lib2;
CONST
   max_nibble = 8;

{define external reference "ALPHANUM"}
FUNCTION alphanumeric(v : integer) : char; EXTERNAL;

{define global routine "HSTRING"}
FUNCTION hstring(value : linteger; nibbles : integer) : string;
   VAR
      i     : integer;
      wstr  : string;
   BEGIN
   IF (nibbles < 1) OR (nibbles > max_nibble) THEN
      hstring := '<BAD_NIB_VALUE>'
   ELSE
      BEGIN
      wstr := '';
      FOR i := nibbles DOWNTO 1 DO
         wstr := concat(wstr, alphanumeric(
                              fieldget(value, i * 4 - 1, 4)));
      hstring := wstr;
      END;
   END;
```

EXAMPLE PROGRAM main.p

```
PROGRAM main(input, output);
VAR
   l    : linteger;

{define external reference "HSTRING"}
FUNCTION hstring(v : linteger; n : integer) : string; EXTERNAL;

BEGIN
WHILE NOT eof DO
   BEGIN
   readln(l);
   writeln(' ':2, hstring(l, 8),
           ' ':2, hstring(l, 6),
           ' ':2, hstring(l, 1),
           ' ':2, hstring(l, 0));
   END;
END.
```

Compile the library programs "lib1.p" and "lib2.p" using the following command line.

```
pp68 lib1.p lib2.p -s -q
```

Create the library by merging the relocatable object files with the following command line.

```
merge >/dd/lib/user.pas.l lib2.r lib1.r
```

Use the following command line to compile the program "main.p" using the library "user.pas.l" to resolve any external routine references.

```
pp68 main.p -q -l=/dd/lib/user.pas.l
```

To execute the compiled program enter the following command line.

```
main
```

This Page Intentionally Blank

CHAPTER 3
A PASCAL LANGUAGE SUMMARY

SYNOPSIS: This section is a summary of the rules used to construct Pascal programs. As mentioned in the Introduction, this manual is not intended to serve as a comprehensive language reference, so the main purpose of this chapter is to provide an overview of ISO standard Pascal plus some characteristics and features of OS-9/68000 Pascal that are system dependent. See Appendix C for a detailed Backus-Naur Form description of the language.

ALTERNATE CHARACTER SETS

The ISO alternate character set is supported by OS-9/68000 Pascal. This allows use of comments, arrays, and pointers on systems with limited character sets. The following table shows the allowable alternate characters:

Standard	Alternate	Usage
{	(*	Begin comment
}	*)	Close comment
[	(.	Begin array range/set element
]	.)	Close array range/set element
^	@	Pointer character

NOTATION USED IN DESCRIPTIONS

In order to give concise descriptions, this manual uses a simple notation system to show how parts of Pascal are used. The rules of this system are:

- Pascal keywords are capitalized.
- Things that the programmer specifies (such as expressions, variable names, definitions, etc). are shown in lower-case.
- Things that may optionally appear ONCE are enclosed by brackets [ ] .
- Things that may optionally appear ONE OR MORE times are enclosed by braces { }

SOURCE PROGRAM FORMAT

The Pascal compiler uses source files which can be prepared using any text editor. Input lines can be up to 200 characters long. Each input line is terminated by a carriage return (ENTER) character. No distinction is made between upper case and lower case letters except within string constants.

COMPILATION MODE

The OS-9/68000 Pascal compiler allows for full ISO Pascal conformance. IF the "-i" option on the compiler command line is specified then only ISO Pascal syntax will be accepted and all OS-9/68000 extentions will be flagged as errors. This allows the user to generate programs that can be ported to any other system which has an ISO Pascal compiler without modification. IF the "-i" option isn't passed on the compiler command line then all of the OS-9/68000 Pascal extentions will be allowed.

IDENTIFIER NAMES

Identifiers are names given to Pascal variables, types, procedure and function names, etc. Identifiers can be of any length and all characters are significant, i.e., used for matching and to determine uniqueness. They include any combination of the following characters.

```
'A'..'Z'   'a'..'z'   '0'..'9'   '_' '$'
```

Note: Use of the underscore or dollar characters is a nonstandard OS-9/68000 Pascal extension. This extension is disabled in ISO compilation mode.

The first character of an identifier must be an upper or lower case letter ('A'..'Z' or 'a'..'z') and upper and lower case letters always match. Here are some examples of Pascal identifiers.

```
x
sum
COUNT
Part20

Last_Year      matches    last_year

5Days          ILLEGAL - can't begin with digit
_temp          ILLEGAL - can't begin with "_"
```

GENERAL PROGRAM ORGANIZATION

A Pascal program starts with the "PROGRAM" statement followed by various optional declarations, then program statements. A general outline of program organization is shown below.

GENERAL PASCAL PROGRAM ORGANIZATION

```
PROGRAM name(files);

LABEL   <label declarations>
CONST   <constant declarations>
TYPE    <type declarations>
VAR     <global variable declarations>

PROCEDURE and/or FUNCTION subprograms

BEGIN
  <main program statements>
END.
```

The main program is called the "outer block", because it is executed first when the Pascal program is run. In simple programs, all the work may be done by the main program statements in the outer block, however, very frequently much of the work is done by subprograms called "procedures" and "functions". These subprograms can contain their own internal ("local") variable, type, label, and similar declarations, and have a similar organization as the outer block including the possibility of having additional subprogram definitions within themselves. All declarations and names declared within subprograms are only "known" within that subprogram or additional subprograms it may contain.

The PROGRAM Statement

The PROGRAM statement must be the first statement of the program. It states the program name and may optionally include names of files (I/O paths) used within the program. Unlike some other Pascal compilers, OS-9/68000 Pascal does not require that files be declared in the PROGRAM statement, however, if you do so they will be checked for correctness. Here are some examples of the PROGRAM statement.

```
PROGRAM test;

PROGRAM sort(infile, outfile, scratchfile);
```

COMMENTS AND BLANK STATEMENTS

Comments consist of text which begin with either a "{" or a "(*" character sequence and end with either a "}" or a "*)" character sequence. They are ignored by the compiler and may appear anywhere in a program. Comments may also extend over multiple lines. Here are some examples.

```
{ This is a comment }
(* This is also a comment *)
{ So
  is
  this }
{ This will also be a comment *)

{ outer comment { nested comment } outer comment }
```

NOTE: Comments may be nested if the ISO compilation mode isn't enabled. This is a nonstandard OS-9/68000 Pascal extension.

Blank lines may also be used freely to improve program readability.

INDENTATION OF LEXICAL LEVELS

It is traditional and considered good programming style (but is not mandatory) to indent statements in Pascal programs so that "lexical levels" of control structures (loops, compound statements, etc.) are clearly visible. Indenting makes the program logic much easier to read and understand. The compiler assists you by printing the lexical level of each statement on the compiler listing.

LABEL Declarations and GOTO Statements

"Labels" are used to indicate places in the program GOTO statements can transfer control to. Frequent use of GOTO statements is unnecessary in Pascal because of the wide variety of structured control statements. Excessive use of GOTOs can result in programs that are hard to read, test, and maintain. Use of GOTOs can be justified in a few special cases, such as error handling or exiting from a deeply nested loop.

Each label used in the program must be declared in the LABEL statement before it is used. Each label must be a unique number in the range of 0 to 9999. GOTO statements can transfer control to a statement outside the current procedure. Here is an example of its use.

```
PROGRAM label_demo;
LABEL 10, 20;
VAR
   x, temp      : integer;
BEGIN
ioabort(input, false);
FOR x := 1 TO 100 DO
   BEGIN
   readln(temp);
   IF ioresult(input) <> 0 THEN
      GOTO 10;  {end if input error}
   writeln('The square root of', temp, ' is ', sqrt(temp));
   END;
GOTO 20;
10:
writeln('input error!!!');
20:
END.
```

IOABORT and IORESULT will be discussed in Chapter 5.

CONSTANT Declarations and Constants

The CONST statement allows a name to be given to a constant value. Although constants may be used in the program without prior declaration, the CONST facility is useful for defining constants that may be changed in future or different versions of the program, and also to increase program readability.

The CONST keyword can be followed by any number of definitions of numeric or string constants. Each definition has the form:

```
<identifier) = <constant> ;
```

The constant definitions end when another Pascal keyword is encountered. On the following page is an example program segment illustrating its use.

```
CONST
   Pi        = 3.1415926;
   HighLimit = 22.5;
   Title     = 'Test Result Summary';
   EndLine   = $0D;
   chx       = 'X';
   null      = '';
```

Numeric Constants

Numeric constants can be of integer, linteger, or real types. Numeric constants may be preceded by "+" or "-" signs. Any numeric constant can contain an underscore after the first character. Integer constants are whole numbers in the range of 0 to 32767. Linteger constants are whole numbers in the range of 0 to 2147483647. OS-9/68000 Pascal will also accept integer and linteger constants in hexadecimal notation using a "$" followed by one or more hexadecimal digits (0-9, A-F). IF 1 thru 4 hexadecimal digits are specified then the value will be of integer type else the value will be of linteger type. Real constants are numbers that include decimal points and/or the exponential "E" notation. Real numbers are stored in the IEEE double precision format (8-byte) which represents approximately 17 significant digits. Here are some examples of numeric constants.

```
-5                legal integer constant -5
9999              legal integer constant 9999
9_999             legal integer constant 9999
$1DF              legal integer constant - hexadecimal 1DF
$1df              legal integer constant - hexadecimal 1DF
120000            legal linteger constant 120000
$1c49a2           legal linteger constant $1C49A2
$8000             legal integer constant -32768
$08000            legal linteger constant +32768

_612              ILLEGAL - can't start with "_"
1200000000        ILLEGAL constant - too big
$1C49A20000       ILLEGAL constant - too big

1.45              legal real constant 1.45
0.0303            legal real constant .0303
15E44             legal real constant 15*10^44
1.234e-12         legal real constant 1.234 * 10^-12
1_000_000.0       legal real constant 1000000.

_100_000          ILLEGAL constant - can't start with "_"
5._56             ILLEGAL constant - can't start with "_" even
                  after "."
```

String and Character Constants

String constants are sequences of 0 to 197 characters enclosed within single quotes. IF the string constant has only a single character inclosed within the quotes then it is considered a character constant. Here are some examples.

```
'X'                 {character constant}
'5'                 {character constant "5" NOT numeric value 5!}
'abcdefg'           {string constant}
'strings can have embedded spaces'      {string constant}
'PAGE 10'           {string constant}
''                  {string constant of zero length, this is a
                     nonstandard OS-9/68000 Pascal extention.}
```

The single quote character used to enclose strings may itself be contained within the string if two are used. Each pair of adjacent single quotes has the literal value of one single quote character. Here are some examples.

```
'Joe''s Grill'      has the actual value of:    "Joe's Grill"
'AABB''''DDEE'      has the actual value of:    "AABB''DDEE"
```

String constants are considered to be of type PACKED ARRAY [1..N] of char, where N is the length of the string constant. Type checking is performed when using a string constant. If a string constant is assigned to an array with a lower bound other than 1, an error will be reported.

Type and Variable Declarations

The TYPE and VAR statements are used to describe and declare variable storage, respectively. One of Pascal's outstanding features is its support for variables, arrays, sets, files, and record structures which can be composed on a number of basic "built-in" data types or user-defined data types. The subject of variable declarations in Pascal is quite complex and beyond the scope of this manual, however, the information that follows describes characteristics of OS-9/68000 Pascal that are specific to its particular implementation of the language.

HIERARCHY OF OS-9/68000 PASCAL DATA TYPES

```
Pointers

Structured Types:
   ARRAY
   STRING     {OS-9/68000 Pascal extension}
   RECORD
   FILE
   SET

Simple types:
   REAL
   Ordinal Types (subranges permitted):
       CHAR
       INTEGER (16-bit signed)
       BOOLEAN
       Enumerated (user-specified)
   LINTEGER (32-bit signed) {OS-9/68000 Pascal extension}
```

STORAGE ALLOCATION

The amount of actual memory space required for each simple variable, array element, or record element of the simple types are, CHAR and BOOLEAN one byte, INTEGER two bytes, LINTEGER four bytes, and REAL eight bytes. The 68000 microprocessor requires that word and long word operands be on even byte boundrays. Because of this requirement all structures with a size greater than one will be assigned to the next even byte offset. On the following page are some examples.

```
VAR
   ch          : char;     {Assigned offset of -1}
   i           : integer;  {Assigned offset of -4 because -3 ia an
                            odd byte offset}
   a           : PACKED ARRAY [1..100] OF char;
                           {Assigned offset of -104, each element of
                            the array is of one byte size}
   b           : ARRAY [1..2] OF
                    RECORD
                    ch     : char;
                    r      : real;
                    END;
                           {Assigned offset of -124, each element of
                            the array has a size of ten bytes.  This
                            is because the field "r" in the RECORD
                            definition is a structure with a size
                            greater than one byte so it will be
                            assigned to the next even byte offset,
                            which is 2, giving a size to the RECORD
                            of 10 bytes.}
```

To see the actual compiler generated offsets for each variable or record field within a program enable the "-t" dump symbol tables option on the compiler command line.

Arrays:

Arrays may be multi-dimensional and can be of any simple, structured, pointer, or user defined type. A new type definition can be included in the array declaration. The array index type must be an ordinal type. OS-9/68000 Pascal permits the character sequences "(." and ".)" to be used as array subscripts in addition to the standard brackets "[ ]".

Sets:

Sets may have up to 256 members each. The ordinal value of the lowest and highest set element allowable is 0 and 255 respectivily. If the set is to contain a series of successive elements, they can be specified using the form 'low..high' as in:

```
workday := [Sunday, Tuesday..Friday];
```

PACKED Structures:

Because the 68000 microprocessor uses byte addressed memory, all arrays, structures, etc., are inherently "packed" (except bit-level BOOLEAN). The PACKED attribute keyword is properly scanned for correctness by the compiler but has no actual storage effect on the structure.

FILE Types:

Each variable declared as a FILE type is associated with a 22-byte control block which is automatically initialized when the code block making the definition is executed, however, the file itself is not automatically opened (except for the standard I/O paths). When the block of code is exited, all files are automatically closed. FILE types cannot be passed by value in subprogram calls. File types are also allowed within RECORD and ARRAY definitions.

PROCEDURE AND FUNCTION DECLARATIONS

Procedures and functions may be declared within the main program, or within other functions or procedures. Declarations are almost identical to the PROGRAM declaration except for the keywords used (PROCEDURE or FUNCTION), and the ability to include a "formal parameter list" which is used to receive or return data. Otherwise, the body of a function or procedure is the same as the main program and may include "local" declarations for constants, labels, variables, etc, as well as executable statements.

PROCEDURE GENERAL FORMAT

```
PROCEDURE name(parameter-list);
LABEL label declarations
CONST constant declarations
TYPE type declarations
VAR local variable declarations
PROCEDURE or FUNCTION subprogram declarations
BEGIN
  procedure-statements;
END;
```

FUNCTION GENERAL FORMAT

```
FUNCTION  name(parameter-list) : result-type;
LABEL label declarations
CONST constant declarations
TYPE type declarations
VAR local variable declarations
PROCEDURE or FUNCTION subprogram declarations
BEGIN
  function-statements;
  {Note assignment to function name must occur within
   scope of funtion}
  name := result-type-value;
END;
```

PROCEDURES are called as statements in a program to perform some operation. FUNCTIONS are called from within expressions and return some value. The format of the parameter list declaration is similar to a VAR declaration and can specify a number of simple or complex variables of any type. If ISO compilation mode isn't enabled then the result-type of a function may be any structured type excluding file type (i.e. ARRAY, STRING, RECORD, or SET). This is an OS-9/68000 Pascal extension.

OS-9/68000 Pascal includes a large number of standard functions which are built-in functions that perform commonly used operations such as evaluation sines, cosines, exponentials, etc. Standard functions also comprise a large part of Pascal's input and output capabilities.

OS-9/68000 Pascal recognizes two special subprogram directives FORWARD and EXTERNAL. FORWARD is used to tell the compiler about the existance of a procedure in advance (e.g., before it is actually declared). FORWARD is required if the procedure(s) or function(s) involved are referenced before they can be declared. Here is an example.

```
PROGRAM forward_example;
  (declarations)
PROCEDURE second(c, d : integer); FORWARD;
PROCEDURE first;
   BEGIN
   a := 1;
   b := 2;
   second(a, b);              {call to FORWARD procedure}
   END;
PROCEDURE second;             {Second now declared param list}
   BEGIN                      {is NOT repeated}
   writeln(c);
   writeln(d);
   END;
BEGIN
first;
END.
```

The EXTERNAL directive tells Pascal that the procedure or function declared is an external assembly language routine created by seperate compilation or hand-written in 68000 assembly language. All external routines will be linked to the main program. This allows librarys of commonly used routines to be used without having to recompile the routines source code. Also, seperate compilation allows a large pascal program to be broken up into smaller units which can be worked on by several programmers. Here is an example of the EXTERNAL directive.

```
PROGRAM main;
VAR
   a, b, c    : integer;

PROCEDURE my_extern(x, y, z : integer); EXTERNAL;

BEGIN {outer block}
my_extern(a, b, c);    {call to external}
END.
```

ASSIGNMENT STATEMENTS AND EXPRESSIONS

The assignment statement evaluates an expression and assigns the result to a variable. Here are some examples.

```
result := temp + 5;

x := sum MOD base;

val := (total / count) + adj

answer := (last > first) AND NOT correct

name := 'STEVE';
```

In general, the variable in which the result is stored may be a pointer type, a simple type, a structured type (except file type), a specific element of an array, a function name, or a specific element of a record structure. The type of the variable and the type of the expression result must match (except when assigning an integer or linteger to a real, or an integer to linteger), or an error will be reported by the compiler. Automatic type conversion is performed when real, integer, or linteger types are mixed within an expression.

Expressions are made up of "operands" (data) and "operators". Operands may be constants, variables, array or record elements, set members, pointers, etc. Numeric operands may have their sign specified by use of the + or - unary argument operators (which are different than add and subtract operators).

The operators are evaluated in precedence (priority) order. If several operators of the same priority appear consecutively, they are evaluated from left to right. The normal precedence may be overridden by grouping of subexpressions in parentheses. The operators and data types that may be used are given in the table on the next page in order of group precedence the lowest precedent operator is evaluated first.

A number of standard functions that perform mathematical and logical functions are also available - see Chapter 7.

PASCAL OPERATORS

Prec	Operator	Operation	Operand type(s)	Result type
1	NOT	logical negation	boolean	boolean
2	*	multiplication	integer, linteger(note 3), or real	see note 1
2	*	set intersection	set	set
2	/	division	integer, linteger(note 3), or real	real
2	DIV	division with truncation	integer or linteger(note 3)	see note 2
2	MOD	modulo	integer or linteger(note 3)	see note 2
2	AND	logical AND	boolean	boolean
2	&	bit-by-bit AND (note 4)	integer or linteger	see note 2
2	<<	shift left (note 4)	integer or linteger	see note 2
2	>>	right shift (note 4)	integer or linteger	see note 2
2	**	power (note 4)	integer, linteger, or real	real
3	+ (unary)	identity	integer, linteger(note 3), or real	same as operand
3	- (unary)	sign inversion	integer, linteger(note 3), or real	same as operand
4	+	addition	integer, linteger(note 3), or real	see note 1
4	+	set union	set	set
4	-	subtraction	integer, linteger(note 3), or real	see note 1
4	-	set difference	set	set
4	OR	logical OR	boolean	boolean
4	!	bit-by-bit OR (note 4)	integer or linteger	see note 2
4	#	bit-by-bit XOR (note 4)	integer or linteger	see note 2
5	=	equality	any set, simple, pointer, or string-type(note 5)	boolean

Prec	Operator	Operation	Operand type(s)	Result type
5	<>	inequality	any set, simple, pointer, or string-type(note 5)	boolean
5	<	less than	any simple or string-type(note 5)	boolean
5	>	greater than	any simple or string-type(note 5)	boolean
5	<=	less or equal	simple or string-type(note 5)	boolean
5	<=	set inclusion (left set included	set in right set)	boolean
5	>=	greater or equal	simple or string-type(note 5)	boolean
5	>=	set inclusion (right set included	set in left set)	boolean
5	IN	member	left operand: any ordinal type, right operand: set of ordinal type	boolean

NOTE 1:
The result type will be real if either operand is real else the result type will be linteger if either operand is linteger else the result type will be integer.

NOTE 2:
The result type will be linteger if either operand is linteger else the result type will be integer.

NOTE 3:
If the ISO compilation mode is selected then the linteger operand(s) are not allowed.

NOTE 4:
If the ISO compilation mode is enabled then this operator is not allowed.

NOTE 5:
When relational operators are used to compare operands of the string-type they denote lexicographic ordering according to the ordering of the ASCII character set.

COMPOUND STATEMENTS

Groups of statements may be collected together and syntactically treated as a single statement. The statements are enclosed with BEGIN and END as shown below.

```
BEGIN
x := y * 44.5;
z := x / 2.0;
END
```

The main body of a Pascal program or subprogram is technically defined as a single "statement", so a compound statement is used to enclose the program statements.

The body of FOR, WHILE, and WITH statements are also syntactically single "statements", so BEGIN and END are used to group multiple statements as one single statament. Here are some examples.

```
WHILE y < 12 DO
   BEGIN
   t := t + d[y];
   y := y + 1;
   END;

FOR i := 1 TO 10 DO
   BEGIN
   a := t[i];
   b := a DIV 5;
   END;

WITH personhead^ DO
   BEGIN
   name := whoami;
   sex := male;
   age := 29;
   END;
```

LOOPING AND CONDITIONAL STATEMENTS

The IF-THEN-ELSE Statement

The IF-THEN statement is Pascal's basic decision making statement. The decision is based on the result of a boolean expression. IF the expression evaluates true then the statement following THEN is executed, otherwise the statement after the ELSE is executed. The ELSE part is optional. Some examples follow:

```
IF x < pi THEN
   writeln('x is smaller than pi');
(next statement)

IF x < pi THEN
   writeln('x is smaller than pi')
ELSE
   writeln('x is larger or equal to pi');
(next statement)
```

Compound statements can be used within IF-THEN statements, and multiple IF-THEN statements can be nested. Here is an example.

```
IF count < required THEN
   writeln('there are too few units')
ELSE
   IF count > too_many THEN
      BEGIN
      hit_max := true;
      writeln('there are too many units');
      END
   ELSE
      writeln('there are the correct number of units');
```

The CASE Statement

The CASE statement uses the value of an expression (that gives an ordinal result) to select one of a numbered list of statements to execute. Here is an example.

```
readln(val);
CASE val OF
1: writeln('value is one');
2: writeln('value is two');
3: writeln('value is three');
4: writeln('value is four');
   END
```

OS-9/68000 Pascal has extended the syntax for case statements in two very useful ways. First, an 'OTHERWISE' selection is provided. If the 'OTHERWISE' selection is not used and a case statement is executed with the value of the selecting expression not appearing in the selection list, a run time case error is caused, and the program is aborted. The 'OTHERWISE' option allows a convenient method for specifying "don't care" or catch-all processing within a case statement. Also for a single case selection, if multiple successive values are part of the same selection, you can use the form 'low..high' as part of the specification. The next example demonstrates the use of both of these enhancements.

```
        CASE nextchar OF
  'a'..'z': processletter;
  '0'..'9': processdigit;
  '$', '_': processspecial;
OTHERWISE : processother;
            END;
```

Constants in the constant list of a case statement can have the form "A..B" which designates the list of values from A through B inclusive. The ordinal value of A must be less than the ordinal value of B. Here is an example.

```
PROGRAM casedemo;
TYPE
   days = (sunday, monday, tuesday, wednesday,
           thursday, friday, saturday);
VAR
   s : days;
BEGIN
s := monday;
CASE s OF
monday..friday:
   writeln(' WEEK DAY ');
OTHERWISE:
   writeln(' WEEKEND ');
   END; { CASE }
END.
```

The REPEAT Statement

The REPEAT statement creates a loop where the exit test is performed at the end of the loop. The test expression must yield a boolean type result. The body of the loop is always executed once. The REPEAT statement is different than other loops in that is does not require a BEGIN..END compound statement because the UNTIL keyword serves to define the end of the loop. The general form for a REPEAT statement is.

```
REPEAT
  (statements);
UNTIL boolean-expression;
```

Here is an example.

```
count := 0;
REPEAT
  writeln('input a number');
  readln(num);
  sum :=sum + num;
  count := count + 1;
UNTIL count = 10;
writeln('the average value is", sum / count);
```

The WHILE-DO STATEMENT

The WHILE-DO statement creates a loop where the exit test is performed at the beginning of the loop. The test expression must yield a boolean type result. If the first test is false, the loop body is never executed. If more than one statement is to be used in the body, a compound statement must be used. The general form for a WHILE-DO statements is.

```
WHILE boolean-expression DO
  (statement);
```

Here is an example.

```
done := false;
count := 0;
WHILE NOT done DO
   BEGIN
   readln(data[count]);
   done := data[count] = 0;
   count := count + 1;
   END;
```

The FOR Statement

This statement creates a loop that uses a specified variable to automatically count iterations of the loop. The two forms of this statement are.

```
FOR variable := start-expr TO end-expr DO statement;

FOR variable := start-expr DOWNTO end-expr DO statement;
```

The FOR control variable must be an ordinal type (char, integer, linteger, etc.), thus, REAL variables cannot be used. The variable must be a simple variable (not an array, structure element, or parameter) declared in the declaration part of the same program or subprogram in which it is used. It can be used within expressions but cannot be altered by program statements in the body of the loop.

When the FOR statement is first executed, the starting and ending expressions are evaluated. Both must be of a compatible type as the control variable. The control variable is set to the result of the starting expression and the loop body is executed. Each time through the loop the control variable is increased by one (or decreased by one if the DOWNTO form was used). The loop terminates when the control variable reaches the value of the end-expression.

Here is an example.

```
PROGRAM fibonacci;
{ print first ten numbers of the fibonacci series }
VAR
   counter, sum : integer;

BEGIN
sum := 0;
FOR counter := 1 TO 10 DO
   BEGIN
   sum := sum + counter; {'counter' used but not changed }
   writeln(sum);
   END;
END.
```

The WITH-DO Statement

This statement provides a convenient way for different elements of the same record to be accessed in a single statement. The syntax of the WITH statement is.

```
WITH <record identifier> {, <record identifier>} DO
   <statement>;
```

Within the statement (which may, of course, be a compound statement), the field names of the record structure are given without the record name. The record name is supplied by the compiler from the name immediately following the WITH keyword. This statement not only saves typing, but it allows the compiler to produce better compiled code by reducing the number of times record addresses must be computed. Here is an example of its use.

```
TYPE
   person = RECORD
            name     : PACKED ARRAY [1..25] OF char;
            age      : integer;
            can_vote : boolean;
            END;

VAR
   index  : integer;
   people : ARRAY [1..3] OF person;

BEGIN
people[1].name := 'Alan Jones                ';
people[2].name := 'Betty Baker               ';
people[3].name := 'David Miller              ';
people[1].age := 14;
people[2].age := 35;
people[3].age := 17;

FOR index := 1 to 3 DO
   WITH people[index] DO
      BEGIN
      can_vote := age >= 18;
      IF can_vote THEN
         writeln(name, ' is old enough to vote');
      END;
END.
```

EXECUTING A PASCAL PROGRAM

The Shell command line form to execute a pascal program is as follows:

```
program-name {memory-size-modifier} {file-redirection} {parameters}
```

memory-size-modifier

This optional Shell modifier in the form of "#nK" or "#n", where 'n' is in kilobytes, will increase the stack area allocated for the program by the indicated amount.

file-redirection

These optional Shell modifiers allow for the redirection of the program's INPUT, OUTPUT, and SYSERR files as follows:

modifier	redirected file
<	INPUT
>	OUTPUT
>>	SYSERR

parameters

All characters which are not Shell modifiers are considered to be parameters to the program. These parameters can be accessed within the program via the predeclared global variables "SYSPARAM" and "SYSPARAMSIZE". These variables are declared as follows:

```
VAR
   sysparamsize : linteger;
   sysparam     : PACKED ARRAY [1..maxint] OF char;
```

At the start of program execution the variable "SYSPARAMSIZE" will be initialized to the number of parameter characters passed to the program. These parameter characters can be accessed by indexing the "SYSPARAM" array from 1 through the value of "SYSPARAMSIZE". Note that because the variable "SYSPARAM" is declared as a large array any index in the range of 1..32767 will be allowed, however any index value greater than the value of "SYSPARAMSIZE" will be accessing memory outside the data area assigned to the program. Also note that the Shell program will pad the parameters passed to the pascal program with a carriage return character ($D), thus all pascal programs initiated by the Shell will have at least one parameter character.

Refer to the OS-9/68000 Operating System User Manual chapter-4 for more information on Shell and it's modifiers.

ABORTIVE ERRORS

If during the execution of a pascal program an error occurs which causes the program to abort, the following error messages will be written to the SYSERR file.

```
Pascal error 064:xxx
064:xxx - text of error message xxx

Procedure #yyyy  Name = zzzzzzzz  Statement #ssssss
Procedure #yyyy  Name = zzzzzzzz  Statement #ssssss
Procedure #yyyy  Name = zzzzzzzz  Statement #ssssss
```

Where "xxx" refers to one of the run-time errors described in Appendix-B. The second line displays the corresponding text of the error message from the file "p.errors". If the error message file "p.errors" can't be opened for any reason then this line won't be displayed.

Lastly a routine calling history will be displayed, with the last routine called (i.e. the routine which had the error) displayed first through the first routine called (i.e. the outer-code block). The value of "yyyy" and the identifing name "zzzzzzzz" will correspond the the values shown in the "Procedure statistics table" report produced by the compiler (see page 2-22). The "Statement #ssss" will appear only if debug code was generated for this routine (see page 2-13). If the statement text is displayed then the value of "ssss" will correspond to the statement currently being executed within this routine.

OS9ERR

OS9ERR is a predeclared global integer variable. When a program terminates the value of this variable will be returned to the calling parent program (in register d1.w). A value of zero, which is the default value at program startup, indicates no error. This variable is useful when the program needs to inform the parent program that it has detected some kind of error. Refer to the "F$Exit" call in the OS-9/68000 Operating System - Technical Manual for more information.

Pascal I/O operations are based on use of standard procedures and functions which are part of the standard library. Since any given computer can have a variety of different input and output hardware devices, the built-in I/O functions must be non-specific and the ISO standard reflects this. Because the OS-9 operating system allows access to all I/O files and devices using essentially the same calling methods, Pascal programs can read or write any system I/O device or file.

Pascal's I/O handling facilities (or lack thereof) and the way they work are commonly cited as a flaw in an otherwise superbly designed language. In order to overcome these traditional limitations, the OS-9/68000 Pascal standard library includes highly intelligent I/O routines and an extensive variety of additional (and non-standard) routines so that Pascal programs can fully utilize the computer's I/O facilities including interactive and random-access I/O.

Pascal files are used to perform all I/O to terminals, printers, disk files, etc., and in general, standard I/O routines will work with any I/O device. This type of operation works well with the device-independent design of the OS-9 operating system, i.e., a Pascal "file" is analogous to an OS-9 "path".

Pascal recognizes two categories of file types, TEXT files ("textfiles") and the structured FILE type. Textfiles are what their name implies, text in the form of characters organized into lines terminated by an end-of-line character. The basic and most commonly used I/O routines operate on textfiles.

In the standard I/O routine descriptions, that are given in the following pages, each procedure's name is followed by a list of expected parameters. You can usually tell which class of I/O the routine applies to by the description of the first identifier in the list as follows.

"text-fileid"	means the routine must be used with textfiles
"struct_fileid"	means the routine must be used with structured file types
"fileid"	means the routine can be used with either textfiles or structured file types.

Routines that use one or more parameters called "external-filenames" directly accept a standard OS-9 file name. The OS-9 filename expression type can be a string constant, a packed character array, a string, or a character constant. File names stored in a character array must be terminated with a character which OS-9 will recognize as the end of a valid path name such as a space or carriage return ($D) character.

PREDEFINED STANDARD I/O FILES

Three file names, "INPUT", "OUTPUT", and "SYSERR", are predefined as textfiles. Output records can be any length, and input records, from the file "INPUT", can be up to 256 characters in length. These three files correspond to the three OS-9 standard I/O paths, and may be redirected when the program is run by means of the OS-9 Shell "<" ">" and ">>" operators.

DEFINITION	OS-9 PATH #	OPEN MODE
INPUT : TEXT[256];	0	Inspection (Input only)
OUTPUT : TEXT[2];	1	Generation (Output only)
SYSERR : TEXT[2];	2	Generation (Output only)

NOTE: The "SYSERR" predefined file is a non-standard extension to OS-9/68000 Pascal.

INTERACTIVE FILES

To make Pascal I/O operations in an interactive environment appear more natural OS-9/68000 Pascal provides a feature called "lazy-io". Normally, whenever a text type file is opened for reading (i.e. RESET), either explicitly or implicitly (as is the case of the predefined file "INPUT"), the file's buffer will be loaded with the first record of the file. For files on disk devices (RBF type files) this is fine but for files associated with interactive devices (SCF type files), like the console terminal, this could prove to be unnatural. It could, for instance, require that the operator supply some data before he could be prompted for that data. What is needed is the ability to defer the actual reading of the data from the interactive device until it is actually needed. This, then, is the essence of "lazy-io".

All uses of the standard procedures "GET", "READ", or "READLN" will cause the file's buffer to be loaded if it is empty. This seems natural and the only difference from "non-lazy-io" processing is that the buffer loading isn't preformed until the actual procedure call is encountered.

All uses of the standard functions "EOF" or "EOLN" will cause the file's buffer to be loaded if it is empty. This is perhaps a less obvious requirement than for the situation just described. "End-of-Line" and "End-of-Line" conditions cannot be determined until the buffer is loaded. You need, therefor, to be careful about testing for either of these conditions before you issue any required prompts for data or you will cause a read to be preformed before the operator knows what is expected.

Finally, references to the current character in a text file via the up-arrow character ("^") as in, Nextchar := input^; , will cause the file's buffer to be loaded if it is empty. The same concern applies here as is described above.

TEXT FILE DECLARATION

When you declare a text type file, via either "FILE OF char" or "TEXT", a line buffer is automatically created by the compiler. This buffer is used to buffer one complete input text line. The use of buffered input is useful for line editing for terminals, such as backspacing or line cancel operations. Since each input text line must fit into this buffer OS-9/68000 Pascal allows the programmer to define the size of this buffer. The declaration "TEXT[buffer_size]" allows a buffer of "buffer_size" characters to be associated with the text file. The "buffer_size" parameter must be an integer constant in the range of 1 thru 8192. Note, this declaration is an OS-9/68000 Pascal extension and if the ISO declarations are used, i.e. "FILE OF char" or "TEXT", a default buffer size of 128 characters will be used. Also, if the text file is only going to be used for writing a buffer size of 1 should be declared since writing to a text file isn't buffered.

WINDOW VARIABLES

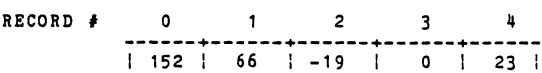
When you declare a file, the compiler automatically creates a file "window variable" of the same component type of the file. This window variable takes on the value of one file component (i.e. record) at a time. You can access only one file record, called the current file record, at any given time. The predefined I/O routines described in this chapter can move the file position, thus changing the value of the file window variable. To denote the file window variable, write the name of the associated file variable and follow it with a carrot ("^") character.

In all Pascal files the first record (i.e. component) is at file position zero (i.e. the records in the file are zero based).

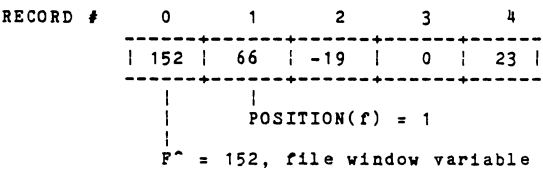
Here is an example of a file declaration and use.

```
PROGRAM filetest(f);
VAR
   f : FILE OF integer;
BEGIN
reset(f);
reposition(f, 3);
get(f);
END.
```

Assume that the file "F" in the current working data directory contains the following information.



After the "reset(f);" statement is executed the file's variables are as follows.



After the "reposition(f, 3);" statement is executed the file's variables are as follows.

```
RECORD #     0       1       2       3       4
           ------+------+------+------+------
           | 152 |  66  | -19  |   0  |  23  |
           ------+------+------+------+------
                                      |
                                      POSITION(f) = 3

             F^ = undefined, file window variable
```

After the "get(f);" statement is executed the file's variables are as follows.

```
RECORD #     0       1       2       3       4
           ------+------+------+------+------
           | 152 |  66  | -19  |   0  |  23  |
           ------+------+------+------+------
                                      |      |
                                      |      POSITION(f) = 4
                                      |
                                      F^ = 0, file window variable
```

RANDOM ACCESS

ISO Pascal only allows sequencial file access. OS-9/68000 Pascal allows a file to be access in a random order.

To read a file record randomly the following steps should be followed. Use the "REPOSITION" procedure to set the file position pointer to the record you want to read. Then issue a "GET" or "READ" call to load that record into the window variable.

To write a record randomly the following steps should be followed. Use the "REPOSITION" procedure to set the file position pointer to the record you want to write. Then issue a "PUT" or "WRITE" call to write the window variable into that record.

Note, both random and sequencial operations are allowed to be mixed.

If you want to use both read and write operations to a file then open that file with the "UPDATE" procedure. Whenever you go from a read to a write operation (or visa versa) a "REPOSITION" call must be issued in between the operations or an error will result. Here is an example.

```
BEGIN
{get record 5}
reposition(f, 5);
get(f);
{update record}
f^.name := 'STEVE';
{put updated record 5}
reposition(f, 5);
put(f);
END
```

APPEND

DECLARATION:

```
PROCEDURE APPEND(fileid {, external_filename, open_mode} );
```

PARAMETERS:

"fileid"	: any file variable.
"external_filename"	: must be an expression who's result type is a string constant, a character constant a string, or a packed array of character.
"open_mode"	: is an integer (16-bit) expression.

FUNCTION:

Opening "fileid" with the APPEND procedure positions the file pointer at the file's end-of-file position and sets the "fileid" access mode to generation (only output operations allowed, put's, write's, writeeof, etc.). This procedure will create the file if it doesn't already exist.

An optional second and third parameter can be specified. If they are not specified then the "external_filename" defaults to the name of the first identifier used to define "fileid" and the "open_mode" defaults to $0303 (READ/WRITE access mode and User READ/WRITE attributes).

If the optional parameters are not specified and "fileid" is an open file then only it's file position and access mode will be changed as described above, otherwise the following operations will be performed.

IF "fileid" is an open file then it will be closed. Then "fileid" will be created using "external_filename" as an OS-9 path name, the eight low order bits of "open_mode" are used to determine the OS-9 file's attribute, and the eight high order bits of "open_mode" are used to determine the OS-9 file's access mode. If this creation fails because the file already exists then "fileid" will be opened using "external_filename" as an OS-9 path name and the eight high order bits of "open_mode" are used to determine the OS-9 file access mode. See I$Create and I$Open in the OS-9/68000 Operating System Technical Manual for more information on OS-9 file attribute and access modes.

EXAMPLES:

```
append(output);
append(output, 'newfile', $0202);
append(filearray[i], concat(dirname, '/filea'), $021b);
```

CLOSE

DECLARATION:

```
PROCEDURE CLOSE(fileid);
```

PARAMETERS:

"fileid" : any file variable.

FUNCTION:

The CLOSE procedure explicity marks the file "fileid" as closed and any system memory allocated when the file was opened is returned. No error is generated if "fileid" isn't open.

In most programs there is no need to use this procedure. Whenever a code block is entered which has declared a file variable, including the outer-code-block, the file's file control block (FCB) is automatically initialized and when the code block is exited the file is automatically closed.

There are two execptions to the above case. If a pointer-variable is declared to point to a file or a record that contains a file then a new(pointer-variable) call will initialize the FCB for the file but the file won't be closed automatically, but can be closed explicity with this procedure. The second case involves the use of the "EXIT" procedure. If you use the "EXIT" procedure to leave one or more code blocks then any files opened within those code blocks are not closed, but they can be closed explicitly by using this procedure.

EXAMPLES:

```
close(output);
close(filearray[i]);
close(pointer^.filevariable);
```

EOF

DECLARATION:

```
FUNCTION EOF{(fileid)} : boolean;
```

PARAMETERS:

"fileid" : any file variable.

FUNCTION:

The EOF function returns a boolean value indicating wheather or not the file pointer is at the end-of-file position, "TRUE" if it is, "FALSE" otherwise.

If the "fileid" parameter is omitted then the predefined file "INPUT" is used.

If the "fileid" file is a text file then "fileid^" will be a space (" ") character whenever the function EOF(fileid) is TRUE.

EXAMPLES:

```
VAR
   b : boolean;

BEGIN
b := eof;
b := eof(filearray[i]);
b := eof(pointer_variable^);
END;
```

EOLN

DECLARATION:

```
FUNCTION EOLN{(text_fileid)} : boolean;
```

PARAMETERS:

"text_fileid" : must be a text file variable.

FUNCTION:

The EOLN function returns a boolean value indicating wheather or not the file pointer is at an end-of-line position, "TRUE" if it is, "FALSE" otherwise.

If the "text_fileid" parameter is omitted then the predefined file "INPUT" is used.

The "text_fileid" parameter must be a text file and "text_fileid^" will be a space (" ") character when the function EOLN(text_fileid) is TRUE.

It is an error to call this function if EOF(text_fileid) is TRUE.

EXAMPLES:

```
VAR
   b : boolean;

BEGIN
b := eoln;
b := eoln(filearray[i]);
b := eoln(pointer_variable^);
END;
```

FILESIZE

DECLARATION:

```
FUNCTION FILESIZE(fileid) : linteger;
```

PARAMETERS:

"fileid" : any file variable.

FUNCTION:

If "fileid" is a text file then this functions returns a linteger value giving the current number of characters in the file.

If "fileid" is a structured file then this function returns a linteger value giving the current number of whole records in the file. If the last record in the file is a partial record then a short-record error will be generated.

If "fileid" isn't a file on an "RBF" type device (disk) then this function returns a value of zero.

EXAMPLES:

```
VAR
   p : linteger;

BEGIN
p := filesize(output);
p := filesize(filearray[i]);
p := filesize(pointer^.filefield);
END;
```

GET

DECLARATION:

```
PROCEDURE GET(fileid);
```

PARAMETERS:

"fileid" : any file variable.

FUNCTION:

If "fileid" is a text file then this procedure causes the character at the current file position to be read into the "fileid" window variable and then increments the file position. This window variable is accesed by the reference "fileid^". If end-of-line or end-of-file is detected during the read then a space (" ") character is loaded into the window variable.

If "fileid" is a structured file then this procedure causes the record at the current file position to be read into the "fileid" window variable and then increments the file position. This window variable is accessed by the reference "fileid^". If the record read is a partial record then a short-record error will be generated and the size of the partial record can be extracted with the "LINELENGTH" function. If end-of-file is detected during the read then the "fileid" window variable contents is undefined.

It is an error if EOF(fileid) is TRUE when this procedure is called.

EXAMPLES:

```
get(input);
get(filearray[i]);
get(structured_file);
```

GETCHAR

DECLARATION:

```
FUNCTION GETCHAR(text_fileid) : char;
```

PARAMETERS:

"text_fileid" : must be a text file variable.

FUNCTION:

The GETCHAR function returns a character value which is the next available character from the text file "text_fileid". The character returned is always the actual character read. For instance, if the end-of-line character is read then the end-of-line character will be returned.

If the file "text_fileid" is on an interactive (SCF) device and it's window variable is undefined (i.e. the file buffer is empty) then the statement:

```
ch := getchar(text_fileid)
```

is equivalent to:

```
BEGIN get(text_fileid); ch := text_fileid^; END
```

and the transfer is unbuffered (i.e. the file buffer remains empty). If a character isn't currently available from the device (i.e. no key has been struck) then this function waits until it is available. This function is normally used in conjunction with the "IOREADY" function.

If the file "text_fileid" isn't on an interactive (SCF) device or it's window variable is defined (i.e. the file buffer was loaded from a previous "GET" or "READ" call) then the statement:

```
ch := getchar(text_fileid)
```

is equivalent to:

```
read(text_fileid, ch
```

GETCHAR
(continued)

If you want to be sure that the "text_fileid" file's buffer is empty, so that this function reads the next character unbuffered, then the following statement should be issued before the first call to this function.

```
readln(text_fileid); {flush current buffer and don't load
                      with next record if SCF type file}
```

EXAMPLES:

```
VAR
   ch : char;

BEGIN
ch := getchar(input);
ch := getchar(filearray[i]);
ch := getchar(pointer^.filefield);
END;
```

EXAMPLE PROGRAM:

```
PROGRAM getchar(input, output);
VAR
   count       : linteger;
   ch          : char;
BEGIN
REPEAT
   {zero counter}
   count := 0;

   {increment counter while no character available}
   WHILE NOT ioready(input) DO
      count := succ(count);

   { get character }
   ch := getchar(input);

   {output results}
   writeln(output, '   ', ord(ch), count);

{loop until <ESC> character entered}
UNTIL ch = chr($1b {ESC character});
END.
```

GETINFO

DECLARATION:

```
PROCEDURE GETINFO(fileid, buffer);
```

PARAMETERS:

"fileid"	: any file variable.
"buffer"	: any variable with a size of 128.

FUNCTION:

This procedure reads the 128 byte option section, of the path descriptor for file "fileid", into the variable "buffer". The variable "buffer" can be of any type (usually it is of record type) as long as it's size is 128 bytes in length. Refer to the OS-9/68000 Operating System Technical Manual for information on path descriptor option area definitions.

EXAMPLE PROGRAM:

```
PROGRAM getinfo(input);
VAR
   buffer    : RECORD  { Partial SCF definition }
               file_class    : (scf, rbf, pipe, sbf, net);
               pd_upc        : boolean;
               pd_bso        : boolean;
               pd_dlo        : boolean;
               pd_eko        : boolean;
               pd_alf        : boolean;
               pd_nul        : char;
               pd_pau        : boolean;
               { fill record to 128 byte size }
               { be careful of structure sizes and alignment }
               filler        : PACKED ARRAY [1..120] OF char;
               END;
BEGIN { mainline }
{ get option area for INPUT file }
getinfo(input, buffer);
IF buffer.file_class = scf THEN
   BEGIN { only update if SCF type file }
   { set echo flag to false (0 = no echo) }
   buffer.pd_eko := false;
   { put updated option area for INPUT file }
   putinfo(input, buffer);
   END;
{ any get/read to file INPUT now won't echo characters }
END.
```

INTERACTIVE

DECLARATION:

```
FUNCTION INTERACTIVE(text_fileid) : boolean;
```

PARAMETERS:

"text_fileid" : must be a text file variable.

FUNCTION:

This function returns a boolean value indicating wheather the file "text_fileid" is associated with an interactive (SCF) type device, "TRUE" if it is, "FALSE" otherwise. Refer to page 5-2 for information on interactive files.

EXAMPLES:

```
VAR
   b : boolean;

BEGIN
b := interactive(input);
b := interactive(filearray[i]);
b := interactive(pointer^.filefield);
END;
```

IOABORT

DECLARATION:

```
PROCEDURE IOABORT(fileid, value);
```

PARAMETERS:

"fileid"	: any file variable.
"value"	: a boolean expression.

FUNCTION:

This procedure allows the enabling (if "value" is TRUE) or disabling (if "value" is FALSE) of the IO abort flag for the file "fileid".

If the IO abort flag is disabled for the file "fileid", and an I/O error occurs on the file "fileid", then the program won't abort and the I/O error values will be saved. After each I/O call on the file "fileid", with it's IO abort flag disabled, you should use the function "IORESULT" to check for any errors. Also, if multiple I/O errors occur on the file "fileid" before the function "IORESULT" is called then only the first error values are saved.

The IO abort flag is by default enabled for all files declared in the program and can be only be disabled by using this procedure.

EXAMPLES:

```
ioabort(input, false);
ioabort(filearray[i], true);
ioabort(pointer^.filefield, false);
```

IOABORT (continued)

EXAMPLE PROGRAM:

```
PROGRAM ioabort(input, syserr);
VAR
   pascal_error : integer;
   os9_error    : integer;
BEGIN
{ disable IO abort flag on file "INPUT" }
ioabort(input, false);

{ open "INPUT" file }
reset(input, 'new_file', $0100);

{ check for errors }
pascal_error := ioresult(input);
IF pascal_error <> 0 THEN
   BEGIN  { got an error }
   { report error }
   sysreport(syserr, pascal_error);
   writeln(syserr);

   { check for OS-9 error }
   os9_error := iosyserr(input);
   IF os9_error <> 0 THEN
      { report os9 error }
      writeln(syserr, 'OS-9 Error #', os9_error:1);
    END;
{ enable IO abort flag on file "INPUT" }
ioabort(input, true);

{ any I/O errors on file "INPUT" now will abort the program }
END.
```

IOREADY

DECLARATION:

```
FUNCTION IOREADY(text_fileid) : boolean;
```

PARAMETERS:

"text_fileid" : must be a text file variable.

FUNCTION:

This function returns a boolean value indicating weather a character is available from the file "text_fileid", "TRUE" if it is, "FALSE" otherwise.

If the file "text_fileid" is on an interactive (SCF) device and it's window variable is undefined (i.e. the file buffer is empty) then a value of "TRUE" is returned if at least one character is in the OS-9 input buffer for the file "text_fileid" otherwise a value of "FALSE" is returned.

If the file "text_fileid" isn't on an interactive (SCF) device or it's window variable is defined (i.e. a previous "GET" or "READ" call loaded the file buffer) then the statement:

```
b := ioready(text_fileid)
```

is equivalent to:

```
b := NOT(eof(text_fileid)).
```

This function is normally used in conjunction with the "GETCHAR" function.

EXAMPLES:

```
VAR
   b : boolean;

BEGIN
b := ioready(input);
b := ioready(filearray[i]);
b := ioready(pointer^.filefield);
END;
```

IOREADY (continued)

EXAMPLE PROGRAM:

```
PROGRAM ioready(input, output);
VAR
   count        : linteger;
   ch           : char;
BEGIN
REPEAT
   {zero counter}
   count := 0;

   {increment counter while no character available}
   WHILE NOT ioready(input) DO
      count := succ(count);

   { get character }
   ch := getchar(input);

   {output results}
   writeln(output, '   ', ord(ch), count);

{loop until <ESC> character entered}
UNTIL ch = chr($1b {ESC character});
END.
```

IORESULT

DECLARATION:

```
FUNCTION IORESULT(fileid) : integer;
```

PARAMETERS:

"fileid" : any file variable.

FUNCTION:

This function returns the last Pascal error posted to the file "fileid" and then zeros the Pascal error value. A value of zero returned indicates that no I/O errors have occured on the file "fileid" since the last call to this function.

This function is normally used in conjunction with the "IOABORT" procedure.

EXAMPLE PROGRAM:

```
PROGRAM ioresult(input, syserr);
VAR
   pascal_error : integer;
   os9_error    : integer;
BEGIN
{ disable IO abort flag on file "INPUT" }
ioabort(input, false);

{ open "INPUT" file }
reset(input, 'new_file', $0100);

{ check for errors }
pascal_error := ioresult(input);
IF pascal_error <> 0 THEN
   BEGIN  { got an error }
   { report error }
   sysreport(syserr, pascal_error);
   writeln(syserr);

   { check for OS-9 error }
   os9_error := iosyserr(input);
   IF os9_error <> 0 THEN
      { report os9 error }
```

```
      writeln(syserr, 'OS-9 Error #', os9_error:1);
   END;
{ enable IO abort flag on file "INPUT" }
ioabort(input, true);
{ any I/O errors on file "INPUT" now will abort the program }
END.
```

IOSYSERR

DECLARATION:

```
FUNCTION IOSYSERR(fileid) : integer;
```

PARAMETERS:

"fileid" : any file variable.

FUNCTION:

This function returns the last OS-9 error posted to the file "fileid" and then zeros the OS-9 error value. A value of zero returned indicates that no OS-9 errors have occured on the file "fileid" since the last call to this function.

This function is normally used in conjunction with the "IOABORT" procedure.

EXAMPLE PROGRAM:

```
PROGRAM ioresult(input, syserr);
VAR
   pascal_error : integer;
   os9_error    : integer;
BEGIN
{ disable IO abort flag on file "INPUT" }
ioabort(input, false);

{ open "INPUT" file }
reset(input, 'new_file', $0100);

{ check for errors }
pascal_error := ioresult(input);
IF pascal_error <> 0 THEN
   BEGIN  { got an error }
   { report error }
   sysreport(syserr, pascal_error);
   writeln(syserr);

   { check for OS-9 error }
   os9_error := iosyserr(input);
   IF os9_error <> 0 THEN
      { report os9 error }
```

```
      writeln(syserr, 'OS-9 Error #', os9_error:1);
   END;
{ enable IO abort flag on file "INPUT" }
ioabort(input, true);
{ any I/O errors on file "INPUT" now will abort the program }
END.
```

LINELENGTH

DECLARATION:

```
FUNCTION LINELENGTH(fileid) : integer;
```

PARAMETERS:

"fileid" : any file variable.

FUNCTION:

This function returns an integer length value, in bytes, of the last record read from, or the next record to be written to the file "fileid".

If the file "fileid" is a structured file then the following description applies. After a "GET" or "READ" call is issued, against the file "fileid", this function will return the actual length of the record read. If a short-record error occurs then this value represents the actual number of bytes read. Before a "PUT" or "WRITE" call is issued, against the file "fileid", this function will return the length of the current record to be written. The procedure "SHORTIO" is used to set this write length to a value other than the file's buffer length.

If the file "fileid" is a text file then the following description applies. After a "GET", "READ", or "READLN" call is issued which loads the file buffer, this function will return the actual length of the record read into the buffer. This value doesn't include the terminating end-of-line character. For example, if this text line "This is a linelength test" is loaded into the file buffer, via a "GET", "READ", or "READLN" call, then this function will return a value of 25. A "WRITELN", "PAGE", or "OVERPRINT" call zero's the linelength value associated with the file "fileid". Otherwise any "PUT" or "WRITE" call will increase the linelength value by the length of the character sequence written. This function will then return the line length (i.e. column position) of the current text record.

EXAMPLES:

```
VAR
   len, column, buflen : integer;

BEGIN
len := linelength(input);
column := linelength(output);
buflen := linelength(strucfile);
END;
```

LINELENGTH (continued)

EXAMPLE PROGRAM:

```
PROGRAM textdemo(input, output);
VAR
   i         : integer;
BEGIN
{check for EOF and load buffer if empty}
WHILE NOT eof(input) DO
   BEGIN
   write(output, 'Line length = ', linelength(input):1);
   {get current column position of "OUTPUT" record}
   i := linelength(output);
   {tab to column 20}
   IF i < 20 THEN
      write(output, ' ':20-i);
   {copy "INPUT" record to "OUTPUT" record}
   WHILE NOT eoln(input) DO
      BEGIN
      write(output, input^);
      get(input);
      END;
   writeln(output);  {terminates current output record}
   readln(input);    {terminates current input record and
                      loads next record}
   END;
END.
```

OPENED

DECLARATION:

```
FUNCTION OPENED(fileid) : boolean;
```

PARAMETERS:

"fileid" : any file variable.

FUNCTION:

This function returns a boolean value indicating wheather the file "fileid" is currently open, "TRUE" if it is, "FALSE" otherwise.

Note that the three predefined files "INPUT", "OUTPUT", and "SYSERR" are always implicitly opened at the beginning of program execution, all other files must be explicitly opened by using the procedures "APPEND", "RESET", "REWRITE", or "UPDATE".

EXAMPLES:

```
VAR
   b : boolean;

BEGIN
b := opened(input);
b := opened(filearray[i]);
b := opened(pointer^.filefield);
END;
```

OVERPRINT

DECLARATION:

```
PROCEDURE OVERPRINT[(text_fileid)];
```

PARAMETERS:

"text_fileid" : must be a text file variable.

FUNCTION:

This procedure provides an overprint capability for text files. If the optional parameter "text_fileid" is omitted then the file "OUTPUT" will be used by default. When this procedure is called it writes a carriage return ($D) character to the file "text_fileid", this causes the next print line to overprint the previous line. This procedure is useful for boldface printing and for underlining words.

EXAMPLE PROGRAM:

```
PROGRAM overprint(input, output);
VAR
   instr       : string[255];
BEGIN
WHILE NOT eof(input) DO
   BEGIN
   {read a line from the file "INPUT"}
   readln(input, instr);
   {write the line to the file "OUTPUT"}
   write(output, instr);
   {overprint line}
   overprint(output);
   {write the line to the file "OUTPUT" again}
   writeln(output, instr);
   END;
END.
```

PAGE

DECLARATION:

```
PROCEDURE PAGE{(text_fileid)};
```

PARAMETERS:

"text_fileid" : must be a text file variable.

FUNCTION:

This procedure provides a page eject capability for text files. If the optional parameter "text_fileid" is omitted then the file "OUTPUT" will be used by default. When this procedure is called it writes a form feed ($C) character to the file "text_fileid", this causes the next print line to print on the top line of the next page. This procedure is somewhat hardware dependent in that the device associated with the file "text_fileid" must recognize the form feed ($C) character.

EXAMPLES:

```
page;
page(syserr);
page(filearray[i]);
page(pointer^.rkdarray[i].filefield);
```

POSITION

DECLARATION:

```
FUNCTION POSITION(fileid) : linteger;
```

PARAMETERS:

"fileid" : any file variable.

FUNCTION:

If the file "fileid" is a structured file then this function returns the current file record number (record number zero is the first record in the file). This record number refers to the next record to be read or written, not the beginning of the last record read or written.

If the file "fileid" is a text file then this function returns the current file byte position (byte position zero is the first byte in the file). This byte position refers to the next buffered record to be read, or the next record to be written. Note that because Pascal buffers input records the file byte position will only be at the beginning of text line records. However since output text records are not buffered the file byte position will be the actual byte position in the file, even if a partial text record is being written (i.e a write call).

This function is useful, with the procedure "REPOSITION", for random access I/O.

EXAMPLES:

```
VAR
   p : linteger;

BEGIN
p := position(output);
p := position(filearray[i]);
p := position(pointer^.filefield);
END;
```

POSITION (continued)

EXAMPLE PROGRAM:

```
PROGRAM position(output);
VAR
   saveloc      : linteger;
   length       : integer;
BEGIN
{open output file}
rewrite(output, 'testfile', $020b);
{write partial text record}
write(output, 'This line is ');
{get file position}
saveloc := position(output);
{write partial text record}
write(output, 'xx characters long.');
{get text record length}
length := linelength(output);
{close text record}
writeln(output);
{seek to saved file position}
reposition(output, saveloc);
{write line length into text record}
write(output, length:2);
END.
```

PUT

DECLARATION:

```
PROCEDURE PUT(fileid);
```

PARAMETERS:

"fileid" : any file variable.

FUNCTION:

If "fileid" is a text file then this procedure causes the character in the "fileid" window variable to be written at the current file position and then increments the file position. This window variable is accesed by the reference "fileid^".

If "fileid" is a structured file then this procedure causes the record in the "fileid" window variable to be written at the current file position and then increments the file position. This window variable is accessed by the reference "fileid^".

It is an error if EOF(fileid) is FALSE when this procedure is called.

EXAMPLES:

```
put(output);
put(structured_file);
put(pointer^.filefield);
```

PUTINFO

DECLARATION:

```
PROCEDURE PUTINFO(fileid, buffer);
```

PARAMETERS:

```
"fileid"          : any file variable.
"buffer"          : any variable with a size of 128.
```

FUNCTION:

This procedure writes the 128 byte option section, of the path descriptor for file "fileid", from the variable "buffer". The variable "buffer" can be of any type (usually it is of record type) as long as it's size is 128 bytes in length. Refer to the OS-9/68000 Operating System Technical Manual for information on path descriptor option area definitions.

EXAMPLE PROGRAM:

```
PROGRAM putinfo(output);
VAR
   buffer   : RECORD  { Partial SCF definition }
              file_class   : (scf, rbf, pipe, sbf, net);
              pd_upc       : boolean;
              pd_bsc       : boolean;
              pd_dlo       : boolean;
              pd_eko       : boolean;
              pd_alf       : boolean;
              pd_nul       : char;
              pd_pau       : boolean;
              { fill record to 128 byte size }
              { be careful of structure sizes and alignment }
              filler       : PACKED ARRAY [1..120] OF char;
              END;
BEGIN { mainline }
{ get option area for OUTPUT file }
getinfo(output, buffer);
IF buffer.file_class = scf THEN
   BEGIN { only update if SCF type file }
   { set pause flag to false (0 = no end of page pause) }
   buffer.pd_pau := false;
   { put updated option area for OUTPUT file }
   putinfo(output, buffer);
   END;
{ any put/write to file OUTPUT now won't pause at end of page }
END.
```

READ

DECLARATION:

```
PROCEDURE READ({fileid,} variable1 {,variable2 .. {,variableN}});
```

PARAMETERS:

"fileid"	: any file variable.
"variable1", "variable2", ... "variableN"	: If the file "fileid" is a text file then these variables can be of type char (or subrange of char), integer (or subrange of integer), linteger, real, or string. Otherwise if the file is a structured file then these variables must be of a type conformable to the file's window variable.

FUNCTION:

If the "fileid" parameter is omitted then the file "INPUT" will be used.

STRUCTURED FILE

If the file "fileid" is a structured file then the statement:

```
read(fileid, variable1, variable2, ... variableN)
```

is equivalent to:

```
BEGIN
variable1 := fileid^; get(fileid);
variable2 := fileid^; get(fileid);
...
variableN := fileid^; get(fileid);
END
```

READ (continued)

TEXT FILE

If the file "fileid" is a text file then the following descriptions apply.

CHAR TYPE

If the variable being read is of type CHAR then the character in the file's window variable is assigned to the variable and the window position is advanced. This is equivalent to:

```
BEGIN
variable := fileid^;
get(fileid);
END
```

INTEGER or LINTEGER TYPE

If the variable being read is of type INTEGER or LINTEGER then a sequence of characters is read, ignoring preceeding end-of-lines and spaces. The first non-space character sequence must be a valid signed integer number. That is it may begin with an optional plus ("+") or minus ("-") character and then follow with one or more digit ("0".."9") characters. The first non-digit character terminates the read. The following text strings could be validly read as integers:

```
 +23
-0
127
-32768
    17
14apples
```

The last text string "14apples" would read as the integer "14" and the file's window variable would contain the character "a" from "apples".

Integers must be in the range of -32768 to +32767 and lintegers must be in the range of -2,147,483,648 to +2,147,483,647.

READ (continued)

REAL TYPE

If the variable being read is of type real then a sequence of characters is read, ignoring preceeding end-of-lines and spaces. The first non-space character sequence must be a valid signed real number. That is it may begin with an optional plus ("+") or minus ("-") character followed by a sequence of one or more digit characters ("0".."9"). This sequence can optionally be followed by a decimal point (".") character and a digit sequence of one or more digits. Finally, this sequence can optionally be followed by an exponent character ("e" or "E"), an optional plus ("+") or minus ("-") character, and a digit sequence of one or more digits. The first non-real character found terminates the read. The following text strings can validly be read as real numbers.

```
 +23
-0
12.37
0.01
13.0
1e27
1E+27
+1E-27
0.31415926534e1
14e4apples
```

STRING TYPE

The reading of a string variable causes a sequence of characters, from the current file window variable position to the end-of-line character, to be assigned to the string variable. If the length of this sequence is greater than the defined maximum length of the string variable then a run-time error will occur. Note that if the current file window variable variable is at the end-of-line position, EOLN(fileid) is TRUE, the string sequence will have a length of zero and the string variable will be assigned a string sequence of zero length (i.e. a null string). After the sequence of characters has been assigned to the string variable the file's window variable will be at the end-of-line position, EOLN(fileid) is TRUE. The following examples show the results of reading various text strings.

TEXT-STRING	STRING-VARIABLE	LENGTH-OF-STRING
"A<end-of-line>"	"A"	1
"Apples<end-of-line>"	"Apples"	6

READ (continued)

TEXT-STRING	STRING-VARIABLE	LENGTH-OF-STRING
"13<end-of-line>"	"13"	2
"<end-of-line>"	""	0

READLN

DECLARATION:

```
PROCEDURE READLN{({{text_fileid} {,variable1 {,variable2 ...
                                  {,variableN}}}});
```

PARAMETERS:

"text_fileid" : must be a text file variable.

"variable1",
"variable2",
...
"variableN" : These optional variables can be of type char (or subrange of char), integer (or subrange of integer), linteger, real or string.

FUNCTION:

If the "text_fileid" parameter is omitted then the file "INPUT" will be used.

This procedure reads sequences of characters from the file "text_fileid" and assigns them to the variable(s). See the procedure "READ" for a description of this process. Then the file's window variable is advanced to the first character after the end-of-line character (i.e. the current buffer record is cleared and the next record is read into the buffer). If the file "text_fileid" is an interactive file then the reading of the next record is defered, refer to page 5-2 for a description of interactive files.

EXAMPLES:

```
readln;
readln(strg);
readln(input, intg, lintg, ch, reel, strg);
readln(filearray[i], strg);
```

REPOSITION

DECLARATION:

```
PROCEDURE REPOSITION(fileid, record_position);
```

PARAMETERS:

```
"fileid"           : any file variable.
"record_position"  : a linteger expression.
```

FUNCTION:

This procedure sets the current file position pointer, for the file "fileid", to the "record_position" value. If the file "fileid" is open in the Inspection or Update modes then the file's window variable is marked as empty. Thus the next "GET", "PUT", "READ", or "WRITE" call, or any call which causes the file buffer to be loaded, will occur at this new file position.

This procedure is used for Random-access I/O. Refer to page 5-3 for information on random-access files.

EXAMPLES:

```
reposition(input, 0);
reposition(struct_file, old_record);
reposition(filearray[i], 100000);
```

RESET

DECLARATION:

```
PROCEDURE RESET(fileid {, external_filename, open_mode} );
```

PARAMETERS:

"fileid"	: any file variable.
"external_filename"	: must be an expression who's result type is a string constant, a character constant a string, or a packed array of character.
"open_mode"	: is an integer (16-bit) expression.

FUNCTION:

Opening "fileid" with the RESET procedure positions the file pointer at the file's beginning-of-file position (the file is rewound) then the file's buffer is loaded with the first record if the file isn't an interactive file. RESET also sets the "fileid" access mode to inspection (only input operations allowed, get's, read's, etc.). This procedure won't create the file if it doesn't already exist.

An optional second and third parameter can be specified. If they are not specified then the "external_filename" defaults to the name of the first identifier used to define "fileid" and the "open_mode" defaults to $0300 (READ/WRITE access mode).

If the optional parameters are not specified and "fileid" is an open file then only it's file position and access mode will be changed as described above, otherwise the following operations will be performed.

IF "fileid" is an open file then it will be closed. Then "fileid" will be opened using "external_filename" as an OS-9 path name and the eight high order bits of "open_mode" are used to determine the OS-9 file access mode. See I$Create and I$Open in the OS-9/68000 Operating System Technical Manual for more information on OS-9 file attribute and access modes.

EXAMPLES:

```
reset(input);
reset(input, 'newfile', $0100);
reset(directoryfile, '/dd/cmds', $8100);
```

REWRITE

DECLARATION:

```
PROCEDURE REWRITE(fileid {, external_filename, open_mode} );
```

PARAMETERS:

"fileid"	: any file variable.
"external_filename"	: must be an expression who's result type is a string constant, a character constant a string, or a packed array of character.
"open_mode"	: is an integer (16-bit) expression.

FUNCTION:

Opening "fileid" with the REWRITE procedure positions the file pointer at the file's beginning-of-file position, all data in the file is erased (it becomes empty), and the "fileid" access mode is set to generation (only output operations allowed, put's, write's, writeeof, etc.). This procedure will create the file if it doesn't already exist.

An optional second and third parameter can be specified. If they are not specified then the "external_filename" defaults to the name of the first identifier used to define "fileid" and the "open_mode" defaults to $0303 (READ/WRITE access mode and User READ/WRITE attributes).

If the optional parameters are not specified and "fileid" is an open file then only it's file position and access mode will be changed as described above, otherwise the following operations will be performed.

IF "fileid" is an open file then it will be closed. Then "fileid" will be created using "external_filename" as an OS-9 path name, the eight low order bits of "open_mode" are used to determine the OS-9 file's attribute, and the eight high order bits of "open_mode" are used to determine the OS-9 file's access mode. If this creation fails because the file already exists then "fileid" will be opened using "external_filename" as an OS-9 path name and the eight high order bits of "open_mode" are used to determine the OS-9 file access mode. See I$Create and I$Open in the OS-9/68000 Operating System Technical Manual for more information on OS-9 file attribute and access modes.

REWRITE (continued)

EXAMPLES:

```
rewrite(output);
rewrite(output, 'newfile', $0202);
rewrite(filearray[i], concat(dirname, '/filea'), $031b);
```

SEEKEOF

DECLARATION:

```
PROCEDURE SEEKEOF(fileid);
```

PARAMETERS:

"fileid" : any file variable.

FUNCTION:

This procedure sets the file position pointer, for the file "fileid", at the end-of-file position. The next "PUT" or "WRITE" call will cause the file's window variable to be append to the file. However, if you attempt a "READ" or "GET" call the end-of-file contition will be returned.

If the file "fileid" is a structured file and the last record in the file is a partial record then the shortio error will be generated.

EXAMPLES:

```
seekeof(output);
seekeof(filearray[i]);
seekeof(pointer^.filefield);
```

SHORTIO

DECLARATION:

```
PROCEDURE SHORTIO(struct_fileid, record_length);
```

PARAMETERS:

```
"struct_fileid"      : must be a structured file variable.
"record_length"      : an integer expression.
```

FUNCTION:

This procedure allows reading or writing of records which are shorter than the declared record length for the file "struct_fileid". The "record_length" parameter, which must be in the range of zero thru the declared file's record length, is placed into the linelength variable in the file-control-block (FCB) for the file "struct_fileid", and is used as the record length for the next "READ", "GET", "WRITE", or "PUT" to the file. After the next I/O operation, which affects the file's position, takes place the linelength variable in the file's FCB will be set back to the declared record size of the file's window variable. Thus only the first read or write operation after the call to this procedure will use the reduced record length.

EXAMPLE PROGRAM:

```
program shortio(input, output, diskin, diskout);
const
  shortrecord       = 22{p$shtrec};
  filerecordsize    = $400;
type
  filerecord        = packed array[1..filerecordsize] of char;
  filetype          = file of filerecord;
var
  diskin, diskout   : filetype;

function accept(prompt : string) : string;
  var
    answer          : string;
  begin
  repeat
    if interactive(input) then
      write(output, prompt);
    if not eof(input) then
      readln(input, answer)
    else
```

SHORTIO (continued)

```
    exit(program);
until length(answer) <> 0;
accept := answer;
end;


function filestat(var f : filetype) : boolean;
  var
    error           : integer;
    ioerr           : integer;
  begin
  error := ioresult(f);
  if error <> 0 then
    if error <> shortrecord then
      begin
      writeln(output, 'Pascal error #', error:1,
                      ' on the source file');
      sysreport(output, error);
      if linelength(output) <> 0 then
        writeln(output);
      ioerr := iosyserr(f);
      if ioerr <> 0 then
        writeln(output, 'OS9 error #', ioerr:1);
      filestat := false;
      end
    else
      filestat := true
  else
    filestat := not eof(f);
  end;

begin
{disable error abort on diskin file}
ioabort(diskin, false);
{open source file, also get first source record}
reset(diskin, accept('Enter source file pathname: '), $0100);
{open result file}
rewrite(diskout, accept('Enter result pathname: '), $020b);
{check for errors on diskin file}
while filestat(diskin) do
  begin
  {move source record to result record}
  diskout^ := diskin^;
  {set write length to read length of source}
  shortio(diskout, linelength(diskin));
  {write the record}
  put(diskout);
```

SHORTIO (continued)

```
  {reset the record length for source file to the default}
  shortio(diskin, sizeof(filerecord));
  {get next source record}
  get(diskin);
  end;
close(diskin);
close(diskout);
end.
```

SYSREPORT

DECLARATION:

```
PROCEDURE SYSREPORT(text_fileid, error_number)
```

PARAMETERS:

```
"text_fileid"       : must be a text file variable.
"error_number"      : an integer expression.
```

FUNCTION:

This procedure allows the programmer to write the error message text for the "error_number" value to the file "text_fileid".

When this procedure is called, the error file "p.errors" in the current execution directory is opened if it wasn't opened by a previous call to this procedure, and the "error_number"th text record is retrieved and written to the file "text_fileid". Note for this procedure the first record number is record number one. If the error file "p.errors" can't be opened or an error occurs while trying to retrieve the text record then nothing is written to the file "text_fileid".

EXAMPLE PROGRAM:

```
PROGRAM sysreport(output);
VAR
   i          : integer;
BEGIN
{disasble math errors}
mathabort := false;
mathresult := 0;
{generate math overflow error}
i := maxint;
i := i * i;
{check for math error}
IF mathresult <> 0 THEN
   BEGIN
   {display math error}
   writeln(output, !Math error #', mathresult:1);
   {display error text}
   sysreport(output, mathresult);
   {check for message written}
   IF linelength(output) <> 0 THEN
```

SYSREPORT (continued)

```
      writeln(output);
   END;
END.
```

UPDATE

DECLARATION:

```
PROCEDURE UPDATE(fileid {, external_filename, open_mode} );
```

PARAMETERS:

"fileid"	: any file variable.
"external_filename"	: must be an expression who's result type is a string constant, a character constant a string, or a packed array of character.
"open_mode"	: is an integer (16-bit) expression.

FUNCTION:

Opening "fileid" with the UPDATE procedure positions the file pointer at the file's beginning-of-file position and sets the "fileid" access mode to update (both input and output operations allowed, get's, put's, read's, write's, writeeof, etc.). This procedure will create the file if it doesn't already exist.

An optional second and third parameter can be specified. If they are not specified then the "external_filename" defaults to the name of the first identifier used to define "fileid" and the "open_mode" defaults to $0303 (READ/WRITE access mode and User READ/WRITE attributes).

If the optional parameters are not specified and "fileid" is an open file then only it's file position and access mode will be changed as described above, otherwise the following operations will be performed.

IF "fileid" is an open file then it will be closed. Then "fileid" will be created using "external_filename" as an OS-9 path name, the eight low order bits of "open_mode" are used to determine the OS-9 file's attribute, and the eight high order bits of "open_mode" are used to determine the OS-9 file's access mode. If this creation fails because the file already exists then "fileid" will be opened using "external_filename" as an OS-9 path name and the eight high order bits of "open_mode" are used to determine the OS-9 file access mode. See I$Create and I$Open in the OS-9/68000 Operating System Technical Manual for more information on OS-9 file attribute and access modes.

UPDATE (continued)

EXAMPLES:

```
update(input, 'newfile', $0303);
update(filearray[i], concat(dirname, '/filea'), $031b);
update(disksector, '/dd@', $0303);
```

WRITE

DECLARATION:

```
WRITE({fileid,} expression1{:fieldwidth{:fracdigits
                {,expression2{:fieldwidth{:fracdigits}} ...
                {,expressionN{:fieldwidth{:fracdigits}}} });
```

PARAMETERS:

"fileid"	: any file variable.
"expression1", "expression2", ... "expressionN"	: If the file "fileid" is a text file then these expressions can be of type BOOLEAN, CHAR, INTEGER, LINTEGER, REAL, STRING, or PACKED ARRAY OF CHAR. Otherwise, if the file "fileid" is a structured file then these expressions must be of a type conformable to the file's window variable.
"fieldwidth"	: If the file "fileid" is a text file then this optional integer expression may be used to specify the output field width.
"fracdigits"	: This optional integer expression may only be specified when writing a real number to a text file. This expression specifies the number of fractional decimal places in the output string.

FUNCTION:

If the "fileid" parameter is omitted then the file "OUTPUT" will be used.

WRITE (continued)

STRUCTURED FILE

If the file "fileid" is a structured file then the statement:

```
write(fileid, expression1, expression2, ... expressionN)
```

is equivalent to:

```
BEGIN
fileid^ := expression1; put(fileid);
fileid^ := expression2; put(fileid);
...
fileid^ := expressionN; put(fileid);
END
```

TEXT FILE

If the file "fileid" is a text file then the following descriptions apply.

INTEGER or LINTEGER TYPE

If the expression is of type INTEGER or LINTEGER and the "fieldwidth" parameter is omitted then a default value of 10 will be used for "fieldwidth". The "fieldwidth" value must be greater than or equal to zero.

A sequence of characters will be written to the file "fileid" as follows. If the "fieldwidth" value is greater than the number of characters required to represent the expression then a sufficient number of spaces (" ") will be written to right justify the expression in the output field. If the "fieldwidth" value is less than the number of characters required to represent the expression then the "fieldwidth" value is ignored and this value is used instead. If the expression is negative a minus sign ("-") is written to the output field, then all signifiant digits of the expression are written to the output field.

If a program contains the statement:

```
write('***', i:3, '***');
```

then for various values of "i" the following output would be produced:

WRITE (continued)

"i" VALUE	OUTPUT
1	*** 1***
-1	*** -1***
0	*** 0***
127	***127***
-127	***-127***
-4096	***-4096***
123456	***123456***

REAL TYPE

If the expression is of type REAL and the "fieldwidth" parameter is omitted then a default value of 20 will be used for "fieldwidth". The "fieldwidth" value must be greater than or equal to zero.

WRITE (continued)

FLOATING POINT FORM

If the "fracdigits" parameter is omitted then the real expression will be written in the "floating-point" form as follows.

If the "fieldwidth" value is less than 9 then a default value of 9 will be used for "fieldwidth". A sequence of characters will be written to the file "fileid" as follows. The first character written will be a minus ("-") character if the expression is negative, otherwise a space (" ") character is written. Next, the first digit of the expression and a decimal point (".") are written. Next, ("fieldwidth" - 8) fractional digits are written. And lastly, the exponent value is written as follows. The exponent character ("E") is written, a plus ("+") or minus ("-") is written, and the exponent value is written as three digits.

For the sample statement:

```
write('***', r:10, '***');
```

the following would be produced for various values of "r":

"r" VALUE	OUTPUT
1.0	*** 1.00E+000***
-1.0	***-1.00E+000***
0.0	*** 0.00E+000***
0.1	*** 1.00E-001***
123.456	*** 1.24E+002***

FIXED POINT FORM

If the "fracdigits" parameter is specified then the real expression will be written in "fixed-point" form as follows.

If the field width value is greater than the number of characters required to represent the expression then a sufficent number of spaces will be written to right justify the expression in the output field. If the "fieldwidth" value is less than the number of characters required to represent the expression then the "fieldwidth" value will default to this value.

WRITE (continued)

The "fixed-point" form is as follows. If the expression is negative a minus ("-") character is written, then ("fieldwidth" - "fracdigits" - 1) digits are written, then a decimal point (".") character is written, and finally "fracdigits" digits are written.

For the sample statement:

```
write('***', r:10:3, '***');
```

the following would be produced for various values of "r":

"r" VALUE	OUTPUT
1.0	*** 1.000***
-1.0	*** -1.000***
0.0	*** 0.000***
0.1	*** 0.100***
123.4567	*** 123.457***
1.23E6	***1230000.000***
-1.234E7	***-12340000.000***

An extension to OS-9/68000 Pascal allows the value of "fracdigits" to be zero. If "fracdigits" has a value of zero then neither the decimal point nor any digits to the right of the decimal point will be written. This convention allows you to write a real number in an integer form.

BOOLEAN TYPE

If the expression is of type BOOLEAN and the "fieldwidth" parameter is omitted then a default value of 10 will be used for "fieldwidth". The "fieldwidth" value must be greater than or equal to zero.

The "fieldwidth" parameter specifies the output field width in which the output string is to be right justified in. If the output string is longer than the "fieldwidth" value then the output string will be truncated to "fieldwidth" value characters. If the boolean expression is true the output string is "TRUE", otherwise the output string is "FALSE".

WRITE (continued)

STRING, CHARACTER, or PACKED ARRAY OF CHARACTER TYPE

If the expression is of type STRING, CHARACTER, or PACKED ARRAY OF CHAR then it will be written to the file as follows.

If the "fieldwidth" parameter is omitted then "fieldwidth" will default to the actual size of the expression. The "fieldwidth" parameter specifies the output field width in which the output string is to be right justified in. If the output string is longer than the "fieldwidth" value then the output string will be truncated to "fieldwidth" value characters.

WRITEEOF

DECLARATION:

```
PROCEDURE WRITEEOF(fileid);
```

PARAMETERS:

"fileid" : any file variable.

FUNCTION:

This procedure marks the file's current file position as the end-of-file position. This procedure is used to shorten or lengthen a file. After having positioned the file pointer, for the file "fileid", to the desired position, via file reads, file writes, or the "REPOSITION" procedure, you simply execute this procedure to mark this position as the end-of-file position. Any records which previously existed beyond this position in the file are, in effect, thrown away. Any records which are added, because the new end-of-file position is higher than the old end-of-file position, will be uninitialized.

WRITE (continued)

EXAMPLE PROGRAM:

```
PROGRAM writeeof(output, newfile);
VAR
   i           : integer;
   newfile     : FILE OF integer;
BEGIN
{create newfile}
rewrite(newfile, 'filex', $020b);
{write 500 records}
FOR i := 1 TO 500 DO
   BEGIN
   newfile^ := i;
   put(newfile);
   END;
writeln(output, 'Filesize = ', filesize(newfile));
{set position to record 155}
reposition(newfile, 155);
{set position 155 as end-of-file}
writeeof(newfile);
writeln(output, 'Filesize = ', filesize(newfile));
END.
```

WRITELN

DECLARATION:

```
PROCEDURE WRITELN{({text_fileid} {,} expression1{:fieldwidth
                  {:fracdigits}} {,expression2{:fieldwidth
                  {:fracdigits}} ... {,expressionN
                  {:fieldwidth{:fracdigits}}} })};
```

PARAMETERS:

"text_fileid"	: must be a text file variable.
"expression1", "expression2", ... "expressionN"	: These expressions can be of type BOOLEAN, CHAR, INTEGER, LINTEGER, REAL, STRING, or PACKED ARRAY OF CHAR.
"fieldwidth"	: This optional integer expression may be used to specify the output field width.
"fracdigits"	: This optional integer expression may only be specified when writing a real number. This expression specifies the number of fractional decimal places in the output string.

FUNCTION:

If the file "text_fileid" parameter is omitted then the file "OUTPUT" will be used.

This procedure writes the expressions as character sequences to the file "text_fileid". See the procedure "WRITE" for a description of this process. After all expressions have been written a end-of-line marker is written to the file. Also the linelength variable in the file's FCB is zeroed.

EXAMPLES:

```
writeln;
writeln('A test');
writeln(output, 'I=', i:3, ' Z*Z=', z*z:1);
```

STRING DECLARATION

The string data type is an OS-9/68000 Pascal extension. Strings are packed arrays of characters with the special attribute that they are treated as variable length items. When a string is declared it is allocated a fixed amount of memory to contain the maximum declared string length. The syntax for a string declaration is as follows.

```
<string-type> ::= STRING | STRING "[" <integer-constant> "]"
```

If the optional brackets and maximum string length are omitted then a default maximum string length of 80 will be used. The <integer-constant> value must be in the range of 1 thru 255. Here are some examples of string declarations.

```
CONST
   max_string_length = 255;
TYPE
   maxstring = string[max_string_length];
VAR
   str1 : string;
   str2 : maxstring;
```

In the above example the string "str1" can contain a string up to 80 characters in length while the string "str2" can contain a string up to 255 characters in length. Internally the string data type is defined as "PACKED ARRAY [0 .. <integer-constant>] OF CHAR", thus a string variable will contain one more byte of storage than the maximum declared string length. Individual elements of a string can be indexed, like normal packed arrays of characters, from 1 thru the maximum declared string length. Element zero, which is inaccessible, contains an unsigned byte denoting the current length of the string. A run-time error will be generated any time you try to index an element in the string whose ordinal index is greater than the current string length.

STRING ASSIGNMENT CONFORMABILITY

OS-9/68000 Pascal has made every attempt to allow conformability between strings, "packed arrays of characters", and characters. You can assign one item type into the other with only a few restrictions.

ASSIGNING A STRING TO AN ARRAY

When a string is assigned to a "packed array of char" and the string contains fewer elements, at the time the assignment is executed, than the destination array then the destination array will be padded with blank (" ") characters to fill it out. Here is an example.

```
PROGRAM string_to_array;
VAR
   arr1 : PACKED ARRAY [1..6] OF char;
   str1 : string;
BEGIN
str1 := 'Abc';  {string length is 3 elements}
arr1 := str1;   {array length is 6 elements, pad with blanks}
writeln('*', arr1, '*'); {output is "*Abc   *"}
END.
```

If you attempt to assign a string to an array and the current string length is greater than the destination array length then a run-time error will be generated.

ASSIGNING AN ARRAY TO A STRING

When a "packed array of char" is assigned to a string the array is stored in its entirety. That is the destination string will be assigned all of the elements of the array and the destination string length will be set to the length of the array. Here is an example program.

```
PROGRAM array_to_string;
VAR
   str1 : string;
   arr1 : PACKED ARRAY [1..6] OF char;
BEGIN
arr1 := '123456';
str1 := arr1; {string is assigned '123456' and length is 6}
writeln('*', str1, '*'); {output is "*123456*"}
END.
```

If you attempt to assign an array which has more elements than the declared maximum string length of the destination string then a run-time error will be generated.

ASSIGNING A CHARACTER TO A STRING

When a character expression is assigned to a string the string length is set to one and the character is stored in element one of the string.

THE NULL STRING

A null string is a string whose current length is zero, i.e. contains no elements. A null string can be generated by the use of string functions or by assigning the null string constant to the string. Here is an example.

```
PROGRAM null_string;
VAR
   str1 : string;
BEGIN
str1 := ''; {str1 has a length of zero, null string}
writeln('*', str1, '*'); {output is "**"}
END.
```

TYPELESS PARAMETERS

The four predeclared routines FILLCHAR, MOVELEFT, MOVERIGHT, and SCAN have certain parameters that may be of any result type. These parameters are refered to as TYPELESS parameters. The argument passed to a routine that accepts a TYPELESS parameter must result in a variable reference. The following program shows examples of valid arguments that can be passed to a TYPELESS parameter.

```
PROGRAM typeless;
TYPE
   a = PACKED ARRAY [0..999] OF char;
   r = RECORD
       b : boolean;
       c : char;
       END;
VAR
   ch : char;
   a1 : a;
   r1 : r;
   p1 : ^r;
BEGIN
new(p1);
{           { These are valid arguments }

  ch, a1, r1, a1[100], r1.b, r1.c, p1, p1^, p1^.b, p1^.c

           { These are invalid arguments }

  'c', 0, 'a string'     Constants are not allowed
  NOT r.b, succ(ch)      Expressions are invalid

END.
```

VECTOR SLICE

The VECTOR SLICE operation allows the programmer to extract a substring of characters from a "STRING" or "PACKED ARRAY OF char" expression. The syntax for the VECTOR SLICE operation is

"array-expression[start-expression FOR length-constant]"

where the "start-expression" value defines the start index within the "array-expression" and the "length-constant" is an integer constant that defines the length of the substring extracted starting at the start index. The result type of the VECTOR SLICE operation (for conformability) is "PACKED ARRAY [1..length-constant] OF CHAR". Note that VECTOR SLICE operation is an OS-9/68000 Pascal extension.

The following program shows an example use of VECTOR SLICE operations.

```
PROGRAM vector(input, output);
VAR
   over : RECORD
          CASE integer OF
         1 : (l       : linteger);
         2 : (i       : integer);
         3 : (c       : PACKED ARRAY [1..4] OF char);
          END;
   disk : FILE OF PACKED ARRAY [0..255] OF char;
BEGIN
reset(disk, '/dd@', $0100);
over.c[2 FOR 3] := disk^[$15 FOR 3]; {extract bootstrap lsn}
over.c[1] := chr(0); {form 32 bit lsn value}
reposition(disk, over.l); {seek to bootstrap}
END.
```

The following pages describe the predeclared string and array functions and procedures of OS-9/68000 Pascal.

CAP

DECLARATION:

PROCEDURE CAP(source)

PARAMETERS:

"source" : must be a variable which is one of the following types, "STRING", "CHAR", or "PACKED ARRAY [1..??] OF CHAR".

FUNCTION:

This procedure causes any lower-case characters in the "source" string to be converted to their upper-case value.

EXAMPLE PROGRAM:

```
PROGRAM cap(output);
VAR
   s     : string;
BEGIN
s := 'This is a mixture of UPPER and lower case characters.';
cap(s);
writeln(output, s);
END.
```

CONCAT

DECLARATION:

```
FUNCTION CONCAT(source1 {, sourceN} ) : string
```

PARAMETERS:

"source1"	
...	
"sourceN"	: one or more expressions whose result type must be one of the following types, "CHAR", "STRING", or "PACKED ARRAY [1..??] OF CHAR".

FUNCTION:

This function results in a string being returned which is formed by concatenating each of the "source" values, one after another. A run-time error will be generated if the result string is greater than 255 characters.

EXAMPLE PROGRAM:

```
PROGRAM concat(output);
VAR
   s, t : string;
BEGIN
t := 'snake-like';
s := concat('There is a long poisonous ', concat(t, 'object'));
writeln(output, s);
END.
```

COPY

DECLARATION:

```
FUNCTION COPY(source, start, count) : string
```

PARAMETERS:

"source"	: an expression whose result type must be one of the following types, "STRING", "CHAR", or "PACKED ARRAY [1..??] OF CHAR".
"start"	: an integer expression.
"count"	: an integer expression.

FUNCTION:

This function results in a string being returned which consists of "count" number of characters taken from the "source" value starting at the "start" element. A run-time error will be generated if an attempt is made to reference a character position which does not exist in the "source" expression, or if the result string has more than 255 characters.

EXAMPLE PROGRAM:

```
PROGRAM copy(output);
VAR
   s     : string;
BEGIN
s := 'This is a very long string.';
s := copy(s, 16, length(s) - 16);
writeln(output, s);
END.
```

DELETE

DECLARATION:

```
PROCEDURE DELETE(string_variable, start, count)
```

PARAMETERS:

"string_variable" : must be a string type variable

"start" : an integer expression

"count" : an integer expression

FUNCTION:

This procedure deletes "count" number of characters from the string "string_variable" starting at the "start" index position in the string. If the "start" and "count" parameters result in an index out of range for the "string_variable" a run-time error will be generated.

EXAMPLE PROGRAM:

```
PROGRAM delete(output);
VAR
   s        : string;
BEGIN
s := 'Today is a very good day.';
delete(s, 12 {index of "very"}, 5 {delete 5 characters});
writeln(output, s);
END.
```

FILLCHAR

DECLARATION:

```
PROCEDURE FILLCHAR(destination, length, value)
```

PARAMETERS:

"destination"	: a typeless parameter.
"length"	: an integer expression.
"value"	: an expression whose result type is one of the following simple types, CHAR, BOOLEAN, INTEGER, LINTEGER, or an enumeration.

FUNCTION:

This procedure fills memory starting at the "destination" memory location thru higher memory for "length" instances of "value". The "value" parameter is treated as an 8-bit item. Thus if "value" is of type INTEGER only the low order 8-bits are used and if the type is LINTEGER a down conversion to INTEGER is performed first.

EXAMPLE PROGRAM:

```
PROGRAM fillchar;
VAR
   a1    : PACKED ARRAY [1..128] OF char;
   b1    : ARRAY [0..1023] OF boolean;
   c1    : ARRAY [1..100] OF (red, green, blue, none);
   d1    : ARRAY [1..10] OF RECORD
           i    : integer;
           l    : linteger;
           END;
BEGIN
fillchar(a1, sizeof(a1), ' ');
fillchar(b1, sizeof(b1), false);
fillchar(c1, sizeof(c1), none);
fillchar(d1, sizeof(d1), 0);
END.
```

INSERT

DECLARATION:

```
PROCEDURE INSERT(source, destination, start)
```

PARAMETERS:

"source"	: must be an expression of one of the following types, "STRING", "CHAR", or "PACKED ARRAY [1..??] OF CHAR".
"destination"	: must be a string variable.
"start"	: an integer expression.

FUNCTION:

The result of this procedure is to insert the "source" string of characters into the "destination" string variable starting at the "start" position in the "destination" string. Any existing characters already in the "destination" string starting at the "start" position are moved to the right to make room for the "source" string. A run-time error will be generated if the "start" expression has a value not in the range of one thru one plus the current "destination" string length or if the insert operation results in a string too large to be stored in the "destination" string.

EXAMPLES PROGRAM:

```
PROGRAM insert(output);
VAR
   s    : string;
BEGIN
s := 'There were three ravens';
insert('old ', s, 18);
writeln(output, s);
END.
```

LENGTH

DECLARATION:

```
FUNCTION LENGTH(source) : integer
```

PARAMETERS:

"source"	: an expression whose result type must be one of the following types, "STRING", "CHAR", or "PACKED ARRAY [1..??] OF CHAR".

FUNCTION:

This function results in an integer value being returned which denotes the current number of characters contained in the "source" value.

EXAMPLE PROGRAM:

```
PROGRAM length(output);
VAR
   s     : string;
BEGIN
s := 'This is a string whose length is';
writeln(output, s, length(s));
END.
```

MOVELEFT

DECLARATION:

```
PROCEDURE MOVELEFT(source, destination, length)
```

PARAMETERS:

"source"	: a typeless parameter.
"destination"	: a typeless parameter.
"length"	: an integer expression.

FUNCTION:

This procedure moves "length" number of bytes from the "source" memory location to the "destination" memory location. The bytes are moved from left (lower address) to right (higher address).

EXAMPLE PROGRAM:

```
PROGRAM moveleft(output);
VAR
   src, dst      : PACKED ARRAY [1..10] OF char;
BEGIN
src := '1234567890';
dst := '**********';
moveleft(src, dst, 5);
writeln(output, dst {'12345*****'});
END.
```

MOVERIGHT

DECLARATION:

```
PROCEDURE MOVERIGHT(source, destination, length)
```

PARAMETERS:

"source" : a typeless parameter.

"destination" : a typeless parameter.

"length" : an integer expression.

FUNCTION:

This procedure moves "length" number of bytes from the "source" memory location to the "destination" memory location. The bytes are moved from right (higher address) to left (lower address).

This procedure accomplishes the same thing as MOVELEFT except when bytes are moved to an overlapping location within the same array.

EXAMPLE PROGRAM:

```
PROGRAM moveright(output);
VAR
   src      : PACKED ARRAY [1..10] OF char;
BEGIN
src := '1234567890';
moveright(src, src[3], 5);
writeln(output, src {'1231234590'});
src := '1234567890';
moveleft(src, src[3], 5);
writeln(output, src {'1231231290'});
END.
```

POS

DECLARATION:

```
FUNCTION POS(pattern, source) : integer
```

PARAMETERS:

"pattern",
"source" : are expressions whose result type must be one of the following types, "STRING", "CHAR", or "PACKED ARRAY [1..??] OF CHAR".

FUNCTION:

This function returns an integer value indicating the first position that the "pattern" string was found in the "source" string. If no match can be found, zero will be returned.

EXAMPLE PROGRAM:

```
PROGRAM pos(output);
VAR
   pattern,
   source     : string;
BEGIN
source := 'Today is the first day of summer.';
pattern := 'day';
writeln(output, 'The position of "', pattern, '" in "',
        source, '" is ', pos(pattern, source):1);
END.
```

SCAN

DECLARATION:

FUNCTION SCAN(length, <partial expr>, source) : integer

PARAMETERS:

"length" : an integer expression.

"<partial expr>" : "=" or "<>" symbol followed by a character expression.

"source" : a typeless parameter.

FUNCTION:

This function scans memory starting at the "source" location until the "<partial expr>" is satisfied, or until abs("length") number of bytes have been scanned, whichever occurs first. If "length" is negative then the scanning goes thru decreasing memory addresses otherwise the scanning goes thru increasing memory addresses. When scanning stops because of the "<partial expr>" evaluating true an integer value is returned that indicates the offset from the "source" memory location to the address at which the scanning stopped. However if the scanning stops because of abs("length") bytes being scanned without the "<partial expr>" being evaluated true then the "length" value is returned as the result.

EXAMPLE PROGRAM:

```
PROGRAM scan(output);
VAR
   test    : string;
BEGIN
test := 'For he on honey dew hath fed';
writeln(output, scan(10, ='h', test[11])); {scan = 0}
writeln(output, scan(10, ='h', test[12])); {scan = 9}
writeln(output, scan(-9, ='h', test[9]));  {scan = -4}
writeln(output, scan(10, <>'h', test[11])); {scan = 1}
writeln(output, scan(-10, =chr(0), test[11])); {scan = -10}
END.
```

STR

DECLARATION:

```
PROCEDURE STR(value, destination)
```

PARAMETERS:

"value" : a linteger expression.

"destination" : must be a string variable.

FUNCTION:

This procedure results in the "destination" string being set to the ASCII character sequence of the "value" expression. If the "value" expression is negative a leading minus ("-") character will be generated in the character sequence. A run-time error be generated if the "destination" string is too small to store the character sequence.

EXAMPLE PROGRAM:

```
PROGRAM str(output);
VAR
   s     : string;
   i     : integer;
BEGIN
i := maxint DIV 1317;
str(i, s);
writeln(output, 'i=', s);
END.
```

STRINGCMP

DECLARATION:

```
FUNCTION STRINGCMP(left_source, right_source) : integer
```

PARAMETERS:

"left_source",
"right_source" : are expressions whose result type must be one of the following types, "STRING", "CHAR", or "PACKED ARRAY [1..??] OF CHAR".

FUNCTION:

This function compares the "left_source" string to the "right_source" string and returns an integer value as follows.

RELATION	RETURNS
"left_source" < "right_source"	-1
"left_source" = "right_source"	0
"left_source" > "right_source"	1

Note that the compare operation of string operands denote lexicographic ordering according to the ordering of the ASCII character set.

EXAMPLE PROGRAM:

```
PROGRAM stringcmp(input, output);
VAR
   left, right    : string;
BEGIN
REPEAT
   write(output, 'Enter left string :');
   readln(input, left);
   write(output, 'Enter right string:');
   readln(input, right);
   writeln(output, 'Result of left compared to right = ',
           stringcmp(left, right):1);
UNTIL false;
END.
```

This Page Intentionally Blank

MATHABORT and MATHRESULT

OS-9/68000 PASCAL has added extensions that allow the programmer to intercept and process mathematical program errors. Mathematical errors include overflow, underflow, divide by zero, out-of-range arguments, and so forth.

The predeclared global boolean variable MATHABORT controls weather the program aborts on a mathematical error. If MATHABORT is TRUE (the default condition when program starts) when a mathematical error occurs then the program will abort. However, if MATHABORT is FALSE when a mathematical error occurs then the program won't abort and the OS-9/68000 PASCAL run-time error (see APPENDIX B) will be stored in the predeclared integer variable MATHRESULT. If more than one mathematical error occurs during execution of a pascal statement only the last error value will be saved in MATHRESULT. Also, if no mathematical error occur then the value in MATHRESULT will not be modified.

The following program shows an example use of MATHABORT and MATHRESULT.

```
PROGRAM MATH(input, output);
VAR
   i, j, k  : integer;
BEGIN
mathabort := false; {disable program abort on math errors}
REPEAT
   write(output, 'Enter I and J :');
   readln(input, i, j);
   mathresult := 0; {clear last math error value}
   k := i * j;
   write(output, i:1, '*', j:1, '=');
   IF mathresult <> 0 {check for math error} THEN
      writeln(output, 'error')
   ELSE
      writeln(output, k:1);
UNTIL false;
END.
```

REALPRECISION

REALPRECISION is a predeclared global real variable and is an OS-9/68000 Pascal extension. This variable is used on all Pascal transcendental routines (SIN, COS, TAN, ARCSIN, ARCCOS, ARCTAN, LOG, LN, SQRT, EXP) along with the power ('**') operator. This variable is initialized to a value of 1E-14 (14 digits of precision) when a pascal program starts. You can change this variable to select more or less precision for these transcendental routines (see page 12-10 in the OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL).

The following program shows an example use of REALPRECISION.

```
PROGRAM realtest(input, output);
VAR
   r    : real;
BEGIN
REPEAT
   write(output, 'Enter real value :');
   readln(input, r);
   realprecision := 1e-8; {set to 8 digit precision}
   writeln(output, 'SIN(', r, ')=', sin(r));
   realprecision := 1e-12; {set to 12 digit precision}
   writeln(output, 'SIN(', r, ')=', sin(r));
UNTIL false;
END.
```

The following pages describe the arithmetic functions and procedures of OS-9/68000 Pascal.

ABS

DECLARATION:

```
FUNCTION ABS(source) : result
```

PARAMETERS:

"source"	: an expression whose result type is one of the following types, REAL, INTEGER, or LINTEGER. The result type of this function has the same type as the "source" expression.

FUNCTION:

This function returns the absolute value of the "source" expression.

ARCCOS

DECLARATION:

```
FUNCTION ARCCOS(source) : real
```

PARAMETERS:

"source"	: an expression whose result type is one of the following types, REAL, INTEGER, or LINTEGER.

FUNCTION:

This function returns the ARC COSINE of the "source" expression in radians. This function uses the variable REALPRECISION to determine the result precision.

ARCSIN

DECLARATION:

```
FUNCTION ARCSIN(source) : real
```

PARAMETERS:

"source"	: an expression whose result type is one of the following types, REAL, INTEGER, or LINTEGER.

FUNCTION:

This function returns the ARC SINE of the "source" expression in radians. This function uses the variable REALPRECISION to determine the result precision.

ARCTAN

DECLARATION:

```
FUNCTION ARCTAN(source) : real
```

PARAMETERS:

"source"	: an expression whose result type is one of the following types, REAL, INTEGER, or LINTEGER.

FUNCTION:

This function returns the ARC TANGENT of the "source" expression in radians. This function uses the variable REALPRECISION to determine the result precision.

CHR

DECLARATION:

```
FUNCTION CHR(source) :-char
```

PARAMETERS:

"source" : an integer expression.

FUNCTION:

The result of this function is a value of char-type whose ordinal value is equal to the value of the "source" expression. An error will be generated if the "source" expression is not in the range of 0..255.

For any value, ch, of char-type, the following is true:

```
chr(ord(ch)) = ch
```

EXAMPLE PROGRAM:

```
PROGRAM chr(output);
VAR
   i     : integer;
BEGIN
FOR i := $20 TO $7E DO
   writeln(output, 'CHR(', i:1, ')=', chr(i));
END.
```

CONVERT

DECLARATION:

```
PROCEDURE CONVERT(result, source)
```

PARAMETERS:

"result"	: must be a variable of one of the following types, INTEGER, LINTEGER, or REAL.
"source"	: an expression whose result type is one of the following types, "CHAR", "STRING", or "PACKED ARRAY [1..??] OF CHAR".

FUNCTION:

The result of this procedure is to convert the "source" character string into its numeric value and to return that value into the "result" variable. An error will be generated if the numeric result value can't be represented in the "result" variable (i.e. value too large). Also an error will be generated if the "source" character string isn't a valid numeric string.

The conversion will start with the first character within the "source" string thru the last character within the "source" string or until a non-numeric character is found.

Note: when converting to a real result, "result" is of type real, the "source" string must be terminated with a null "chr(0)" character.

EXAMPLE PROGRAM:

```
PROGRAM convert(input, output);
VAR
   i    : integer;
   r    : real;
   s    : string;
BEGIN
REPEAT
   write('Enter integer string :');
   readln(s);
   convert(i, s);
   writeln('Value =', i);
   write('Enter real string :');
   readln(s);
```

CONVERT (continued)

```
   convert(r, concat(s, chr(0)));
   writeln('Value =', r);
UNTIL false;
END.
```

COS

DECLARATION:

```
FUNCTION COS(source) : real
```

PARAMETERS:

"source"	: an expression whose result type is one of the following types, REAL, INTEGER, or LINTEGER.

FUNCTION:

This function returns the COS of the "source" expression. The "source" expression is in radians. This function uses the variable REALPRECISION to determine the result precision.

EXP

DECLARATION:

```
FUNCTION EXP(source) : real
```

PARAMETERS:

"source"	: an expression whose result type is one of the following types, REAL, INTEGER, or LINTEGER.

FUNCTION:

This function returns the EXPONENTIAL of the "source" expression, that is, e**source. This function uses the variable REALPRECISION to determine the result precision.

FIELDGET

DECLARATION:

```
FUNCTION FIELDGET(source, start, count) : result
```

PARAMETERS:

"source" : either an integer or linteger expression.

"start" : an integer expression.

"count" : an integer expression.

FUNCTION:

If the "source" expression is of type integer then the following statements apply:

The "start" expression must be in the range of 0..15.

The "count" expression must be in the range of 1..16.

The "source" expression is considered to be a string of 16 binary bits numbered 15 downto 0 with bit-0 being the least significant bit.

The function result is of type integer.

otherwise, if the "source" expression is of type linteger then these statements apply:

The "start" expression must be in the range of 0..31.

The "count" expression must be in the range of 1..32.

The "source" expression is considered to be a string of 32 binary bits numbered 31 downto 0 with bit-0 being the least significant bit.

The function result is of type linteger.

FIELDGET (continued)

The result of this function is to extract "count" number of bits from the "source" expression starting at the "start" bit number. The result is right justified and left filled with zeros to form the result value. Bits are extracted in a wrap-around manner. Thus if the statement "fieldget($5432, 2, 6)" were executed, bits 2, 1, 0, 15, 14, and 13 would be extracted, resulting in a value of $0012 being returned. If the "source" expression were of type linteger then the wrap-around would occur starting at bit 31 instead of bit 15.

EXAMPLE PROGRAM:

```
PROGRAM fieldget(input, output);
VAR
   num  : linteger;

PROCEDURE hex(value : linteger; digits : integer);
   VAR
      i, v   : integer;
   BEGIN
   FOR i := digits DOWNTO 1 DO
      BEGIN
      v := fieldget(value, digits * 4 - 1, 4);
      IF v > 9 THEN
         write(chr(ord('A') - 10 + v))
      ELSE
         write(chr(ord('0') + v));
      END;
   writeln;
   END;

BEGIN
WHILE NOT eof DO
   BEGIN
   readln(num);
   hex(num, 8);
   hex(num, 6);
   hex(num, 4);
   hex(num, 1);
   END;
END.
```

FIELDPUT

DECLARATION:

```
PROCEDURE FIELDPUT(destination, start, count, source)
```

PARAMETERS:

"destination"	: must be an integer or linteger variable.
"start"	: an integer expression.
"count"	: an integer expression.
"source"	: an integer or linteger expression.

FUNCTION:

If the "destination" variable is of type integer then the following statements apply:

The "start" expression must be in the range of 0..15.

The "count" expression must be in the range of 1..16.

The "source" expression and "destination" variable are considered to be a string of 16 binary bits numbered 15 downto 0 with bit-0 being the least significant bit.

otherwise, if the "destination" variable is of type linteger then these statements apply:

The "start" expression must be in the range of 0..31.

The "count" expression must be in the range of 1..32.

The "source" expression and "destination" variable are considered to be a string of 32 binary bits numbered 31 downto 0 with bit-0 being the least significant bit.

FIELDPUT (continued)

The result of this procedure is to extract the "count" number of least significant bits from the "source" expression and to insert those bits into the "destination" variable starting at the "start" bit number. Bits are stored in a wrap-around manner. For example, if "count" is 6 and "start" is 2 then bits 5, 4, 3, 2, 1, and 0 of the "source" expression would be stored in bits 2, 1, 0, 15, 14, and 13 of the "destination" variable. If the "destination" variable is of type linteger then the wrap-around would occur at bit 31 instead of bit 15.

Any bits in the "destination" variable which were not used by this procedure are left unmodified.

EXAMPLE PROGRAM:

```
PROGRAM fieldput;
TYPE
   rkd  = RECORD  {record size is 8 bytes}
          b    : ARRAY [1..4] OF boolean;
          c    : char;
          i    : -1..14;
          END;

FUNCTION pack(r1 : rkd) : integer;
{pack record of 8 bytes into 2 byte integer}
   VAR
      result : integer;
      inx    : integer;
   BEGIN
   result := 0;
   {pack boolean array b into bits 0 thru 3}
   FOR inx := 1 TO 4 DO
      fieldput(result, inx - 1, 1, ord(r1.b[inx]));
   {pack character c into bits 11 thru 4}
   fieldput(result, 11, 8, ord(r1.c));
   {pack subrange i into bits 15 thru 12}
   inx := r1.i + 1; {adjust to zero base}
   fieldput(result, 15, 4, inx);
   pack := result;
   END;

FUNCTION unpack(v1 : integer) : rkd;
{unpack a 2 byte integer into an 8 byte record}
   VAR
      r1     : rkd;
```

FIELDPUT (continued)

```
   inx     : integer;
BEGIN
{unpack boolean array b from bits 0 thru 3}
FOR inx := 1 TO 4 DO
   r1.b[inx] := fieldget(v1, inx - 1, 1) = 1;
{unpack character c from bits 11 thru 4}
r1.c := chr(fieldget(v1, 11, 8));
{unpack subrange i from bits 15 thru 12}
r1.i := fieldget(v1, 15, 4) - 1;
unpack := r1;
END;
```

FLOAT

DECLARATION:

```
FUNCTION FLOAT(source) : real
```

PARAMETERS:

"source" : an expression whose result type is one of the following types, INTEGER, LINTEGER, or REAL.

FUNCTION:

The result of this function call is a value of real-type whose value is equal to the "source" expression. This function will force an explict type conversion.

EXAMPLE PROGRAM:

```
PROGRAM float(output);
VAR
   r    : real;
   l    : linteger;
BEGIN
l := maxlint;
r := float(l) * 1_000_000;
{Without the float call this statement would produce a math
overflow run-time error since the multiply operation would be done
on linteger types and the result could not be represented in a
linteger value}
writeln(output, r);
END.
```

INT

DECLARATION:

```
FUNCTION INT(source) : integer
```

PARAMETERS:

"source" : a linteger or integer expression.

FUNCTION:

The result of this function call is a value of integer-type whose value is equal to the "source" expression. This function will force an explict type conversion.

EXAMPLE PROGRAM:

```
PROGRAM int(output);
VAR
   l    : linteger;
BEGIN
l := maxint;
writeln(output, int(l));
END.
```

LN

DECLARATION:

```
FUNCTION LN(source) : real
```

PARAMETERS:

"source"	: an expression whose result type is one of the following types, REAL, INTEGER, or LINTEGER.

FUNCTION:

This function returns the NATURAL LOGARITHM of the "source" expression. The "source" expression must have a value greater than zero. This function uses the variable REALPRECISION to determine the result precision.

LOG

DECLARATION:

```
FUNCTION LOG(source) : real
```

PARAMETERS:

"source"	: an expression whose result type is one of the following types, REAL, INTEGER, or LINTEGER.

FUNCTION:

This function returns the COMMON LOGARITHM of the "source" expression. The "source" expression must have a value greater than zero. This function uses the variable REALPRECISION to determine the result precision.

LONG

DECLARATION:

```
FUNCTION LONG(source) : linteger
```

PARAMETERS:

"source" : a linteger or integer expression.

FUNCTION:

The result of this function call is a value of linteger-type whose value is equal to the "source" expression. This function will force an explict type conversion.

EXAMPLE PROGRAM:

```
PROGRAM long(output);
VAR
   i     : integer;
BEGIN
i := maxint;
writeln(output, long(i) * 1000);
{Without the long call this statement would produce a math
overflow run-time error since the multiply operation would be done
on integer types and the result could not be represented in a
integer value}
END.
```

LROUND

DECLARATION:

```
FUNCTION LROUND(source) : linteger
```

PARAMETERS:

"source" : must be a real expression.

FUNCTION:

The result of this function is the linteger value of the rounded "source" expression. If the "source" expression is positive, then this function call is equivalent to "ltrunc(source + 0.5)", otherwise if the "source" expression is negative then this function call is equivalent to "ltrunc(source - 0.5)".

An error will be generated if the result value can't be represented in a linteger value.

EXAMPLES:

```
lround(12.3) = 12
lround(12.7) = 13
lround(4.5)  = 5
lround(-4.5) = -5
lround(-0.5) = -1
lround(-100000.49) = -100000
```

LTRUNC

DECLARATION:

```
FUNCTION LTRUNC(source) : linteger
```

PARAMETERS:

"source" : must be a real expression.

FUNCTION:

The result of this function is the linteger value of the "source" expression after any fractional part has been discarded. An error will be generated if the result value can't be represented in a linteger value.

EXAMPLES:

```
ltrunc(12.3)  = 12
ltrunc(12.7)  = 12
ltrunc(0.5)   = 0
ltrunc(-0.5)  = 0
ltrunc(-12.3) = -12
ltrunc(-100000.9) = -100000
```

ODD

DECLARATION:

```
FUNCTION ODD(source) : boolean
```

PARAMETERS:

"source" : an integer or linteger expression.

FUNCTION:

This function returns a boolean result indicating weather the "source" expression is an odd value. TRUE is returned if it is, FALSE otherwise.

EXAMPLE PROGRAM:

```
PROGRAM odd(output);
VAR
   i     : integer;
BEGIN
FOR i := -100 to 100 DO
   IF odd(i) THEN
      writeln(output, i:4, ' is odd.');
END.
```

ORD

DECLARATION:

```
FUNCTION ORD(source) : result
```

PARAMETERS:

"source" : an expression of an ordinal type (i.e. CHAR INTEGER, BOOLEAN, ENUMERATION, LINTEGER, or SUBRANGE), or a POINTER type.

FUNCTION:

The result of this function is the ordinal number of the value of the "source" expression. The result type of this function is INTEGER, unless the "source" expression is of LINTEGER or POINTER type which then returns a result of LINTEGER type.

EXAMPLE PROGRAM:

```
PROGRAM ord(output);
VAR
   ch    : char;
   p     : ^char;
   e     : (red, green, blue);
BEGIN
new(p);
writeln(output, 'ORD(TRUE)=', ord(true));
FOR ch := 'a' TO 'z' DO
   writeln(output, 'ORD(', ch, ')=', ord(ch));
writeln(output, 'ORD(GREEN)=', ord(green));
writeln(output, 'ORD(POINTER)=', ord(p));
END.
```

PRED

DECLARATION:

```
FUNCTION PRED(source) : result
```

PARAMETERS:

"source" : an expression of an ordinal type (i.e. CHAR INTEGER, BOOLEAN, ENUMERATION, SUBRANGE).

FUNCTION:

The result of this function is the ordinal value that is one less than that of the "source" expression, if such a value exists. An error will be generated if such a value does not exist. The result type of this function is of the same type as the "source" expression.

EXAMPLE PROGRAM:

```
PROGRAM pred;
VAR
   c    : char;
   b    : boolean;
   e    : (red, blue);
   i    : integer;
   l    : linteger;
BEGIN
c := ' ';
b := true;
e := blue;
i := 0;
l := maxlint;
c := pred(c);
b := pred(b);
e := pred(e);
i := pred(i);
l := pred(l);
END.
```

RANDOM

DECLARATION:

PROCEDURE RANDOM(seed)

PARAMETERS:

"seed" : must be a linteger variable.

FUNCTION:

This procedure produces a pseudo-random linteger value from the current "seed" value and returning the resulting pseudo-random number in the "seed" variable. The pseudo-random value is contained in the low order 31-bits of the linteger variable which results in the pseudo-random numbers being in the range of 0 thru 2147483647.

Note: as with most pseudo-random number generators the upper bits of the random number are more random than the lower bits, thus if you want a random number with less than 31 bits you should overdefine the upper bits of the "seed" variable.

RANDOM (continued)

EXAMPLE PROGRAM:

```
PROGRAM randomtest(output);
CONST
   loopcount  = 400000;  {number of darts to throw}
VAR
   seed       : RECORD
                CASE boolean OF
                 true  : (l    : linteger); {linteger random seed}
                 false : (i    : integer);
                         {overdefine upper 16-bits of
                          seed, thus return an integer
                          random number}
                END;
   loop       : linteger;
   x, y       : linteger;
   incircle   : linteger;
BEGIN
seed.l := 0; {initialize pseudo-number to 0}
incircle := 0; {initialize no darts in circle}
FOR loop := 1 TO loopcount DO
   BEGIN
   random(seed.l); {generate random number}
   x := seed.i; {extract random x value}
   random(seed.l); {generate random number}
   y := seed.i; {extract random y value}
   IF (sqr(x) + sqr(y)) <= sqr(long(maxint)) THEN
      {random dart throw is in circle}
      incircle := succ(incircle);
   END;
writeln('The value of PI calculated by throwing ',
        loopcount:1, ' random');
writeln('darts at a circle inclosed within a square is ',
        ((incircle / loopcount) * 4));
END.
```

ROUND

DECLARATION:

```
FUNCTION ROUND(source) : integer
```

PARAMETERS:

"source" : must be a real expression.

FUNCTION:

The result of this function is the integer value of the rounded "source" expression. If the "source" expression is positive, then this function call is equivalent to "trunc(source + 0.5)", otherwise if the "source" expression is negative then this function call is equivalent to "trunc(source - 0.5)".

An error will be generated if the result value can't be represented in an integer value.

EXAMPLES:

```
round(12.3) = 12
round(12.7) = 13
round(4.5)  = 5
round(-4.5) = -5
round(-0.5) = -1
```

SIN

DECLARATION:

```
FUNCTION SIN(source) : real
```

PARAMETERS:

"source"	: an expression whose result type is one of the following types, REAL, INTEGER, or LINTEGER.

FUNCTION:

This function returns the SINE of the "source" expression. The "source" expression is in radians. This function uses the variable REALPRECISION to determine the result precision.

SQR

DECLARATION:

```
FUNCTION SQR(source)
```

PARAMETERS:

"source"	: an expression whose result type is one of the following types, REAL, INTEGER, or LINTEGER. The result type of this function has the same type as the "source" expression.

FUNCTION:

This function returns the SQUARE of the "source" expression (i.e. "source" ** 2).

SQRT

DECLARATION:

```
FUNCTION SQRT(source) : real
```

PARAMETERS:

"source"	: an expression whose result type is one of the following types, REAL, INTEGER, or LINTEGER.

FUNCTION:

This function returns the SQUARE ROOT of the "source" expression. If the "source" expression is less than zero an error will be produced. This function uses the variable REALPRECISION to determine the result precision.

SUCC

DECLARATION:

```
FUNCTION SUCC(source) : result
```

PARAMETERS:

"source" : an expression of an ordinal type (i.e. CHAR INTEGER. BOOLEAN, ENUMERATION, SUBRANGE).

FUNCTION:

The result of this function is the ordinal value that is one more than that of the "source" expression, if such a value exists. An error will be generated if such a value does not exist. The result type of this function is of the same type as the "source" expression.

EXAMPLE PROGRAM:

```
PROGRAM succ;
VAR
   c    : char;
   b    : boolean;
   e    : (red, blue);
   i    : integer;
   l    : linteger;
BEGIN
c := ' ';
b := false;
e := red;
i := 0;
l := -maxlint;
c := succ(c);
b := succ(b);
e := succ(e);
i := succ(i);
l := succ(l);
END.
```

TAN

DECLARATION:

```
FUNCTION TAN(source) : real
```

PARAMETERS:

"source"	: an expression whose result type is one of the following types, REAL, INTEGER, or LINTEGER.

FUNCTION:

This function returns the TANGENT of the "source" expression. The "source" expression is in radians. This function uses the variable REALPRECISION to determine the result precision.

TRUNC

DECLARATION:

```
FUNCTION TRUNC(source) : integer
```

PARAMETERS:

"source" : must be a real expression.

FUNCTION:

The result of this function is the integer value of the "source" expression after any fractional part has been discarded. An error will be generated if the result value can't be represented in an integer value.

EXAMPLES:

```
trunc(12.3)  = 12
trunc(12.7)  = 12
trunc(0.5)   = 0
trunc(-0.5)  = 0
trunc(-12.3) = -12
```

This Page Intentionally Blank

OS-9/68000 Pascal fully supports all of the standard heap storage management procedures as defined in the ISO language specification. OS-9/68000 Pascal has added additional heap storage management procedures for compatability with older pascal systems and for greater utilization of the heap.

HEAPREQUEST and HEAP STRUCTURE

When the procedures "NEW" or "VARNEW" is called some number of bytes will be allocated from a special area of memory called "heap storage".

The "heap storage" area at program startup has zero heap blocks allocated, that is no heap storage area is currently allocated. Thus the first call to "NEW" or "VARNEW", or any subsequent calls to "NEW" or "VARNEW" with the current "free heap" areas in the heap blocks too small, will allocate a new heap storage block.

The size of the allocated heap storage block is determined by the value of the predeclared global integer variable HEAPREQUEST, which is initialized to a value of 4 at program startup. The value of HEAPREQUEST is multiplied by 1024 (i.e K-byte) and will be the size of the allocated heap storage block unless the size required from the "NEW" or "VARNEW" call is greater in which case that size value will be used.

If the value of HEAPREQUEST is less than or equal to zero then the size of the allocated heap storage block will be the largest memory fragment the operating system can grab.

As shown in the previous paragraphs the "heap storage" is arranged as a number of heap storage blocks, which has a limit of 28 blocks, which are allocated dynamically as required. The size of these blocks are determined by the value of HEAPREQUEST at the time the block is allocated.

Also, when the procedures "DISPOSE", "VARDISPOSE", or "RELEASE" cause the heap storage blocks lastly allocated to become completely unused, that is no area within the block is currently allocated to a pointer, the heap storage blocks will be returned to the operating system.

The following pages describe the HEAP storage management procedures.

DISPOSE

DECLARATION:

```
PROCEDURE DISPOSE(pointer {, variant1 {, variantN }})
```

PARAMETERS:

"pointer"	: must be a variable of pointer type.
"variant1" ... "variantN"	: If the "pointer^" was defined using the variant form of "NEW" then these variants must match exactly the variants used in the "NEW" call. See the "NEW" procedure in this chapter.

FUNCTION:

This procedure is the counterpart to the "NEW" procedure, it deallocates the storage for the "pointer^" in the heap storage, and assigns the value of "NIL" to the "pointer" variable. The heap area freed by this procedure is put on a "free heap" linked list so that a "NEW" or "VARNEW" call can reuse it.

EXAMPLE PROGRAM:

```
PROGRAM dispose;
TYPE
   arry = ARRAY [1..100] OF integer;
VAR
   ar    : ^arry;
BEGIN
new(ar);
fillchar(ar^, sizeof(arry), 0);
dispose(ar);
END.
```

MARK

DECLARATION:

```
PROCEDURE MARK(pointer)
```

PARAMETERS:

"pointer" : must be a variable of pointer type.

FUNCTION:

This procedure is provided for compatability with some of the older pascal compilers which didn't provide the "DISPOSE" procedure. A call to this procedure stores the current top-of-heap pointer value into the "pointer" variable. Along with the companion procedure "RELEASE", which is described in this charter, "MARK" provides a simple heap storage management scheme. As a rule, your program should only use "NEW" and "DISPOSE" (or also "VARNEW" and "VARDISPOSE") or only use "NEW", "VARNEW", "MARK", and "RELEASE" to manage heap storage. "MARK" is not part of the standard language definition and should be phased out of any existing programs where possible.

EXAMPLE PROGRAM:

```
PROGRAM mark;
VAR
   ptr  : ^integer;
   ary  : ARRAY [1..100] OF ^integer;
   i    : integer;
BEGIN
mark(ptr); {save top-of-heap value}
FOR i := 1 TO 100 DO
   BEGIN
   new(ary[i]);
   ary[i]^ := i;
   END;
release(ptr); {restore top-of-heap to saved value}
END.
```

NEW

DECLARATION:

```
PROCEDURE NEW(pointer {, variant1 {, variantN }})
```

PARAMETERS:

"pointer" : must be a variable of pointer type.

"variant1"
...
"variantN" : If the "pointer^" is of type RECORD that contains variants, then these variants may be specified to allocate only enough heap area to store the variant record. If the variants are not specified then enough heap area is allocated to store the largest variant record.

FUNCTION:

This procedure dynamically allocates storage for "pointer^" from the "heap storage", and assigns the address of that allocation to the "pointer" variable. The "free heap" linked list is searched for an area to fit the "pointer^", if one can't be found a new heap storage block will be allocated.

EXAMPLE PROGRAM:

```
PROGRAM new;
TYPE
   weight_class = (slim, reg, big);
   sample_rec   = RECORD
                  age    : integer;
                  CASE ismale : boolean OF
                   true  : (shoe_size : real);
                   false : (ring_size : integer;
                            CASE weight : weight_class OF
                             slim : (i  : integer);
                             reg  : (r  : real);
                             big  : (u  : linteger));
                  END;
VAR
   p1, p2, p3, p4 : ^sample_rec;
BEGIN
heaprequest := 1; {set heap block grap size to 1k-byte}
new(p1);
new(p2, true);
```

NEW (continued)

```
new(p3, false);
new(p4, false, big);
END.
```

PROGINFO

DECLARATION:

```
PROCEDURE PROGINFO(packet)
```

PARAMETERS:

"packet" : must be a variable with a size of 22 bytes. This variable is usually declared as follows :

```
VAR
   packet : RECORD
            freeheapblocks       : integer;
            totalheapallocated   : linteger;
            freeheaptotal        : linteger;
            maximumheapallocate  : linteger;
            stackused            : linteger;
            stackfree            : linteger;
            END;
```

FUNCTION:

This procedure returns current HEAP and STACK information in the variable "packet" as follows:

"freeheapblocks" is set to the number of heap storage blocks that are currently unused. The maximum number of heap storage blocks is 28.

"totalheapallocated" is set to the total amount of "heap storage" currently allocated in the heap storage blocks.

"freeheaptotal" is set to the total amount of heap areas unallocated within the heap storage blocks. "NEW" or "VARNEW" will try to use these areas before allocating a new heap storage block.

"maximumheapallocate" is set to the size of the largest unallocated heap area on the "free heap" linked list. Any "NEW" or "VARNEW" call that tries to allocate a pointer whose size is less than or equal to this value will allocate the pointer from the "free heap" list.

"stackused" is set to the current amount of stack used by the program. This stack includes the global stack and any local stacks used by nested calls to procedures and functions.

PROGINFO

"stackfree" is set to the current amount of stack unused by the program.

EXAMPLE PROGRAM:

```
PROGRAM proginfo(output);
VAR
   packet : RECORD
            freeheapblocks      : integer;
            totalheapallocated  : linteger;
            freeheaptotal       : linteger;
            maximumheapallocate : linteger;
            stackused           : linteger;
            stackfree           : linteger;
            END;
   ptr    : ARRAY [1..100] OF ^linteger;
   i      : integer;
BEGIN
FOR i := 1 TO 100 DO
   new(ptr[i]);
FOR i := 5 TO 10 DO
   dispose(ptr[i]);
proginfo(packet);
writeln('Free heap blocks =', packet.freeheapblocks);
writeln('Total heap allocated =', packet.totalheapallocated);
writeln('Free heap total =', packet.freeheaptotal);
writeln('Maximum heap allocate =', packet.maximumheapallocate);
writeln('Stack used =', packet.stackused);
writeln('Stack free =', packet.stackfree);
END.
```

RELEASE

DECLARATION:

```
PROCEDURE RELEASE(pointer)
```

PARAMETERS:

"pointer" : must be a variable of pointer type.

FUNCTION:

This procedure is the companion to the "MARK" procedure, it sets the "top-of-heap" pointer to the value of the "pointer" variable. The "pointer" variable must have been set using the "MARK" procedure. Any pointer allocated using the "NEW" or "VARNEW" procedure after the "MARK(pointer)" statement will become deallocated when the "RELEASE(pointer)" statement is executed.

EXAMPLE PROGRAM:

```
PROGRAM mark;
VAR
   ptr  : ^integer;
   ary  : ARRAY [1..100] OF ^integer;
   i    : integer;
BEGIN
mark(ptr); {save top-of-heap value}
FOR i := 1 TO 100 DO
   BEGIN
   new(ary[i]);
   ary[i]^ := i;
   END;
release(ptr); {restore top-of-heap to saved value}
END.
```

VARNEW

DECLARATION:

PROCEDURE VARNEW(pointer, size)

PARAMETERS:

"pointer" : must be a variable of pointer type.

"size" : a linteger expression.

FUNCTION:

This procedure dynamically allocates "size" number of bytes of storage from the "heap storage", and assigns the address of that allocation to the "pointer" variable. The "free heap" linked list is searched for an area to fit the "size", is one can't be found a new heap storage block will be allocated. This procedure provides a sophisticated memory management capability, but should only be used by those programmers who thoroughly understand heap storage and its use. Note that the "size" parameter overides the implied size of the "pointer^" structure.

With this procedure a programmer can simulate conformant arrays as well as greatly reduce the amount of memory used for some types of "heap storage" allocation, particularly for string data types.

EXAMPLE PROGRAM:

```
PROGRAM varnew(input, output);
TYPE
   large_array  = ARRAY [1..10000] OF boolean;
   maxstring    = string[255];
   pline_record = ^line_record;

   line_record  = RECORD
                    next : pline_record;
                    line : ^maxstring;
                    END;
VAR
   newline      : maxstring;
   linehead     : pline_record;
   work         : pline_record;
   barray       : ^large_array;
   count        : integer;
```

VARNEW (continued)

```
BEGIN
count := 0;
linehead := NIL;
WHILE NOT eof DO
   BEGIN
   readln(newline);
   count := count + 1;
   new(work);
   WITH work^ DO
      BEGIN
      varnew(line, length(newline) + 1);
      {allocate enough heap to store store newline. Note the
       "+ 1" because string variable has a 1 byte length field}

      line^ := newline; {assign newline to string}
      {be absolutely sure never to assign a larger item
       than the declared size in a varnew call or the
       heap structure will be destroyed}

      next := linehead;
      END;
   linehead := work;
   END;

varnew(barray, count); {allocate only count items in array}
fillchar(barray^, count, false);

vardispose(barray, count);

WHILE linehead <> NIL DO
   BEGIN
   WITH linehead^ DO
      BEGIN
      writeln(line^);
      vardispose(line, length(line^) + 1);
      {dispose of allocated line^, use size of stored string}
      work := next;
      END;
   dispose(linehead);
   linehead := work;
   END;

END.
```

VARDISPOSE

DECLARATION:

```
PROCEDURE VARDISPOSE(pointer, size)
```

PARAMETERS:

"pointer"	: must be a variable of pointer type.
"size"	: a linteger expression.

FUNCTION:

This procedure is the counterpart of the "VARNEW" procedure, it deallocates the storage for "pointer^" in the "heap storage", and assigns the value of "NIL" to the "pointer" variable. The heap area freed by this procedure is put on a "free heap" linked list so that a "NEW" or "VARNEW" call can reuse it. If disposing of "pointer" causes the last heap storage block(s) to become unallocated, then those heap storage block(s) will be returned to the operating system.

An error will be generated if the "size" parameter is negative or isn't in range of the "size" parameter that was passed to the "VARNEW" call that created the "pointer".

EXAMPLE PROGRAM:

```
PROGRAM vardispose(input, output);
TYPE
   large_array  = ARRAY [1..10000] OF boolean;
   maxstring    = string[255];
   pline_record = ^line_record;

   line_record  = RECORD
                  next : pline_record;
                  line : ^maxstring;
                  END;
VAR
   newline      : maxstring;
   linehead     : pline_record;
   work         : pline_record;
   barray       : ^large_array;
   count        : integer;
```

VARDISPOSE (continued)

```
BEGIN
count := 0;
linehead := NIL;
WHILE NOT eof DO
   BEGIN
   readln(newline);
   count := count + 1;
   new(work);
   WITH work^ DO
      BEGIN
      varnew(line, length(newline) + 1);
      {allocate enough heap to store store newline. Note the
       "+ 1" because string variable has a 1 byte length field}

      line^ := newline; {assign newline to string}
      {be absolutely sure never to assign a larger item
       than the declared size in a varnew call or the
       heap structure will be destroyed}

      next := linehead;
      END;
   linehead := work;
   END;

varnew(barray, count); {allocate only count items in array}
fillchar(barray^, count, false);

vardispose(barray, count);

WHILE linehead <> NIL DO
   BEGIN
   WITH linehead^ DO
      BEGIN
      writeln(line^);
      vardispose(line, length(line^) + 1);
      {dispose of allocated line^, use size of stored string}
      work := next;
      END;
   dispose(linehead);
   linehead := work;
   END;

END.
```

This Page Intentionally Blank

The following pages describe the OS-9/68000 Pascal miscelaneous routines. All routines except for the routines "PACK" and "UNPACK" are extensions to the ISO standard.

ADDRESS

DECLARATION:

```
FUNCTION address(variable-access) : linteger
```

PARAMETERS:

"variable-access" : is the name of any variable declared within the program. It can be a simple name, or a pointer, array, or record reference.

FUNCTION:

This function returns the actual 32-bit real memory address for the referenced "variable-access".

EXAMPLES:

```
laddr := address(simple);
laddr := address(complex^.r[5].q);
laddr := address(input); {address of file input's fcb}
```

EXIT

DECLARATION:

```
PROCEDURE EXIT(routine)
```

PARAMETERS:

"routine"	: must be the name of a visable procedure or function or the reserved word "PROGRAM".

FUNCTION:

This procedure allows premature exiting of the current routine, a higher level routine within nested routine calls, or the entire program.

IF EXIT is called and the "routine" parameter is the reserved word "PROGRAM", the program is terminated immediately and control returns to the OS-9 operating system.

If EXIT is called and the "routine" parameter is the name of a procedure or function, then the routine activation list (i.e. the current calling structure of who called whom) is searched from the current routine to the outer-block routine for the named routine "routine". When the named routine "routine" is found program execution will continue to the caller of that routine, that is the named routine will be exited. An error will be generated if the named routine "routine" can't be found on the routine activation list. Note: if the named routine was a function and no assignment was made to that function before it was exited then it will return an undefined value.

Also note that any routines exited by this call to EXIT, including the outer-block, will not automatically close any open files opened within those routines.

EXIT (continued)

EXAMPLE PROGRAM:

```
PROGRAM exittest(input, output);
VAR
   r     : real;

PROCEDURE check_io_errors(VAR f : text);
   VAR
      paserr  : integer;
      os9err  : integer;
   BEGIN
   paserr := ioresult(f);
   os9err := iosyserr(f);
   IF paserr <> 0 THEN
      writeln('Pascal error #', paserr:1);
   IF os9err <> 0 THEN
      writeln('OS-9 error #', os9err:1);
   IF (paserr <> 0) OR (os9err <> 0) THEN
      exit(PROGRAM);
   END;

BEGIN
ioabort(input, false);
REPEAT
   readln(r);
   check_io_errors(input);
UNTIL false;
END.
```

HALT

DECLARATION:

```
PROCEDURE HALT(value)
```

PARAMETERS:

"value" : a linteger expression.

FUNCTION:

When this procedure is executed, the program will be aborted and a halt error message will be displayed indicating that a programmed halt was executed and showing the value of the "value" parameter.

EXAMPLES:

```
halt(0);
halt(iosyserr(input));
halt($1000_0000);
halt(-1);
```

OS9

DECLARATION:

```
PROCEDURE OS9(packet, os9function)
```

PARAMETERS:

"packet"	: must be a variable with a size of 64 bytes.
"os9function"	: an integer expression.

FUNCTION:

This procedure is used to call the OS-9 operating system directly from a pascal program.

When this procedure is executed the following occurs, the 68000 registers d0-d7,a0-a6 are loaded from the register packet, then the OS-9 system call defined by the "os9function" value is performed, if an error occured on the call, as indicated by the carry flag being set, a boolean type variable in the register packet will be set, and lastly the 68000 registers d0-d7,a0-a6, which were possibly modified by the system call, are copied back into the register packet.

The register packet definition should be defined as shown in the folling program example. This allows you to access any 68000 register by byte (d0.b), word (d0.w), or long (d0.l).

Refer to the OS-9/68000 Operating System Technical Manual for information on operating system calls.

OS9 (continued)

EXAMPLE PROGRAM:

```
PROGRAM os9(input, output);
CONST
   i$delete  = $87;  {I$Delete system call value}
TYPE
   register  = RECORD  {definition for 68000 register}
               CASE integer OF
              0 : (l         : linteger); {long word access}
              1 : (highword  : integer;
                   w         : integer);  {word access}
              2 : (byte3     : char;
                   byte2     : char;
                   byte1     : char;
                   b         : char);     {byte access}
               END;

   os9packet = RECORD  {OS9 procedure packet definition}
               error      : boolean;  {set true if carry bit
                                       set after call to OS9}
               reserved   : integer;  {unused}
               d0         : register; {68000 d0 register image}
               d1         : register; {68000 d1 register image}
               d2         : register; {68000 d2 register image}
               d3         : register; {68000 d3 register image}
               d4         : register; {68000 d4 register image}
               d5         : register; {68000 d5 register image}
               d6         : register; {68000 d6 register image}
               d7         : register; {68000 d7 register image}
               a0         : register; {68000 a0 register image}
               a1         : register; {68000 a1 register image}
               a2         : register; {68000 a2 register image}
               a3         : register; {68000 a3 register image}
               a4         : register; {68000 a4 register image}
               a5         : register; {68000 a5 register image}
               a6         : register; {68000 a6 register image}
               END;

VAR
   name      : string;
   os9pkt    : os9packet;
```

OS9 (continued)

```
BEGIN {outer block}
writeln('Enter a list of files to delete.');
writeln('Hit <eof> when done.');
WHILE NOT eof DO
   BEGIN
   readln(name); {get file name}
   name := concat(name, chr(0)); {terminate name with null}
   os9pkt.a0.l := address(name[1]); {set a0.l to address of first
                                     character in name}
   os9pkt.d0.b := chr(3); {set d0.b to user read/write}
   os9(os9pkt, i$delete); {call os9 to delete file}
   IF os9pkt.error THEN
      writeln('OS9 error #', os9pkt.d1.w:1,
              ' occured on deletion of file ', name);
   END;
END.
```

PACK

DECLARATION:

```
PROCEDURE PACK(a, i, z)
```

PARAMETERS:

"a"	: an unpacked array variable.
"i"	: an expression with the same type as "a"'s index.
"z"	: a packed array variable with the same component type as "a".

FUNCTION:

The PACK procedure copies components of an unpacked array variable to a packed array variable. The number of components in "a" must be greater than or equal to the number of components in "z". "PACK(a, i, z)" assigns the components of "a", starting with "a[i]" to the array "z", starting with "z[low-bound-of-z]", until all components in "z" are filled. Packing need not start with the first component of array "a"; for example, "PACK(a, 5, p)" packs components "a[5]" thru "a[24]" into components "p[1]" thru "p[20]".

Note: The OS-9/68000 Pascal compiler packs all data to the byte level and the packed attribute does not cause any savings in data storage. Thus this procedure is not useful for any data space saving and is included only for conforamability with the ISO standard.

EXAMPLE PROGRAM:

```
PROGRAM pack;
VAR
   a     : ARRAY [1..24] of 0..15;
   p     : PACKED ARRAY [1..20] of 0..15;
BEGIN
fillchar(a, sizeof(a), 0);
pack(a, 5, p);
END.
```

SIZEOF

DECLARATION:

```
FUNCTION SIZEOF(item) : integer
```

PARAMETERS:

"item" : can be the name of a declared variable or a declared type. Only simple names can be used, no indexing, field referencing, or pointer dereferencing can be used.

FUNCTION:

The result of this function is an integer value indicating the number of bytes of storage which is required to contain the "item".

EXAMPLE PROGRAM:

```
PROGRAM sizeof;
TYPE
   large_array = ARRAY [1..10000] OF boolean;
VAR
   b    : large_array;
BEGIN
fillchar(b, sizeof(large_array), true);
END.
```

SYSTIME

DECLARATION:

```
PROCEDURE systime(result, timetype)
```

PARAMETERS:

"result" : must be a variable with a size of 10 bytes.

"timetype" : an integer expression.

FUNCTION:

This procedure will return the current system date and time in the variable "result" based upon the value of "timetype". If the value of "timetype" is 0 then the date and time will be returned in GREGORIAN format and the variable "result" should be defined as follows:

```
VAR
   result  : RECORD
             dayofweek : integer; {0=sunday to 6=saturday}
             zero      : char;    {always 0}
             hour      : char;
             minute    : char;
             second    : char;
             year      : integer;
             month     : char;
             day       : char;
             END;
```

Otherwise if the value of "timetype" is 1 then the date and time will be returned in JULIAN format and the variable "result" should be defined as follows:

```
VAR
   result  : RECORD
             dayofweek : integer;  {0=sunday to 6=saturday}
             seconds   : linteger; {since midnight}
             day       : linteger; {julian daynumber}
             END;
```

Refer to the F$Time system call in the OS-9/68000 Operating System technical manual for more information.

SYSTIME (continued)

EXAMPLE PROGRAM:

```
PROGRAM systime(output);
VAR
   time     : RECORD
              dayofweek : integer; {0=sunday to 6=saturday}
              zero      : char;    {always 0}
              hour      : char;
              minute    : char;
              second    : char;
              year      : integer;
              month     : char;
              day       : char;
              END;
   work     : integer;
BEGIN
systime(time, 0);
write('Todays date is ', ord(time.month):1, '/');
write(ord(time.day) div 10:1, ord(time.day) mod 10:1, '/');
writeln(time.year:1);
END.
```

UNPACK

DECLARATION:

```
PROCEDURE UNPACK(z, a, i)
```

PARAMETERS:

"a"	: an unpacked array variable.
"i"	: an expression with the same type as "a"'s index.
"z"	: a packed array variable with the same component type as "a".

FUNCTION:

The UNPACK procedure copies components of packed array variable to an unpacked array variable. The number of components in "a" must be greater than or equal to the number of components in "z". "UNPACK(z, a, i)" assigns the components of "z", starting with "z[low-bound-of-z]" to the array "a", starting with "a[i]", until all components in "z" are used.

Note: The OS-9/68000 Pascal compiler packs all data to the byte level and the packed attribute does not cause any savings in data storage. Thus this procedure is not useful for any data space saving and is included only for conforamability with the ISO standard.

EXAMPLE PROGRAM:

```
PROGRAM unpack;
VAR
   a     : ARRAY [1..10] of char;
   p     : PACKED ARRAY [1..10] of char;
   i     : integer;

PROCEDURE process_components(VAR ch : char);
   BEGIN
   ch := chr(0);
   END;

BEGIN
FOR i := 1 to 10 do
   p[i] := chr(i);
unpack(p, a, 1);
```

UNPACK

```
FOR i := 1 to 10 do
   process_components(a[i]);
END.
```

APPENDIX A
COMPILER ERROR MESSAGES

Following is the text of OS-9/68000 Pascal compiler error messages. Messages consist of a number, followed by a colon, followed by a description. Following the text of many of the error messages is a brief description of what the compiler was scanning or looking for which caused the error to occur, this will in many cases aid the user in correcting his program. In a few instances, further information is also given which indicates the most likely error and/or correction associated with the message.

1: Simple type expected.
*When looking for a simple type, a valid starting token could not be found, i.e. a left parenthesis, plus sign, minus sign, integer constant, real constant, string constant, or an identifier.
*When looking for a simple type, an identifier was found which has no valid type information yet determined.

2: Identifier expected.
*When scanning an enumeration declaration, the identifier found was not an identifier as part of the required identifier list.
*When scanning a field list within a record, a comma was found, but the token after was not an identifier.
*With a record declaration, CASE was found, but the token after is not an identifier.
*When scanning a type declaration, up arrow was found, but the token after is not an identifier.
*When scanning CONST declarations, expected to find an identifier being declared.
*When scanning TYPE declarations, expected to find an identifier being declared.
*When scanning VAR declarations, expected to find an identifier begin declared.
*When scanning a procedure or function parameter list, found PROCEDURE or FUNCTION, but the token after is not an identifier.
*When scanning a function declaration, ":" was found, but it was not followed identifier naming the type of result.
*When scanning a procedure or function parameter list, found a list of identifiers followed by a colon, but the token after is not an identifier naming type of identifier(s).
*PROCEDURE or FUNCTION is not followed by an identifier giving routine name.
*A period is found indicating reference to a field within a record, but the period is not followed by an identifier naming the field.
*When scanning a parameter list, expected to find an identifier naming a parameter.
*FOR is not followed by an identifier naming the control variable.
*WITH is not followed by an identifier naming the record.
*PROGRAM is not followed by an identifier naming the program.
*When scanning the list of file names after PROGRAM, an identifier naming a file was expected.
**Possible common programming errors which might trigger this message include:
-An extra comma was found in a list.
-A reserved word is used as a variable name.

3: PROGRAM expected.
*PROGRAM must be the first token in a program outside of any preceding comments.

4: Right parenthesis expected.
*Expected a right parenthesis here to balance a previous left parenthesis.

5: Colon expected.
*When scanning a field list within a record declaration, expected a colon to terminate list.
*When scanning a case selection either in a statement or in a record declaration, expected a colon to terminate selection list.
*When scanning VAR declarations, expected a colon to terminate a list of identifiers being declared.
*Label number preceding a statement is not terminated by a colon.
*When scanning a function declaration, expected a colon terminator to introduce the function type result.

6: Unexpected symbol found.
*Token scanned is not a valid next token for the type of scanning being performed. This is a sort of catch all error for whenever the scanned token doesn't fall within the set of tokens that the scanner knows how to handle next. Frequently, this error is preceded by other error numbers, in this case, the error usually means that due to the other errors for this line the scanner has lost track of what should be happening next in the sentence. If there are no preceding errors for this sentence, then the token is simply invalid for the syntax of the statement at this point.
**Possible common programming errors which might trigger this message include:
-A semicolon precedes ELSE in an IF statement.
-This or previous statement is missing a required semicolon;
-An extra END is encountered.
-DO is used instead of BEGIN.
-Spaces are used within an identifier name.
-A keyword has been misspelled.
-A comment is malformed.
-A string or character constant is malformed.
-A malformed double character operator is found such as => instead of >= or = instead of := or one or more spaces or end of lines occurs between the first and second character.
-A comma is missing between elements in a list.
-An equal sign is used instead of a colon for declaring a record within a record.

7: Parameter list expected.
*When scanning a procedure or function declaration, an unexpected token was found which is not a left parenthesis which would introduce a parameter list, or the token is not one which would validly terminate the declaration.
*When scanning a statement, found a call to a procedure or function which requires a parameter list, but no list is found.
*When scanning a parameter list, found a semicolon which separates parameter items, but the next token is not a valid parameter list item.

8: OF expected.
*Within a record declaration, CASE followed by a tagfield is not followed by OF.
*While scanning a type declaration, ARRAY followed by bounds is not followed by OF.
*While scanning a type declaration, SET is not followed by OF.
*While scanning a type declaration, FILE is not followed by OF.
*While scanning a statement, CASE followed by an expression is not followed by OF.

9: Left parenthesis expected.
*When scanning a record declaration with a CASE part, a selection is found followed by a colon but is not followed by a left parenthesis which introduces the fields for the selection(s).
*When scanning a statement, a procedure or function call is found which requires a parameter list, but no left parenthesis was found which introduces the parameter list.

10: Type expected.
*A type declaration was expected and an identifier was found for which no type information is known.
*A type declaration was expected and did not find a valid first token, i.e. an up arrow, PACKED, ARRAY, RECORD, SET, FILE, a left parenthesis, a plus sign, a minus sign, an integer constant, a real constant, a string constant, or an identifier.
*When scanning a type declaration, found PACKED not followed by ARRAY, RECORD, SET, or FILE.
*When scanning a function declaration, found a colon introducing the result type, but no type or an unknown type is found following the colon.
**Possible common programming errors which might trigger this message include:
-Using a reserved word as a variable name.
-Using a colon instead of equal sign in a type declaration.

11: Left bracket expected.
*When scanning a type declaration, found ARRAY but not a left bracket introducing the array bounds.

12: Right bracket expected.
*Expected a right bracket to match a previously found left bracket.

13: END expected.
*When scanning a type declaration, found RECORD but not terminating END.
*When scanning a case statement, cannot find terminating END.
*BEGIN is missing its terminating END.

14: Semicolon expected.
*Looking for a semicolon terminator and didn't find one here.

15: Integer expected.
*When scanning a label declaration, expected an integer constant label number.
*GOTO is not followed by an integer constant label number.

16: Equal sign expected.
*When scanning a CONST declaration, identifier is not followed by an equal sign.
*When scanning a TYPE declaration, identifier is not followed by an equal sign.

17: BEGIN expected.
*A declaration part was scanned which is not part of an empty outer block, but the declaration part is not followed by BEGIN.

18: Invalid declaration part.
*A declaration part was scanned, but it is not followed by BEGIN or a program terminating period.

19: Field list expected.
*When scanning a record declaration, RECORD is not followed by an identifier introducing a field list or CASE.

20: Comma expected.
*For this procedure or function call another parameter is expected, and the separating comma is not found.
*When scanning the list of file names following PROGRAM, the file name is not followed by a comma or right parenthesis.

21: Program terminating period expected here.
*Last token in a Pascal program must be a period.

22: A 10 byte structure is expected.
*Scanning the standard procedure SYSTIME the first parameter is not a variable with a size of 10.

23: Expected a hex digit.

*When scanning a hexadecimal number, found the dollar sign which indicates the start of a hexadecimal number, but the current character being scanned should be and is not a hexadecimal digit (0 through 9, 'A' through 'F', or 'a' through 'f').

24: Double period expected.
*Subrange simple type is being scanned, a double period is expected to separate low range from high range.

25: Invalid ordering of declaration parts.
*Declarations are required to be in the standard order of LABEL, CONST, TYPE, VAR, and PROCEDURE or FUNCTION.

26: Comma or colon expected.
*When scanning a record declaration, found a fieldlist, but the next token in not a comma to separate identifiers or a colon to terminate identifier list.
*When scanning a parameter list, found a list of identifiers, but the next token is not a comma to separate identifiers or a colon to terminate identifier list.

27: Constant is out of range.
*Compile time check of allowable range for constant shows that the constant is out of range.

28: Identifier, VAR, PROCEDURE, or FUNCTION expected.
*When scanning a procedure or function parameter list, didn't find a valid next token.

29: Invalid scope of name due to previous use.
*An identifier is being defined which was previously referenced in this block.

30: Error in real constant, digit expected.
*A string of digits was found followed by a period (but not a double period), but no digit was found after the period, this is an invalid real number according to the language spec. You may need to put a zero after the period.
*A real number was found which ends with an "e" or "E" followed by an optional sign but not followed by a digit. After the "E" must come an integer exponent value.

31: String constant must be contained on a single source line.
*An opening single quote character was found which indicates the start of a string constant, but the matching ending single quote character was not found on the same line of source. All strings must be fully contained on a single line.

32: Integer constant exceeds range.
*An integer constant was scanned which has a value greater than 2,147,483,647.

33: Too many nested scopes of identifiers.
*When scanning a type declaration, found RECORD, but the symbol table cannot be further nested.
*When scanning a procedure or function declaration, the symbol table cannot be further nested.
*When scanning a WITH statement, the symbol table cannot be further nested.

34: Too many nested procedures and/or functions.
*PROCEDURE or FUNCTION found, but the symbol table cannot be further nested.

35: Defined function is not assigned a value.
*Within the scope of the function definition no assignment was made to the function identifier.
**All functions must return a value.

36: Too many errors detected for this source line.
*Ten or more errors were detected during the scanning of this source line. All errors past the ninth are ignored. Usually when this many errors are detected for a single line of source, it is because some other error has triggered secondary errors. If this is the case, correcting the primary error may get rid of the secondary errors.

37: Field width parameter negative.
*While scanning the standard procedures WRITE or WRITELN the field width parameter was found to be a negative constant.

38: Constant value must be greater than zero.
*An array reference using the <expression> FOR <count> language extension is scanned but <count> is zero or negative - it must be greater than or equal to one.

39: Duplicate identifier in PROGRAM statement parameters.
*Within the PROGRAM statement parameter list an identifier has been used more than once.

40: Can't use NIL in CONST declaration.
*NIL is a reserved word which is used in reference to pointers but does not have a value which may be assigned within a CONST declaration.

41: Expected an unpacked array.
*While scanning the standard procedures PACK or UNPACK an unpacked array type was expected but not found.

42: Expected a packed array.
*While scanning the standard procedures PACK or UNPACK a packed array type was expected but not found.

43: Undefined FORWARD procedures or functions found.
*A PROCEDURE or FUNCTION was declared with the attribute FORWARD, but the body of the procedure or function was not found by the time is was required to be found.

44: Integer or Linteger variable name expected.
*A variable of either integer or linteger type is expected here but wasn't found.

45: Lexical seperation error, number can't be terminated with a letter.
*A number token can not be terminated with an alphabetic letter.

46: Case selector is out of range of base type.
*Within a TYPE declaration a variant record entry has been declared with a value which is not in range of the record case determinator type. An example is to declare a record with a variant record whose selector is a subrange. One of the possible record formats then uses a label which is out of the selector variable's subrange.

47: A long integer is not allowed here.
*A linteger type was scanned and is not valid here.

48: A character type is expected here.
*While scanning the standard procedure SCAN a character type was expected but not found.

49: Selector can not be a long integer type.
*The selector of a CASE statement can not be a linteger type.

50: Constant expected.
*Looking for a constant, but did not find a valid token to start a constant, i.e. a plus sign, minus sign, integer constant, real constant, string constant, or an identifier.
**Possible common programming errors which might trigger this message include:
-Three periods in a row are found where a double period is intended.
-Finding a list when an array index specification is needed.

51: ':=' expected.
*A statement is being scanned, an identifier was found which is not a procedure or function name, therefore, it must be an assignment statement, but the next token is not ':='.
*A FOR statement is begin scanned, FOR was followed by an identifier name but was not further followed by ':='.
**Possible common programming errors which might trigger this message include:
-An equal sign is found where := is intended.
-An identifier is misspelled.

52: THEN expected.
*An IF statement is being scanned. IF was followed by a boolean expression but the next token is not THEN.

53: UNTIL expected.
*A repeat statement is being scanned, REPEAT followed by a statement list was found, but the terminating UNTIL was not found.

54: DO expected.
*A WHILE statement is being scanned. WHILE was followed by a boolean expression but the next token is not DO.
*A FOR statement is being scanned. FOR <ident>:=<expr> TO/DOWNTO <expr> was found, but the next token is not DO.
*A WITH statement is being scanned. WITH was followed by an identifier, but the next token is not DO.

55: TO or DOWNTO expected.
*A FOR statement is begin scanned. FOR <ident>:=<expr> was found, but the next token is not TO or DOWNTO.

56: '=' or '<>' expected.
*While scanning the standard procedure SCAN a token of '=' or '<>' was expected but not found.

57: A 64 byte structure is expected.
*While scanning the standard procedure OS9 the first parameter is not a variable with a size of 64.

58: Factor expected.
*Looking for a factor but didn't find a valid first token, i.e. an integer constant, real constant, string constant, identifier, left parenthesis, left bracket, or NOT.
**Possible common programming errors which might trigger this message include:
-A real constant is intended but there are no digits before the decimal point.
-A malformed string constant is found.
-A malformed double character operator is found (see error message 6 for examples).

59: Variable expected.
*When scanning a sentence, a variable reference followed by up arrow, period, or left bracket was expected.

60: Attempted to load or store the value of an address.
*While attempting to generate an icode instruction, a form of object reference was found which would require the loading or storing of an address as a value.

61: Expected a file name.
*A procedure or function call is being scanned which requires as a parameter the name of a file here.

62: A procedure or function name is expected.
*While scanning the standard procedure EXIT an identifier was found but it wasn't a procedure or function name.

64: Type size exceeds range.
*A type having a size greater than 32767 is being defined.

65: Can't return a structure containing a file component.
*A function result is not allowed to contain a file.

66: Can't use VARNEW/VARDISPOSE with a structure containing a file.
*A pointer containing a file component can only be created or destroyed with NEW or DISPOSE.

67: File types are not allowed in a variant.
*While scanning a record variant definition a file type was being defined which is not allowed.

68: A 22 byte structure is expected.
*While scanning the standard procedure PROGINFO a variable with a size of 22 was expected but not found.

69: Can't assign to a record or array containing a file component.
*The variable to the left of the ':=' is either an array or record which contains a file and this is not allowed.

70: Text file size declaration is out of range.
*While scanning the text buffer declaration the buffer size wasn't in the range of 1 thru 8192.

71: Can't pass structure containing a file component by value.
*An actual procedure or function parameter is being declared which contains a file component.

72: Variable size exceeds range.
*A VAR declaration section is being defined and it's size exceeds 32767.

73: Label not in range of 0 thru 9999.
*A label is being defined which is not in the range of 0 thru 9999.

74: Non ISO standard feature used.
*With the ISO only compilation flag set a non iso feature was scanned.

75: Expected a simple type or set type.
*While scanning a standard procedure parameter a simple type or set type was expected but not found.

76: Too many $PUSH's without $POP's.
$PUSH compiler pseudo op has been nested too deep.

nd youlthe appropriate license forms. Please feel free to contact

77: Illegal $SET option.
*While scanning the compiler $SET command an illegal option was found.

78: No path name specified on $INCLUDE command line.
*While scanning the compiler $INCLUDE command no path name was found.

79: Linteger variable expected.
*While scanning the standard procedure RANDOM a linteger type variable was expected but not found.

80: GOTO into outer code block illegal with seperate compilation enabled.
*While scanning a GOTO statement the destination was found to be in the outer code block but the seperate compilation flag is set which makes this reference illegal.

81: A VECTOR can be used only on a "PACKED ARRAY OF CHAR" or STRING type.
*When using the array index vector slice operation, the data type the vector operation was being performed on wasn't of the type "PACKED ARRAY OF CHAR" or STRING.

97: Include files nested too deep.
*Unable to open include file because include file nesting is at its maximum level.

98: OS-9 error on include file open.
*An OS-9 error occured while trying to open an include file.

99: Real constant exceeds range.
*The real number constant exceeds the value that can be represented by the IEEE double precision format.

100: String or packed character array expected.
*An iso string-type is expected. This includes a character string enclosed in quotes or a packed array of char with a low bound of 1.
*An expression of type string is expected.

101: Identifier is already declared.
*A new identifier is being declared, but the identifier already has been declared.
**Possible common programming errors which might trigger this message include:
-An identifier is misspelled.
-An identifier is being used both as a simple type and as an enumeration identifier.

102: Low bound exceeds high bound.
When scanning a subrange the low bound of the range is found to be numerically higher than the high bound.

103: Identifier is not of the appropriate class.
*The identifier named does not have valid attributes for use here.
**Possible common programming errors which might trigger this message include:
-A string or character constant is malformed.
-Previous errors were found in the declaration of the identifier.

104: Identifier is not declared.
*The identifier found has not been previously defined and this is not a forward pointer declaration.
**Possible common programming errors which might trigger this message include:
-An identifier is misspelled.
-A keyword is being used as an identifier name.
-A string or character constant is malformed.

105: Sign is not allowed here.
*When looking for a constant, a sign followed by an identifier name was found, but the identifier is not equivalent to a real or integer constant.

106: A number is expected.
*When looking for a constant, a non-string was found which does not resolve to a numeric value.

107: Incompatible subrange types.
*When scanning a simple type, found a subrange, but the token to the right of the '..' is not compatible with the token to the left.

108: FILE is not allowed here.
*When scanning a type declaration, found FILE OF FILE. Files of file are invalid.

109: Type must not be real.
*When scanning a subrange declaration, found a real range constraint. Only integer ranges are allowed.
*When scanning a case discriminant (tagfield identifier), found a real-type identifier.
*When scanning array declaration, found ARRAY [REAL].

110: Tagfield must be a scalar or subrange.
*When scanning a case discriminant (tagfield) identifier, found an identifier which was not a scalar-type or subrange-type.

111: Incompatible with tagfield type.
*When scanning a case selection list, the type of the selection value is Incompatible with the type of the case discriminant (tagfield).

113: Index type must be a scalar or a subrange.
*When scanning an array bounds declaration, the type of the bound given is not a scalar-type or a subrange-type.

114: Base type must not be real.
*When scanning a set type declaration, found SET OF REAL.

115: Base type must be a scalar or a subrange.
*When scanning a set type declaration, the type of the set being declared is not a scalar-type or a subrange-type.

116: Error in type of standard procedure parameter.
*When scanning a call of a standard routine, the parameter found is not a valid type. See the user manual section on standard functions and procedures for a description of valid parameters.

117: Unsatisfied forward reference.
*The list of identifiers following were previously declared to be forward but their actual declarations were not found.

119: Procedure was declared FORWARD, repetition of parameter list not allowed.
*When scanning a procedure declaration and found a left parenthesis which introduces a parameter list but the routine was previously declared as forward and the previous declaration is the only place the the parameter list can be declared.

120: Function result type cannot be a file, tagfield, or variant.
*When scanning a function declaration and a colon was found introducing a type for the function result and it is a file, tagfield, or variant which are illegal.

121: FILE parameter must not be passed by value.
*When scanning a procedure or function parameter list and encountered attempt to pass a file by value, files can only be passed by name.

122: Function was declared forward, repetition of parameter list is not allowed.
*When scanning a function declaration and encountered a left parenthesis which introduces a parameter list but the function was previously declared as forward and the previous declaration is the only place that the parameter list can appear.

123: Missing result type in FUNCTION declaration.
*When scanning a function declaration and did not find a colon which introduces the result type of the function.

124: F-format allowed for real values only.
*When scanning parameter list for WRITE or WRITELN call and found extraneous colon introducing a decimal width constraint but such a constraint is valid only for printing a real value and the item being printed is not real-type.

125: Not enough parameters given.
*When scanning a call to READ or WRITE and no parameters are given, either give a file name and/or a list of items or use READLN or WRITELN.

126: Number of parameters does not agree with the declaration.
*When scanning a procedure or function call and the number of parameters given does not agree with the declaration. For user routines look at the previous declaration. For standard routines see the section in the user manual describing standard procedures and functions.

128: Result type of parameter function does not agree with its declaration.
*When scanning a procedure or function call and the parameter being passed is a function which does not have the correct result type.

129: Operands are of Incompatible types.
*Left argument of IN operator is not compatible with the right argument.
*Left argument of a relational operator is not compatible with the right argument.
*Left argument of the ':=' operator is not compatible with the right argument.
**Possible common programming errors which might trigger this message include:
-Attempting to assign a real result to an integer (use TRUNC or ROUND).
-Using a slash character instead of DIV.
-Assigning a string constant to a non-packed array of char
-Assigning a string constant to a packed array of char with a lower bound not of 1
-An identifier is misspelled.

130: Expression must be a SET type.
*Right argument of IN operator must be a set-type.

131: Only equality and inequality test allowed on pointers.
*Pointers can only be compared as '=' or '<>'.

132: Strict inclusion test not allowed.
*Between sets only the following relationals are allowed: '<=', '>=', '=', '<>'.

133: Comparison of file, record, or array types are not allowed.
*No relational operators can be applied to file, record, or array types.

134: Illegal type of operand(s).
*Asterisk can only be used to multiply integers and/or reals or to intersect sets.
*Slash can only be used to divide integers and/or reals.
*DIV can only be used to divide integers.
*MOD can only be used to divide integers.
*AND can only be used to disjuct booleans.
*OR can only be used to conjunct booleans.
*Plus can only be used as a unary sign for reals or integers or to add reals and/or integers or to union sets.
*Minus can only be used as a unary sign for reals or integers or to subtract reals and/or integers or to intersect sets.
**Possible common programming errors which might trigger this message include:
-Errors were encountered in the previous declaration of the identifier.
-Malformed double character operator (see error message 6 for examples).

135: Type of operand must be BOOLEAN.
*Expression after NOT must be boolean type.

136: Element type of a set must be a scalar or a subrange.
*After a left bracket introducing a set construction was found, a token is found which is not a scalar or subrange and thus cannot be a member of a set.

137: Element type is not compatible with the set.
*An element of a set was scanned but the element is not a member of the base set.

138: Attempt to index a non-array variable.
*A left bracket was scanned introducing an indexing expression but the item to the left of the left bracket is not an array-type.

139: Index type is not compatible with the declaration.
*Scanned an index expression but the type of the expression is not compatible with the array object being indexed.
**Possible common programming errors which might trigger this message include:
-Errors detected during previous declaration of identifier.
-Identifier is misspelled.

140: Attempt to select a field of a non-record variable.
*A period was scanned introducing a field reference but the item to the left of the period is not a record-type.

141: Type of variable must be a FILE or a POINTER.
*An up arrow was scanned introducing a pointer or file reference but the item to the left of the up arrow is not a file-type or a pointer-type.

142: Illegal parameter substitution.
*When scanning a procedure or function call and the parameter scanned does not have the same type as declared in the function/procedure declaration. For user routines, see the previous declaration. For standard routines, see the section in the user manual describing standard procedures and functions.

143: Illegal type of FOR control loop variable, must be a scalar or subrange.
*Scanning a FOR statement and the identifier following FOR is not scalar-type or subrange-type.

144: Illegal type of expression.
*Scanning a case statement and the expression following CASE is not a non-real scalar-type.

145: Type conflict.
*Scanning a FOR statement, found FOR <ident>:=<expr> and possibly TO/DOWNTO <expr> but <expr> is not compatible with the type of <ident>.
**Possible common programming errors which might trigger this message include:
-Errors detected during previous declaration of identifier.
-Identifier is misspelled.

146: Assignment of files is not allowed.
*Item to the left of ':=' in an assignment statement cannot be file-type.

147: Case selection type is Incompatible with selecting expression.
*Scanning a case statement, found a selection value but the type of the value is not compatible with the type of the expression following CASE.

148: Subrange bounds must be scalar.
*Scanning a subrange simple type and a string is found.

149: Index type must not be INTEGER or LINTEGER.
*Scanning a type declaration and found ARRAY [INTEGER].
*Scanning a type declaration and fornd ARRAY [LINTEGER].

150: Assignment to a standard function is not allowed.
*Scanning an assignment statement and object to the left of ':=' is a name of a standard function.

151: Assignment to a formal function is not allowed.
*Attempt to assign a value to a formal <ident> which is a function name.

152: No such field in this record.
*Scanned a period which introduces an identifier which is a field name within a record but the identifier does not name a valid field name within the referenced record, check spelling of identifier or previous declaration of record.

153: String size declaration is out of range.
*While scanning a string declaration, the size declaration of the string was found not to be in the range of 1 thru 255.

154: Actual parameter must be a variable name.
*Scanning a procedure or function call and the parameter being scanned must be passed by name but the token found is not a variable name.

155: FOR control variable must not be declared at an intermediate level.
*Scanning a FOR statement and the identifier of the control variable is not declared local to this block. Pascal language specification says that the control variable of a FOR statement must not be a formal parameter or be a variable declared at a different level than the block containing the FOR statement.

156: Multidefined case label.
*Scanning a case statement and a case selection or OTHERWISE appears more than once.

158: Missing corresponding variant declaration.
*Scanning call to standard procedure NEW or DISPOSE and token after a comma does not name a valid value of a variant. Check spelling of token, correct nesting of variant specifications for this call, or previous record declaration.

160: Previous declaration was not FORWARD.
*Scanning a procedure or function declaration but routine has been previously declared and did not have FORWARD attribute.

161: Attempt to declare FORWARD again.
*Scanning a procedure or function declaration and found FORWARD or EXTERNAL but routine has been previously declared with one of these attributes.

162: Parameter must be a valid variant value.
*Scanning call to standard procedure NEW or DISPOSE and token after a comma is not a valid tagfield-type.

163: This standard procedure cannot be passed by name.
*Cannot pass a standard procedure as a parameter.

164: Must be a procedure declared in your program.
*While scanning the standard procedure EXIT a procedure name within scope was expected but not found.

165: Label is already defined.
*Scanning a statement, found a label preceding the statement but this label has already been encountered in this block.

166: Label is already declared.
*Scanning a LABEL declaration and label number scanned has already been declared a label in this block.

167: Undeclared label.
*Scanning a GOTO statement and label number scanned is unknown in this block.
*Scanning a statement, found a label preceding the statement but this label has not been declared in a LABEL declaration for this block.

168: Undefined label.
*The label number(s) following were declared in a LABEL declaration for this block but were never encountered.

169: Error in base set.
*Scanning a type declaration, found SET OF INTEGER, integer sets not allowed - use SET OF CHAR or SET OF subrange-type.

170: Must be a string variable name.
*While scanning the standard procedure INSERT, DELETE, or STR a string variable was expected but not found.

172: Unknow or recursive type reference.
*An unknown type has been referenced within a type or a type has used itself within its own type definition.

174: For control variable cannot be assigned a value.
*The control variable for a FOR loop can not be the target of an assignment statement within the body of the for loop.

177: Assignment to function identifier is not allowed here.
*Scanning an assignment statement and object of assignment is the name of a function but the function declaration is not this block. Assignments to function names can only be made inside the block which declares the function.

178: Record variant is already defined.
*Scanning a record declaration and within a CASE a selection has been scanned which has already been selected.

180: A null string is invalid.
*A string constant of zero characters is not allowed in ISO compilation mode.

189: Pointer variable is expected.
*Scanning a call to standard procedure NEW or DISPOSE and first parameter is not a pointer-type.
*Scanning a call to standard procedure MARK or RELEASE and first parameter is not pointer-type.

201: Length of object is larger than the destination string.
*An assignment to a string variable is made but the size of the source argument is larger than the string variable size.

210: Expected a boolean expression.
*Scanning a call to the standard procedure IOABORT which requires a boolean expression for this parameter.
*Scanning an IF statement and did not find a boolean expression following IF.
*Scanning a REPEAT statement and did not find a boolean expression following UNTIL.
*Scanning a WHILE statement and did not find a boolean expression following WHILE.

212: A text file is not allowed here.
*While scanning the standard procedure SHORTIO a file of text type was found and a text type file is not allowed here.

213: A 128 byte structure is expected.
*While scanning the standard procedure GETINFO or PUTINFO a variable with a size of 128 was expected but not found.

216: Must be type real, integer, or linteger.
*Scanning a call to the standard procedure ABS or SQR, and the argument is not integer-type or real-type.

217: Must be type real.
*Scanning a call to the standard procedure TRUNC or ROUND, and the argument is not real-type.

218: Scalar expected.
*Scanning a call to the standard procedure PRED or SUCC, and the argument is not scalar-type.
*Scanning a FOR statement and found FOR <ident>:=<expr> and possible TO/DOWNTO <expr> but <expr> is not scalar-type.

230: Must be a text file type.
*The following standard procedures can only operate on text files: READLN, WRITELN, INTERACTIVE, PAGE, OVERPRINT, and SYSREPORT.

231: User scalar types not implemented for reading or writing.
*Attempt to read or write user scalar types using one of the following standard procedures: READ, READLN, WRITE, or WRITELN.

232: Set has too many elements.
*Scanning a set declaration, found a definition which requires a lower ordinal limit of less than zero or a higher ordinal limit of greater than 255. This version of the compiler allows sets of up to 256 elements.

233: Too many procedures have been declared.
*An attempt is being made to define more than 4095 procedures within a program block.

234: Integer value expected.
*An array reference using the <expression> FOR <count> language extensions is scanned but <expression> is not an integer-type.

235: Integer or linteger value expected.
*An expression which has a result type of integer or linteger was expected but not found.

236: Type indicator is not defined for this object.
*Trying to determine the object type for a load or store icode instruction, and a file-type, tagfield-type, or variant-type was found which cannot be handled.

237: Variant or tagfield is not allowed here.
*The identifier scanned is either a variant-type or tagfield-type, neither of which is allowed here.

238: Period, BEGIN, PROCEDURE, or FUNCTION expected.
*After having scanned a function or procedure body a period (end of source program), PROCEDURE or FUNCTION (define another routine), or BEGIN (begin an outer block) is expected.

APPENDIX B
RUN-TIME ERROR MESSAGES

The following is the text of OS-9/68000 Pascal run-time error messages. Pascal run-time error messages consist of the number 64, followed by a colon, followed by the error number, followed by a description. Following the text of the error messages is a brief description of the error, this will in many cases aid the user in correcting his program.

064:001 - string array index range error.
*When indexing a string data type the index value was less than one or the index value was greater than the current length of the string.

064:002 - set range error.
*A set expression has a member which is not allowed as a member of the set variable being assigned, or the set parameter being passed.

064:003 - byte subrange range error.
*An expression is out-of-range of the declared subrange.

064:004 - boolean range error.
*An boolean expression was found to be neither TRUE or FALSE.

064:005 - pointer is NIL.
*When dereferencing a pointer the pointer had a value of NIL.

064:006 - pointer range error
*When dereferencing a pointer the pointer had a value outside the current limits of the heap areas.

064:007 - integer subrange range error.
*An expression is out-of-range of the declared subrange.

064:008 - case selector range error.
*The selecting expression has a value that is not in the case selection list.

064:009 - math overflow/underflow error.
*A mathematical operation on a data type produced a result that could not be represented in this data type.

064:010 - integer to character conversion overflow.
*The CHR standard function was passed a value outside the range of 0..255.

064:011 - linteger to integer conversion overflow.
*During a conversion from a linteger to an integer value the linteger value wasn't in the range of -32768..32767.

064:012 - real to linteger conversion overflow.
*The standard function TRUNC, LTRUNC, ROUND, or LROUND was passed a real value that could not be represented in a linteger data type.

064:013 - divide by zero.
*An attempt was made to divide with a divisor of zero.

064:014 - string to integer no digits found.
*While reading or converting a character sequence into an integer or linteger variable a digit character was expected but not found.

064:015 - os9 error on file operation.
*An OS-9 error occured during a file operation.

064:016 - file not open.
*A file operation was attempted on a file that wasn't open.

064:017 - file not in generate mode.
*An output type operation (i.e WRITE, PUT, WRITEEOF etc) was attempted on a file but the file wasn't opened in the generate mode (via REWRITE, UPDATE, or APPEND).

064:018 - eof is false.
*An output type operation (i.e WRITE, PUT, WRITEEOF etc) was attempted on a file but the file had it's EOF status false.

064:019 - eof is true.
*An input type operation (i.e READ, GET, etc) was attempted on a file but the file had it's EOF status true.

064:020 - file not in inspection mode.
*An input type operation (i.e READ, GET, etc) was attempted on a file but the file wasn't opened in the inspection mode (via RESET or UPDATE).

064:021 - file width is negative.
*The file print width parameter is negative.

064:022 - get of short record.
*A file operation was performed which loaded the file window variable of a structured file but the record loaded was of a smaller size than that required by the window variable.

064:023 - unprocessed file error.
*When enabling the abort flag for this file a previous unprocessed file error was detected.

064:024 - file record number out of range.
*The record number specified for this file is too large or negative.

064:025 - not enough memory to start program.
*The stack allocation for the program was too small to allow the program to start. To increase the amount of stack allocation for the program use the "#<number>" shell operator.

064:026 - can't find active procedure.
*A GOTO out-of-block or the EXIT standard procedure can't find the required procedure on the current activation linked list.

064:027 - stack overflow.
*A procedure was called but it requires more stack area than that currently available. To increase the amount of stack allocation for the program use the "#<number>" shell operator.

064:028 - programmed halt.
*The standard procedure HALT was executed.

064:029 - string overflow.
*An assignment of a string expression resulted in a string too large to be stored in the result string variable.
*The standard string functions CONCAT or COPY resulted in a string result with a length greater than 255.

064:030 - os9 error on system call.
*An OS-9 error occured during a system call.

064:031 - heap overflow.
*During a call to the standard procedures NEW or VARNEW the heap area needed to be expanded but all 28 heap blocks were currently used or no system memory was available at this time for the expansion.

064:032 - new or dispose parameter size error.
*A call was made to the standard procedures VARNEW or VARDISPOSE but the size expression had a negative or incorrect value.

064:033 - heap structure is destroyed.
*During the execution of a heap access routine it was determined that the heap structure was destroyed. This can be caused by disposing of a pointer and using a previous copy of that pointer, or by creating a pointer using the variant version of NEW and then accessing the pointer using the wrong tag thus destroying the heap linked list structure.

064:034 - pointer has an odd address.
*During a pointer dereference the pointer was found to have an odd address.

064:035 - copy parameter range error.
*The parameter(s) passed to the standard function COPY are in error.

064:036 - delete parameter range error.
*The parameter(s) passed to the standard procedure DELETE are in error.

064:037 - insert parameter range error.
*The parameter(s) passed to the standard procedure INSERT are in error.

064:038 - fieldget parameter range error.
*The parameter(s) passed to the standard function FIELDGET are in error.

064:039 - fieldput parameter range error.
*The parameter(s) passed to the standard procedure FIELDPUT are in error.

064:040 - shortio parameter range error.
*The buffer length parameter passed to the standard procedure SHORTIO was less than zero or greater than the window variable's size.

064:041 - unimplemented trap call.
*A trap call was made to the OS-9 operating system but it was not previously been initialized.

064:042 - os9 error on math routine load.
*During the linking of the MATH1 or MATH2 trap handlers a OS-9 error occured.

064:043 - real function parameter error.
*A parameter passed to a real transcendatal routine is in error.

064:044 - real conversion format error.
*During the reading or converting of a character sequence into a real value an invalid real format error was detected.

APPENDIX C
OS-9/68000 PASCAL LANGUAGE SYNTAX

The following information describes the syntax of the OS-9/68000 Pascal language. The "Backus-Naur Formalism" will be used to describe the correct structure of Pascal statements. The following symbols are meta-symbols, symbols that describe other symbols, belonging to BNF, but they are not part of the Pascal language.

Meta-symbol	Meaning
::=	- Shall be defined to be.
{ }	- Curly brackets indicate possible repetition of the enclosed symbols.
[ ]	- Brackets indicate that the enclosed symbols are optional.
|	- Alternately.
' ' or " "	- Characters enclosed within quotes are used literally.
( )	- Grouping: use of any one of the constructs.
.	- End of definition.

APPENDIX C
OS-9/68000 PASCAL LANGUAGE SYNTAX

```
   Meta-symbol     Meaning
--------------     ------------------------------------------------

<program heading> ::= PROGRAM <identifier> [ '(' <file identifier>
                                { ',' <file identifier> } ')' ] ';' .

<file identifier> ::= <identifier> .

<identifier> ::= <letter> { (letter | digit | '_' | '$') } .

<letter> ::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' |
             'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' |
             'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | 'a' |
             'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' |
             'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' |
             't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' .

<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' |
            '9' .

<block> ::= <label declaration part>
            <constant definition part>
            <type definition part>
            <variable declaration part>
            <procedure and function declaration part>
            <statement part> .

<label declaration part> ::= <empty> |
                             LABEL <label> { ',' <label> } ';' .

<label> ::= <unsigned integer> .

<constant definition part> ::= <empty> |
                               CONST <constant definition>
                               { ';' <constant definition> } ';' .

<constant definition> ::= <identifier> '=' <constant> .

<constant> ::= <unsigned number> | <sign> <unsigned number> |
               <constant identifier> |
               <sign> <constant identifier> | <string-const> .

<unsigned number> ::= <unsigned integer> | <unsigned real> |
                      <unsigned hexinteger> .

<unsigned integer> ::= <digit> { ('_' | <digit>) } .

<unsigned hexinteger> ::= '$' <hexdigit> { ('_' | <hexdigit>) } .

<hexdigit> ::= <digit> | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'a' |
               'b' | 'c' | 'd' | 'e' | 'f' .
```

Meta-symbol	Meaning

```
<unsigned real> ::= <unsigned integer> '.' <unsigned integer> |
                    <unsigned integer> '.' <unsigned integer>
                    ( 'E' | 'e' ) <scale factor> |
                    <unsigned integer> ( 'E', 'e' ) <scale factor> .

<scale factor> ::= [<sign>] <unsigned integer> .

<sign> ::= '+' | '-' .

<constant identifier> ::= <identifier>

<string-const> ::= "'" <character> { <character> } "'" .

<type definition part> ::= <empty> |
                           TYPE <type definition>
                           { ';' <type definition> } ';' .

<type definition> ::= <identifier> '=' <type> .

<type> ::= <simple type> | <structured type> | <pointer type> .

<simple type> ::= <scalar type> | <subrange type> |
                  <type identifier> .

<scalar type> ::= '(' <identifier> { ',' <identifier>} ')' .

<subrange type> ::= <constant> '..' <constant> .

<type identifier> ::= <identifier> .

<structured type> ::= <unpacked structured type> |
                      PACKED <unpacked structured type> .

<unpacked structured type> ::= <array type> |
                               <record type> |
                               <set type> |
                               <file type> .

<array type> ::= ARRAY '[' <index type> { ',' <index type>} ] OF
                 <component type> .

<index type> ::= <simple type> .

<component type> ::= <type> .

<record type> ::= RECORD <field list> END .

<field list> ::= <fixed part> | <fixed part> ';' <variant part> |
                 <variant part> .
```

Meta-symbol	Meaning

```
<fixed part> ::= <record section> { ';' <record section> } .

<record section> ::= <field identifier> { ',' <field identifier> }
                     ':' <type> | <empty> .

<variant part> ::= CASE <tag field> <type identifier> OF <variant>
                   { ';' <variant> } .

<tag field> ::= <field identifier> ':' | <empty> .
<variant> ::= <case label list> ':' '(' [ <field list> [';'] ] ')' .

<case label list> ::= <case label> { ',' <case label> } .

<case label> ::= <constant> .

<set type> ::= SET OF <base type> .

<base type> ::= <simple type> .

<file type> ::= FILE OF <type> .

<pointer type> ::= '^' <type identifier> .

<variable declaration> ::= <empty> |
                           VAR <variable definition>
                           { ';' <variable definition> } ';' .

<variable definition> ::= <identifier> {',' <identifier>} ':'
                          <type> .

<procedure and function declaration part> ::= <empty> |
          { <procedure or function declaration> ';' } .

<procedure or function declaration> ::= <procedure declaration> |
                                        <function declaration> .

<procedure declaration> ::= <procedure heading> <block> .

<procedure heading> ::= PROCEDURE <identifier> [ '('
                        <formal parameter section>
                        {';' <formal parameter section>} ')' ] ';'

<formal parameter section> ::= <parameter group> |
                               VAR <parameter group> .


<parameter group> ::= <identifier> { ',' <identifier> } ':'
                      <type identifier> .
```