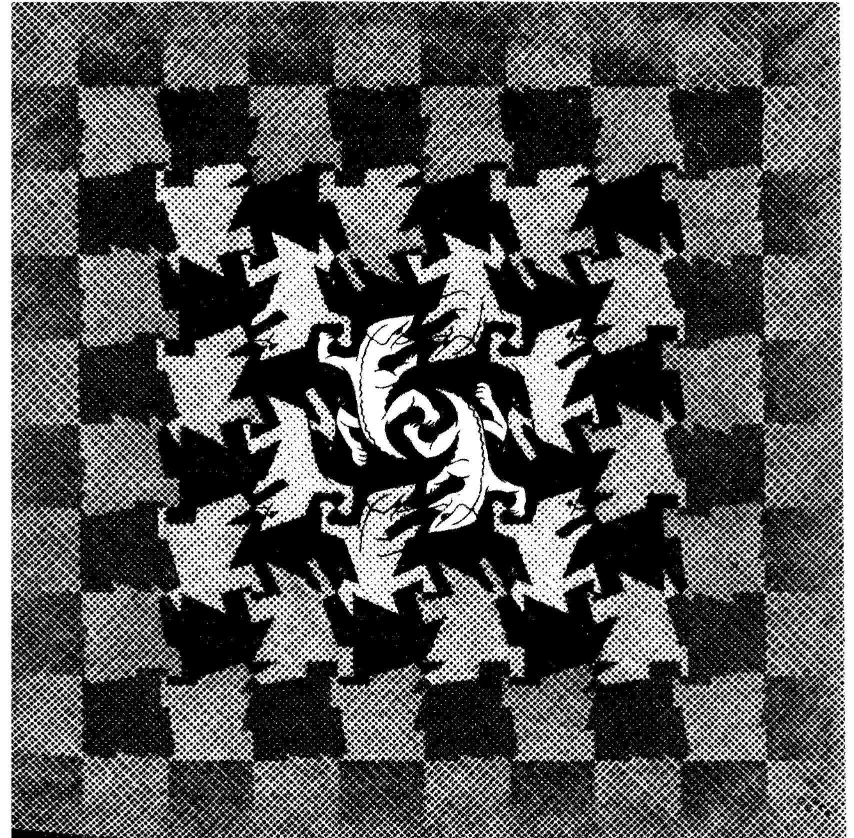
Pascal

Programmer's Manual   Volume 2



Notice

Apple Computer reserves the right to make improvements in the product described in this manual at any time and without notice.

Disclaimer of All Warranties And Liabilities

Apple Computer makes no warranties, either express or implied, with respect to this manual or with respect to the software described in this manual, its quality, performance, merchantability, or fitness for any particular purpose. Apple Computer software is sold or licensed "as is." The entire risk as to its quality and performance is with the buyer. Should the programs prove defective following their purchase, the buyer (and not Apple Computer, its distributor, or its retailer) assumes the entire cost of all necessary servicing, repair, or correction and any incidental or consequential damages. In no event will Apple Computer be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the software, even if Apple Computer has been advised of the possibility of such damages. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

This manual is copyrighted. All rights are reserved. This document may not, in whole or part, be copied, photocopied, reproduced, translated or reduced to any electronic medium or machine readable form without prior consent, in writing, from Apple Computer.

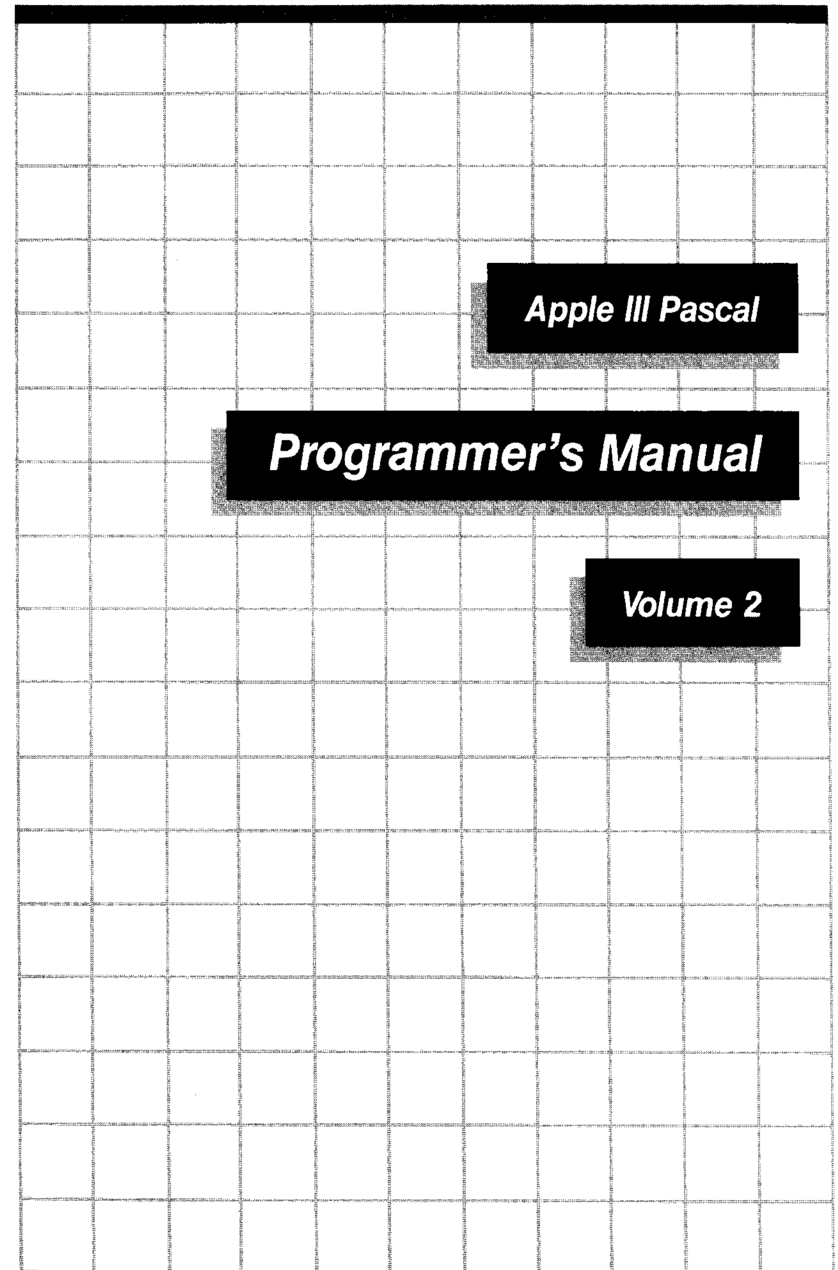
© 1981 by Apple Computer
10260 Bandley Drive
Cupertino, California 95014
(408) 996-1010

© BEELDRECHT, Amsterdam/VEGA, NY
Collection Haags Gemeentemuseum

Written by David Cásseres

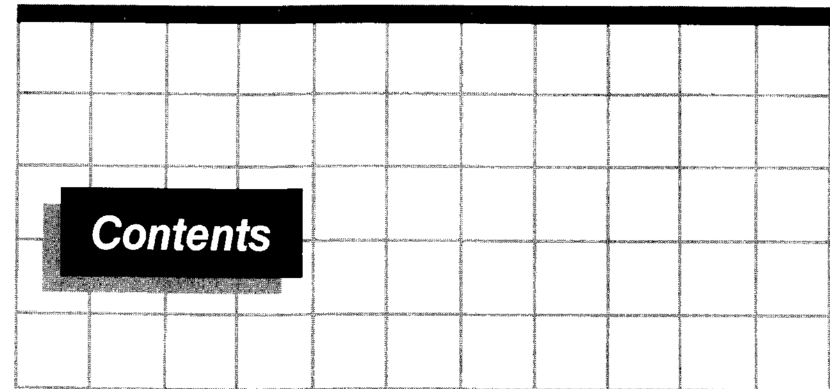
The word Apple and the Apple logo are registered trademarks of Apple Computer.

Reorder Apple Product #A3L0003



Acknowledgements

The Apple III Pascal system is based on UCSD Pascal.
"UCSD PASCAL" is a trademark of the Regents of the University of California. Use thereof in conjunction with any goods or services is authorized by specific license only and is an indication that the associated product or service has met quality assurance standards prescribed by the University. Any unauthorized use thereof is contrary to the laws of the State of California.



Volume II—Appendices

Preface ix

A The TRANSCEND and REALMODES Units 1

2 Introduction
2 The Units
4 The Functions
5 The Remainder (REM) Function

B The PGRAF Unit 11

13 Overview
18 Memory Usage
19 Saving and Loading Display Buffers
19 Summary of PGRAF Routines
2Ø Initial Conditions
2Ø GRAPHIXMODE
21 GRAPHIXON and TEXTON
21 PENCOLOR and FILLCOLOR
22 VIEWPORT
23 INITGRAFIX
23 MOVETO, LINETO, and DOTAT
24 MOVEREL, LINEREL, and DOTREL
24 FILLPORT

25 XYCOLOR, XLOC, and YLOC
25 Text in Graphics
26 DRAWIMAGE
29 The Color Table
31 The Transfer Option
33 NEWFONT and SYSFONT
34 GSAVE and GLOAD
35 The CP28Ø Mode
36 Reading From the Graphics Driver
37 The PGRAF Interface

C The CHAINSTUFF Unit 39

4Ø The SETCHAIN Procedure
41 The SETCVAL Procedure
41 The GETCVAL Procedure
42 An Example of Chaining

D The APPLESTUFF Unit 45

46 The RANDOM Function
48 The RANDOMIZE Procedure
48 The KEYPRESS Function
49 The JOYSTICK Procedure
49 The SOUND Procedure
5Ø The Internal Date and Time
52 PADDLE, BUTTON, and NOTE

E Floating-Point Arithmetic 55

56 Introduction
59 Exceptions
62 Floating-Point Format
64 Arithmetic with Denormalized Numbers
65 Infinity Arithmetic and Comparisons
69 NaNs
71 Accuracy
76 Real Arithmetic Environments
77 Exception Handling
78 Arithmetic Modes
83 Summary of the Floating-Point System
85 Bibliography

F The Apple III Pascal Compiler 87

88 Introduction
88 Diskette Files Needed
89 Using the Compiler
93 Compiler Option Syntax
94 Options that Do Not Affect Program Code
98 Error Checking Options
1ØØ Control of Segments and Libraries
1Ø2 The USING Option
1Ø2 The INCLUDE Option
1Ø3 Special Compilation Mode
1Ø4 Conditional Compilation
1Ø9 Compiling Apple II Code
11Ø Compiler Option Summary

G Special Techniques 113

114 Introduction
114 Representation of Scalar Values
116 Implications
119 Representation of Arrays
12Ø Representation of Real Values
12Ø Free Union Variants
124 Byte-Oriented Built-Ins Revisited
125 Special Uses of UNITSTATUS

H Comparison To Apple II Pascal 127

128 OTHERWISE Clause in CASE Statement
128 SOS Pathnames
128 SOS Device Driver Support
128 Graphics
129 New Procedures
129 New Data Types
129 Real Arithmetic
129 Library Files and Units
13Ø Memory Organization
13Ø The UNITSTATUS Procedure
13Ø Runtime Segment Table
13Ø Conditional Compilation
131 The CHAINSTUFF Unit
131 Compiling Apple II Code
131 File Variable Size

131 Compiler Options
131 Procedure Complexity
131 System Globals

I Syntax Diagrams 133

134 Compilation
134 Program
135 Unit
135 Intrinsic Unit heading
135 Regular Unit heading
136 Interface
136 Implementation
137 Block
137 Uses Declarations
138 Label Declarations
138 Constant Declarations
138 Constant
138 Type Declarations
139 Type
139 Simple Type
139 User-defined Scalar Type
140 Subrange Type
140 Pointer Type
140 Set Type
140 String Type
140 Array Type
141 Record Type
141 Field List
141 Variant Part
142 File Type
142 Variable Declarations
142 Procedure Definition
143 Function Definition
143 Parameter List
143 Parameter Declaration
143 Compound Statement
144 Statement
144 Assignment Statement
144 Procedure Call
145 With Statement
145 Goto Statement
145 For Statement
145 Repeat Statement
146 While Statement
146 If Statement
146 Case Statement

146 Case Clause
147 Otherwise Clause
147 Expression
147 Simple Expression
148 Term
148 Factor
149 Variable reference
149 Function Call
149 Set Constructor
150 Unsigned Constant
150 Unsigned Number
150 Unsigned Integer
151 Identifier

J Tables 153

154 Table 1: Execution Errors
155 Table 2: I/O Errors
157 Table 3: Reserved Words
158 Table 4: Predefined Identifiers
159 Table 5: Compiler Error Messages
164 Table 6: ASCII Character Codes
165 Table 7: Standard I/O Devices
166 Table 8: Size Limitations

K The TURTLEGRAPHICS Unit 167

168 Using Apple II TURTLEGRAPHICS with the Apple III

Figures and Tables 169

Index 177

Preface

The Apple III Pascal system is described in three manuals:

Apple III Pascal: Intoduction, Filer, and Editor
Apple III Pascal Program Preparation Tools
Apple III Pascal Programmer's Manual (Volumes 1 and 2)

Before using the Apple III Pascal system or reading its manuals, you should be familiar with starting up the Apple III as described in the Apple III Owner's Guide.

When you are familiar with the contents of that manual, begin reading the Apple III Pascal: Introduction, Filer, and Editor manual. The Filer and the Editor described in this manual are needed by everyone who uses the Pascal system. If you are familiar with the Apple II Pascal system, this manual will show you the differences in operation between the two systems.

Apple III Pascal Program Preparation Tools is the next manual that you should read before you start to develop Pascal and assembly-language programs to run on the Apple III. The components of the Apple III Pascal system covered in this manual include

- The Linker, used to combine separately developed program segments stored in libraries with your application program.

- The Apple III Pascal 65Ø2 Assembler, used to translate assembly-language source files produced by the Pascal Editor into machine-language code files.

- The Librarian, used to put commonly used routines into libraries for use with application programs.

Your main source of information while developing Pascal programs will be the two volumes of the Apple III Pascal Programmer's Manual, which contain a complete description of the Pascal language on the Apple III and the use of the Apple III Pascal Compiler.

The Contents of This Manual

This manual describes the complete Apple III Pascal language. Except for the introductory material in Chapters 1 and 2, this is an explanatory reference manual rather than a textbook; it does not assume that you know anything about Pascal, but it does assume that you are familiar with computer programming in some language.

Please note that a large and detailed index is provided at the end of this manual; you will probably need it when you are using the manual for reference purposes. The index does not point to every occurrence of a word or phrase in the manual; instead it points to the pages that have significant information about the topic associated with the word or phrase.

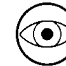
Volume 1 of this manual contains the chapters; Volume 2 contains the appendices and the index. Here is a brief description of the contents:

- Chapter 1 is an introduction to the Pascal language, comparing it with other well-known languages and giving a very simple program as an example.
- Chapter 2 is an extensive overview of Pascal. Every major concept and construction in the language is introduced here at an intuitive level.
- Chapters 3 through 11 provide complete, detailed information about every major feature of the language.
- Chapters 12 through 15 provide complete, detailed information about the more specialized features of the language. These features are needed for certain large or specialized programs.
- Appendices A through E describe the standard library facilities of Apple III Pascal. These are sets of procedures and functions for special purposes such as graphics, audio, joystick inputs, and special arithmetic features.

- Appendix F is a complete reference manual for the Apple III Pascal Compiler, including details of operation and all of the Compiler options.
- Appendices G through J are supplementary information on various topics. In particular, Appendix J is a collection of useful tables.
- Appendix K provides information on the use of Apple II TURTLEGRAPHICS on the Apple III.

Two special symbols are used throughout this manual to draw your attention to particular items of information.

The pointing hand indicates something particularly interesting or useful.



The eye is used for points you need to be cautious about.

Syntax Diagrams

Throughout this manual, the syntax of the Pascal language is indicated by means of syntax diagrams, also known as "railroad tracks." These diagrams are easy to follow once you are used to them: begin at the upper left and follow the arrows. Every possible path through the diagram represents a valid construction in Pascal. For example:

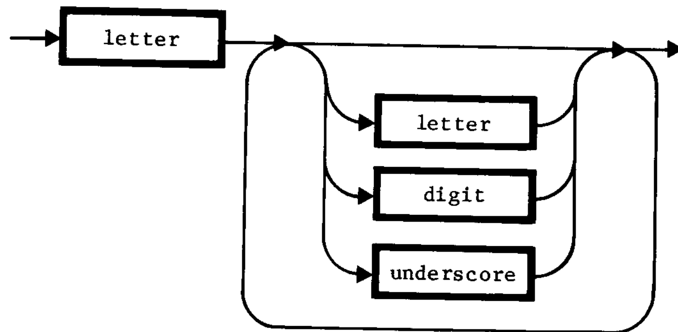
while statement

This diagram tells us that a "while statement" consists of the word WHILE, followed by an expression, followed by the word DO, followed by a statement.

The words WHILE and DO are enclosed in rounded "bubbles;" this means that they are reserved words or symbols of the language, to be typed as shown. The words expression and statement are in boxes with square corners; this means that they are higher-level constructions, which have their own syntax diagrams.

Here is an example where there is more than one path through the diagram:

<u>identifier</u>



This tells us that an identifier begins with a letter, and this letter may be followed by a letter, a digit, an underscore, or nothing. From here, there is the possibility of looping back to add another letter, digit, underscore, or nothing. This can be repeated indefinitely (in principle), so the syntax says that an identifier can be of any length. In practice, of course, there is a limit which the syntax does not show.

Note that Appendix I contains a full set of syntax diagrams.

Syntax of Procedure and Function Calls

Pascal provides a number of built-in procedures and functions which are activated by means of "calls." Most of these use a simple kind of syntax in which there is only one path through the diagram, and in these cases a diagram is not shown. Instead, a "form" is given; for example, the form of the REWRITE procedure is

```
REWRITE ( FILEID, PATHNAME )
```


The word REWRITE is the name of the procedure, and is to be typed as shown; all words in parentheses are names for "parameters," to be replaced with actual expressions or variable identifiers as explained in the text. In this example, FILEID is to be replaced by the identifier of a "file variable" and PATHNAME is to be replaced by a string of characters that is the pathname of a file.

A few procedures have a more complex form of syntax, and syntax diagrams are used for these.

A

The TRANSCEND and REALMODES Units

Introduction

The transcendental, square root, and remainder functions are not built into Apple III Pascal. Instead, they are provided in two units, TRANSCEND and REALMODES, which are intrinsic units in the SYSTEM.LIBRARY file. This appendix describes TRANSCEND and REALMODES and the precision of their functions.

The first part of the appendix is an overview of the units and how to use them. The second part is a description of the functions for users concerned with the mathematical precision of the functions. A section on each function presents a precise description of the value the function takes and the value it returns.

The Units

The REALMODES unit contains the remainder and square root functions.

The TRANSCEND unit contains the following transcendental functions:

sine
cosine
arctangent
e to the x
natural logarithm
decimal logarithm

To compile or execute a program that uses TRANSCEND, both TRANSCEND and REALMODES must be either in the SYSTEM.LIBRARY file on the system diskette or in the program library file. See Chapter 14.

Remainder is a new function added to UCSD Pascal in order to conform to the IEEE floating-point standard. Square root and the transcendental functions listed above have been modified to take advantage of the increased capabilities of the IEEE floating-point standard. The improvements include:

- faster square root;
- increased accuracy in natural log, e to the x, square root, and decimal logarithm;
- the ability to produce and compute with
  - infinities,
  - NaNs (Not a Number), and
  - denormalized numbers (those in the range 1.2E-38 to 7E-46) which diminish the effect of underflow to be comparable to that of rounding errors;
- increased domain in sine, cosine, and arc tangent. (Previously ATAN(x) was defined only if -4.6Ø27Ø5E18 < x < 4.6Ø27Ø5E18. Now ATAN(x) is defined for all real values including infinities and NaNs.)

To use the remainder or the square root function, a program must have a USES declaration containing the identifier REALMODES immediately after the program heading. For example, the following USES declaration makes the public functions of the REALMODES unit available to the program:

```
PROGRAM ALGEBRAIC;
USES REALMODES;
...
```

To use the transcendental functions, a program must have a USES declaration containing <u>both</u> the identifiers REALMODES and TRANSCEND immediately after the program heading. Since the TRANSCEND unit uses the REALMODES unit, REALMODES must appear in the USES declaration before TRANSCEND. For example, the following USES declaration makes the public functions of the TRANSCEND unit available to the program:

```
PROGRAM GEOMETRIC;
USES REALMODES,TRANSCEND;
...
```

The Functions

This section of the appendix describes the effect of the functions on the arithmetic modes of the calling program, defines a NaN, and describes each function. At the end of the appendix is a summary of special values and results showing the argument, mode, result, and signal for each function.

Modes

The IEEE floating-point environment supplies the programmer with arithmetic that can be customized for special use. Functions that use these options must restore the environment of the program that called them before they return control to the calling program. The remainder, square root, and transcendental functions save the arithmetic modes of the calling program for later restoration and then set switches for

- round-to-nearest mode, and

- no-halt mode on overflow, underflow, or floating-point-to-integer conversion. If one of these events occurs during the calculation of a transcendental function, the event isn't reported to the calling routine.

The called function restores the floating-point modes of the calling program just before returning control to the calling program. The status of the overflow, underflow, and floating-point conversion signals is restored to the status they held before the call. A function inherits the Warning/Normalizing and Affine/Projective modes from the calling program.

When the square root or remainder function or one of the transcendental functions is called by a program, the following sequence occurs:

1. function call
2. store calling program modes
3. set modes the function needs
4. compute function values
5. restore calling program modes
6. exit to calling program

For further explanation of modes and exceptions, see Appendix E.

NaNs

A NaN (Not a Number) is a diagnostic assigned to a floating-point variable. It is produced as the result of an invalid floating-point operation (such as Ø divided by Ø). If the argument of a transcendental function is a NaN, the result produced is also that NaN. (If the argument is a trapping NaN, and the Halt on Invalid switch is set, the program halts.) For further explanation of NaNs, see Appendix E.

The Square Root(SQRT) Function

The SQRT(x) function takes any non-negative real value, x (including infinity), and returns the square root of x, except when

x is negative;
x is denormalized and Warning mode is set; or
x is infinity in Projective mode.

For any of these exceptions, the Invalid Operation Signal will be set. If the Halt on Invalid switch is not set, a diagnostic NaN will be returned.

The Remainder(REM) Function

The REM(x,y) function is defined by the following relation when y is not zero:

REM(x,y) = x - y * n

where n is the integer nearest x/y. When the fractional part of

x/y is exactly 1/2, then n is the even integer nearest x/y. (For example, if x/y = 3.5, then n = 4; if x/y = 6.5, then n = 6.) The remainder function is exact, except when

y is zero;
x is infinite; or
y is denormalized and Warning mode is set.

For any of these exceptions, the Invalid Operation Signal will be set. If the Halt on Invalid switch is not set, a diagnostic (NaN) will be returned.

The Sine(SIN) Function

The SIN(x) function takes an angle, x, in radians, and returns the sine of that angle when

-1Ø2942.13 < x < 1Ø2942.13

For arguments outside this interval, a diagnostic NaN is produced. The accuracy of the SIN function cannot be guaranteed outside this interval because of argument reduction errors.

The Cosine(COS) Function

The COS(x) function takes an angle, x, in radians, and returns the sine of that angle when

-1Ø2942.13 < x < 1Ø2942.13

For arguments outside this interval, a diagnostic NaN is produced. The accuracy of the COS function cannot be guaranteed outside this interval because of argument reduction errors.

The Arctangent(ATAN) Function

The ATAN(x) (or inverse tangent) function takes any real number, x (including + or - infinity), and returns the angle that is the arctangent.

R = ATAN(x)   -pi/2 <= R <= pi/2

The Natural Logarithm(LN) Function

The LN(x) function takes any non-negative positive real value x, and returns its natural logarithm except when

x is denormalized (1.175494E-38 > x > 7.ØØ6492E-46) in Warning mode;
x is negative; or
x is Ø in Projective mode.

For any of these exceptions the Invalid Operation Signal is set. If the HALT on Invalid switch is not set, a diagnostic NaN is returned.

The Logarithm(LOG) Function

The LOG(x) function takes any non-negative positive real value, x, and returns its base 1Ø logarithm except when

x is denormalized (1.175494E-38 > x > 7.ØØ6492E-46) in Warning mode;
x is negative; or
x is Ø in Projective mode.

For any of these exceptions the Invalid Operation Signal is set. If the Halt on Invalid switch is not set, a diagnostic NaN is returned.

The Exponential(EXP) Function

The EXP(x) function takes a real value, x, and computes e raised to the x power except when

x is + or - infinity in Projective mode.

This causes the Invalid Operation Signal to be set. If the Halt on Invalid switch is not set, a diagnostic NaN is returned.

Summary of Special Values and Results

The figure below summarizes special values and results for each function.

The square root and transcendental functions almost always set the Inexact Signal (except for special cases, such as SIN(Ø) = Ø).

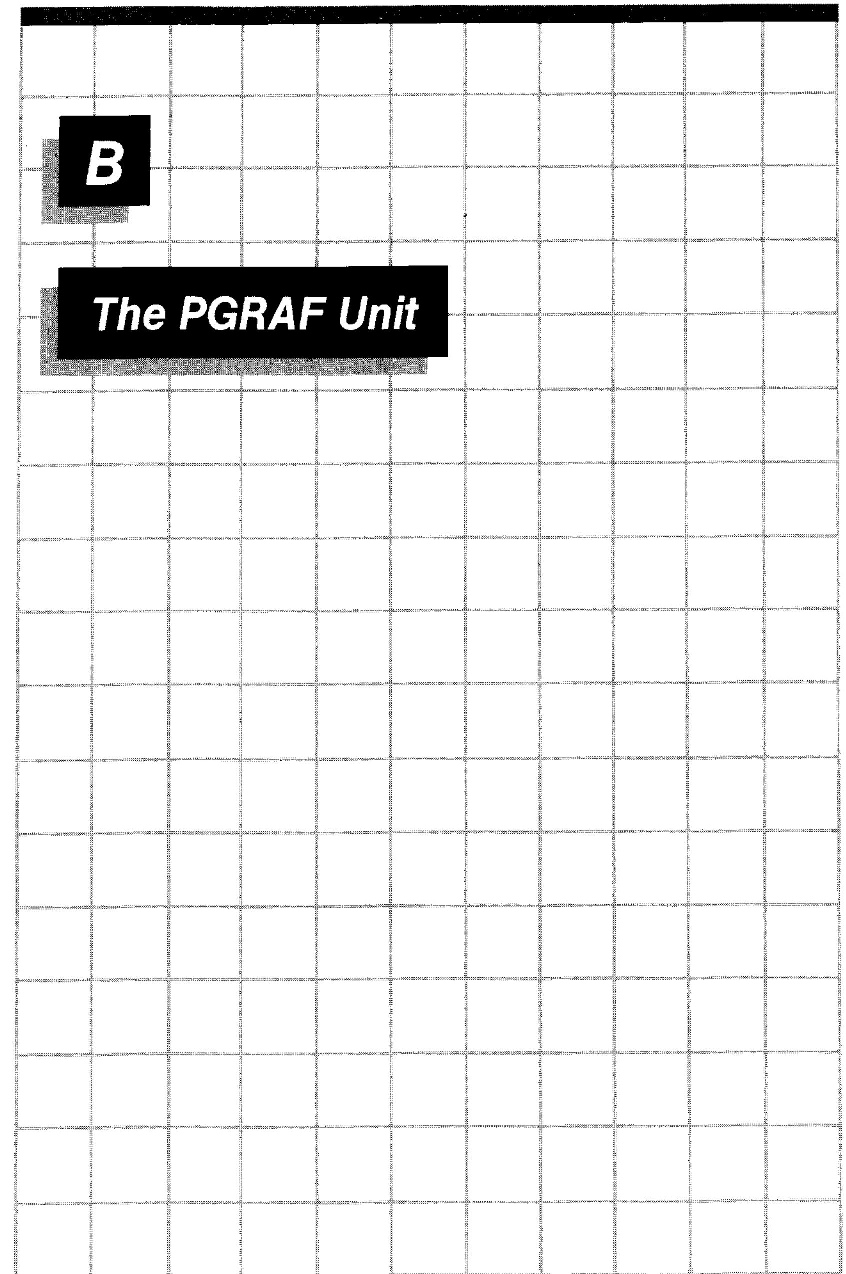
SQUARE ROOT, REMAINDER, AND TRANSCENDENTAL FUNCTIONS:
Summary of Special Values and Results

Function	Argument	Mode	Result	Signal
SQRT	-0	any	-0	none
	+infinity	Affine	+infinity	none
	+infinity	Proj	NaN	Invalid Operation
	negative	any	NaN	Invalid Operation
	Denorm	Warning	NaN	Invalid Operand
	NaN	any	argument	none
REM (x,y)	y = 0	any	NaN	Invalid Operation
	x is infinite	any	NaN	Invalid Operation
	y is a Denorm	Warning	NaN	Invalid Operand
	NaN	any	argument	none
SIN	out of range*	any	NaN	Argument Reduction Error
	NaN	any	argument	none
COS	out of range*	any	NaN	Argument Reduction Error
	NaN	any	argument	none
ATAN	+infinity	any	pi/2	Inexact
	-infinity	any	-pi/2	Inexact
	NaN	any	argument	none

*The range for sine and cosine is $-102942.136 < x < 102942.13$.

SQUARE ROOT, REMAINDER, AND TRANSCENDENTAL FUNCTIONS:
Summary of Special Values and Results

Function	Argument	Mode	Result	Signal
LN	+infinity	Affine	+infinity	none
	+infinity	Proj	NaN	Invalid Operation
	negative	any	NaN	Invalid Operation
	Denorm	Warning	NaN	Invalid Operand
	+/-0	any	-infinity	none
	NaN	any	argument	none
LOG	+infinity	Affine	+infinity	none
	+infinity	Proj	NaN	Invalid Operation
	negative	any	NaN	Invalid Operation
	Denorm	Warning	NaN	Invalid Operand
	+/-0	any	-infinity	none
	NaN	any	argument	none
EXP	+infinity	Affine	+infinity	none
	+infinity	Proj	NaN	Invalid Operation
	-infinity	Affine	0	none
	-infinity	Proj	NaN	Invalid Operation
	NaN	any	argument	none



B

The PGRAF Unit

The PGRAF unit provides a convenient Pascal interface to the system's graphics driver, which is known by the SOS device name .GRAFIX or the Pascal device name GRAPHIC: (or unit #3:). (Complete details on the graphics driver are given in the Standard Device Drivers Handbook.)

PGRAF is an intrinsic unit in the SYSTEM.LIBRARY file. To compile or execute a program that uses the PGRAF unit, this unit must be either in the SYSTEM.LIBRARY file on the system diskette, or in the program library (see Chapter 14).

To use the facilities of the PGRAF unit, the program must have a USES declaration containing the identifier PGRAF, immediately after the program heading; for example,

```
PROGRAM PLOTCURVES;
USES PGRAF, REALMODES, TRANSCEND;
...
```

The public procedures, functions, and data types of the PGRAF unit are then available to the program.

Before any program that uses PGRAF can be executed, you must use the system-level Options command to reserve the necessary memory space for graphics display buffers. Details are given below in the section on "Memory Usage."

Throughout this appendix, you will find references to "default" values and options. Many of these defaults are provided by the graphics driver, rather than by PGRAF, and can be changed at the driver level via the System Configuration Program on the UTILITIES diskette; see the Standard Device Drivers Handbook for details.

Note that the PGRAF unit is not the only way to use the graphics driver; you can also use UNITWRITE (see Chapter 12) to send characters directly to the graphics driver, referencing it as unit number 3. Similarly, you can use UNITREAD to input characters from the graphics driver (see last section of this appendix). Use a "mode" value of 12 with UNITREAD and UNITWRITE when communicating with the graphics driver.

Overview

Before describing the actual procedures and functions of the PGRAF unit, we present an overview of the concepts and operations involved.

Graphic Displays

An Apple III graphic display can be thought of as a rectangular array of dots (sometimes called "pixels"). An X,Y coordinate system is superimposed on this dot array; the origin (Ø,Ø) is at the lower left-hand corner of the array, with X increasing to the right and Y increasing toward the top of the display. This is strictly an integer coordinate system. The height of the display is always 192 dots, with Y coordinates in the range Ø..191; the width in dots depends on the selected "graphics mode" as explained below.

Another feature of the display is an invisible cursor which is used as a position reference in certain operations. There are also procedures for moving the cursor without affecting the display.

Graphics Modes

There are four distinct modes for Apple III graphics. Each mode is characterized by the number of dots in each horizontal row on the physical screen and by the colors available:

- BW28Ø: In this mode the only available colors are black and white. The screen is treated as 28Ø dots wide and 192 dots high; that is, X coordinates are in the range Ø..279 and Y coordinates are in the range Ø..191.

- BW56Ø: This is identical to BW28Ø except that the horizontal scale is 56Ø dots instead of 28Ø. X coordinates are in the range Ø..559 and Y coordinates are in the range Ø..191.

- COL14Ø: In this mode, 16 colors are available. The horizontal scale is 14Ø dots; X coordinates are in the range Ø..139 and Y coordinates are in the range Ø..191.

- CP28Ø: In this mode, 16 colors are available but there are special limitations. The horizontal scale is 28Ø dots; X coordinates are in the range Ø..279 and Y coordinates are in the range Ø..191. A full explanation of this mode is left until the end of this appendix.

The digits in the identifier of each mode indicate the horizontal scale.

In each mode, two distinct display buffers are available, as explained below. The size of these buffers depends on the graphics mode selected. Before executing any program that uses PGRAF, you must use the system-level Options command to tell the system how much memory must be reserved for display buffers. Details are given below in the section on "Memory Usage."

Dots and Lines

PGRAF provides a set of procedures for plotting dots and lines. When PGRAF plots a line, it does so by plotting a sequence of dots; thus everything that PGRAF does can be thought of in terms of dots.

A dot is plotted by giving its X,Y coordinates. A line is plotted by giving one pair of X,Y coordinates; the result is a line from the current cursor position to the specified coordinates. Alternatively, you can give X and Y displacements instead of absolute coordinates; the displacements are taken relative to the cursor position.

Colors

There are sixteen colors, with the following identifiers and ordinalities:

Ordinality	Identifier	Ordinality	Identifier
Ø	BLACK	8	BROWN
1	MAGENTA	9	ORANGE
2	DARKBLUE	1Ø	GREY2
3	PURPLE	11	PINK
4	DARKGREEN	12	GREEN
5	GREY1	13	YELLOW
6	MEDBLUE	14	AQUA
7	LIGHTBLUE	15	WHITE

In the black-and-white modes, all colors other than BLACK are converted to WHITE. Also, note that the colors produce a 16-level grey scale on the Apple III's black/white video output.

Control of Color

At all times, there is a currently selected pen color and a currently selected fill color. PGRAF provides procedures for selecting these colors. By default the pen color is WHITE and the fill color is BLACK.

In the simplest way of using graphics, plotting a dot changes its color to the current pen color. A specified area of the display can be "erased" by a filling operation; this changes all the dots in the area to the fill color.

A great deal can be done with just these simple techniques. More powerful techniques make use of two controllable processes that affect every plotting or filling operation:

- A color table is used to modify the color used for plotting or filling. The resulting color at any point depends on the source color (pen color or fill color) and may also depend on the existing color of the dot.

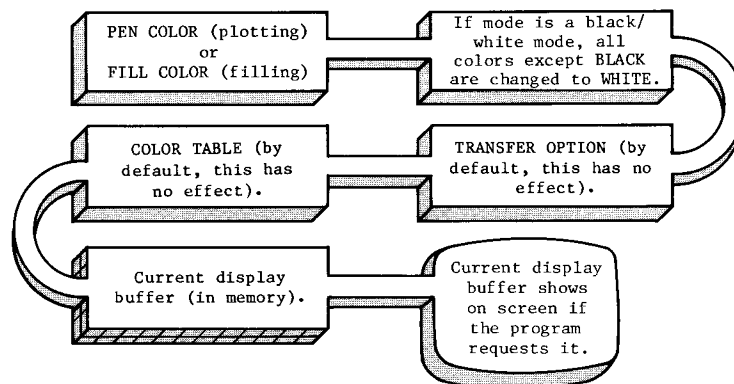
  The color table is always applied to the pen color when a dot is plotted, or the fill color when a dot is filled. By default, the color table specifies that the result is always the same as the source color, but you can change this.

- A transfer option is used to determine how the resulting color from the color table is applied to the actual dot on the display. The effect on the display depends on the color from the color table, and may also depend on the existing color of the dot.

The transfer option is always applied to the color that results from applying the color table. By default, the transfer option specifies that the result from the color table is the new color of the dot, but you can change this.

By changing either the color table or the transfer option, you can cause the colors to be modified before they are actually placed on the display. Usually, only one of these methods is used at a time (usually the color table); however, exotic combinations may prove useful in certain cases.

The use of the color table and the transfer option are explained further on. The following diagram shows the transformations that are always applied to a color before it appears on the display:



The "current display buffer" concept is explained further on.

The Viewport

One of the PGRAF procedures allows you to define the boundaries of the current viewport. This is the area of the display that can be affected by plotting and filling operations; by default, the viewport is the whole display. If the program tries to plot or fill outside the viewport there is no effect. If a line is plotted and any portion of it is outside the viewport, only the part that is in the viewport is actually plotted.

Note, however, that the motion of the invisible cursor is not limited by the viewport boundaries.

The FILLPORT procedure fills the current viewport with the fill color; this is a useful way of clearing the viewport.

Display Buffers

Up to this point we have used the term "display" and avoided the term "screen." The reason is that the output from graphics procedures does not go directly to the screen but to the current display buffer. A display buffer is a memory area containing a coded representation of dot colors on a screen. The graphics output affects the data in the current display buffer, but the current display buffer is not shown on the physical screen until the program specifically requests this.

If enough memory has been reserved (see "Memory Usage" below), two display buffers are available simultaneously for the current graphics mode. This means that a program can set up a display on the screen, change to a different display buffer, and create a different display without disturbing the screen. When the new display is ready, the program can cause the screen to show the new buffer.

Text on a Graphics Display

By using WRITE or UNITWRITE, a program can put characters on a graphics-mode display. Each character is drawn in the current pen color, on a background of the current fill color; these colors may be changed by the color table or transfer option.

By default, the characters are drawn in the same system character font used in text mode. Alternatively, the program can switch to a user-defined font (which can, if desired, have a different character size than the system font).

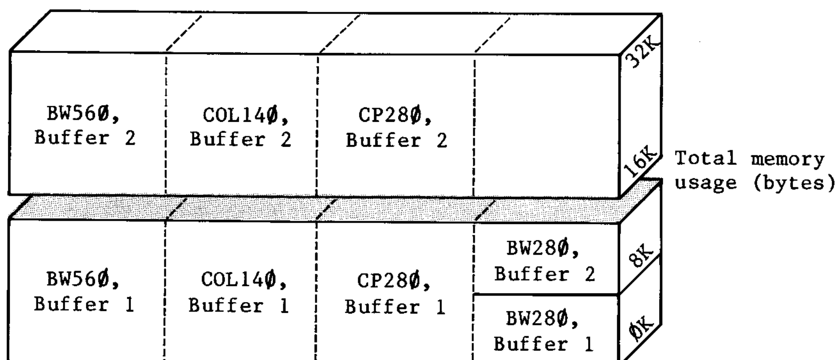
Copying an Image

A program can use internal data (such as a packed array of boolean) to represent dots on the display. A specialized procedure is provided to transfer the pattern of bits in the array to a pattern of dots on the the display, plotting one dot (with the current pen color) for a 1 bit and filling one dot

(with the current fill color) for a Ø bit. The colors may be modified by the color table or transfer option. This is a high-speed procedure and is useful for doing animation.

Memory Usage

For each of the four graphics modes, two display buffers are defined (referenced by the numbers 1 and 2). Before you can execute a program that uses graphics you must first use the system-level Options command to tell the system how much memory is required for graphics buffers. You can determine this from the following diagram.



For any one mode, the two buffers are separate; and the two BW28Ø buffers are separate from buffer 2 of any other mode. Buffers that are separate from each other can be used independently to store different images.

Note that the space required by graphics buffers is subtracted from the memory space available for the program itself.

If your program attempts to use a graphics mode or buffer that has not had the required space allocated via the Options command, PGRAF will halt the program with an error message.

Saving and Loading Display Buffers

Simple methods are provided for saving the current display buffer in a diskette file, and for subsequently retrieving it from the file to the current display buffer.

Summary of PGRAF Routines

The remainder of this appendix is concerned with the actual operation of the procedures and functions of PGRAF. They are:

- GRAFIXMODE to select the graphics mode and the current display buffer; GRAFIXON to show the current buffer on the screen; TEXTON to show the normal text display on the screen.

- PENCOLOR and FILLCOLOR to set colors for plotting and filling. In all operations that use these colors, the colors may be modified by the color table and the transfer option.

- VIEWPORT to set the boundaries of the viewport.

- INITGRAFIX to reinitialize the conditions for graphics operations. The color table and transfer option are set to normal, the viewport to full screen, and the cursor to the lower left corner. Nothing else is changed.

- LINETO, LINEREL, DOTAT, DOTREL, and DRAWIMAGE for plotting; FILLPORT for filling the viewport; MOVETO and MOVEREL for moving the cursor.

- XYCOLOR, XLOC, and YLOC are functions that return information about the current display.

- SETCTAB and XFROPTION to change the color table and transfer option.

- NEWFONT for changing to a user-defined font for text in graphics, and SYSFONT for restoring the normal system font.

- GSAVE for saving the current display buffer in a specified diskette file, and GLOAD for retrieving a saved image into the current display buffer.

Initial Conditions

When you execute a program that contains a USES PGRAF declaration, the PGRAF unit goes through a one-time initialization sequence before any of the main program's statements are executed. This initialization commands the graphics driver to go into its default state; unless the driver's defaults have been changed via the System Configuration Program, the defaults are

Graphics mode:	BW28Ø
Display buffer:	1
Viewport:	Full screen
Cursor position:	Ø,Ø (lower left corner)
Pen color:	White
Fill color:	Black
Transfer option:	Normal (Ø)
Color table:	Normal (see below)
Font for text in graphics:	Current system font.

The "normal" color table and transfer option mean that the pen color and fill color are not altered during the plotting or filling operations.

GRAFIXMODE

The GRAFIXMODE procedure sets the current graphics mode and selects a display buffer. It takes two parameters, which are of types GMODE and GBUF; these types are defined in the PGRAF unit and can be used by any program that uses PGRAF:

```
GMODE = (BW28Ø, CP28Ø, BW56Ø, COL14Ø);
GBUF = 1..2;
```


The form for calling GRAFIXMODE is

```
GRAFIXMODE ( MODE, BUFFER )
```

where MODE is an expression with a result of type GMODE and BUFFER is an expression with a result of type GBUF. For example,

```
GRAFIXMODE(BW28Ø, 2)
```

changes the current graphics mode to BW28Ø and selects buffer 2. This does not affect the screen; it simply causes subsequent graphics operations to affect display buffer 2 of BW28Ø mode. This buffer is not shown on the screen until requested by the GRAFIXON procedure (see below).

GRAFIXON and TEXTON

The GRAFIXON procedure takes no parameters. It causes the current display buffer to appear on the screen. Note that this is the only way to cause a newly selected buffer to appear on the screen. A program that uses PGRAF begins executing with the screen still displaying the normal text display; therefore it must call GRAFIXON at some point in order to put any graphics on the screen.

The TEXTON procedure takes no parameters. It causes the current text-mode display to appear on the screen. Note that while a graphics buffer is being shown on the screen, any operations that would normally put text on the text display still do so; if the program subsequently calls TEXTON, the text display that appears on the screen will reflect these operations.

When a program terminates while showing graphics on the screen, a TEXTON operation is automatically performed to return the screen to normal text mode. This includes both normal termination at the end of a program, and error halts.

PENCOLOR and FILLCOLOR

The PENCOLOR and FILLCOLOR procedures set the colors to be used for plotting and filling operations, repectively. They each take a single parameter, which is of type SCREENCOLOR; this type is

defined in the PGRAF unit and can be used by any program that uses PGRAF:

```
SCREENCOLOR = ( BLACK, MAGENTA, DARKBLUE, PURPLE, DARKGREEN,
                GREY1, MEDBLUE, LIGHTBLUE, BROWN, ORANGE,
                GREY2, PINK, GREEN, YELLOW, AQUA, WHITE );
```

The form for calling PENCOLOR is

```
PENCOLOR ( COLOR )
```

where COLOR is an expression with a result of type SCREENCOLOR. For example,

```
PENCOLOR(LIGHTBLUE)
```

changes the pen color to LIGHTBLUE; subsequent plotting operations will use this color (which may be modified by the color table and transfer option).

The form for calling FILLCOLOR is

```
FILLCOLOR ( COLOR )
```

where COLOR is an expression with a result of type SCREENCOLOR. For example,

```
FILLCOLOR(YELLOW)
```

changes the fill color to YELLOW; subsequent filling operations will use this color (which may be modified by the color table and transfer option).

VIEWPORT

The VIEWPORT procedure sets the boundaries of the viewport. The viewport is simply the area of the display that can be affected by plotting and filling operations. The form for calling VIEWPORT is

```
VIEWPORT ( LEFT, RIGHT, BOTTOM, TOP )
```

where all four parameters are expressions with results of type integer. If any parameter exceeds a boundary of the current

graphics mode, it is replaced by the applicable boundary value.

INITGRAFIX

The INITGRAFIX procedure has no parameters and can be called at any time. It reinitializes four of the operating conditions to their default state:

- The color table is set to its normal state, i.e. no effect on specified colors.
- The transfer option is set to its normal state, i.e. no effect on color results from the color table.
- The viewport is set to full screen.
- The cursor is moved to the (Ø,Ø) point, i.e. the lower left corner of the screen.

MOVETO, LINETO, and DOTAT

All three of these procedures move the cursor to a new position, which is specified by its absolute X and Y coordinates. MOVETO does nothing else except move the cursor; LINETO draws a line from the original cursor position to the new one, using the current pen color; and DOTAT plots a single dot at the new cursor position, using the current pen color. The pen color may be modified by the color table and transfer option. The forms for calling MOVETO, LINETO, or DOTAT are

```
MOVETO ( XCOORD, YCOORD )

LINETO ( XCOORD, YCOORD )

DOTAT ( XCOORD, YCOORD )
```

where XCOORD and YCOORD are expressions with results of type integer and indicate absolute X and Y coordinates on the display.

If any part of a line drawn by LINETO lies outside the current viewport, that part of the line is not plotted. Likewise DOTAT

will not plot a dot outside the current viewport. In all cases, however, the cursor is moved to the new position even if it lies outside the viewport.

MOVEREL, LINEREL, and DOTREL

These procedures are similar to MOVETO, LINETO, and DOTAT, except that the new cursor position is specified by its X and Y displacements relative to the original cursor position. The forms for calling MOVEREL, LINEREL, or DOTREL are

```
MOVEREL ( DX, DY )

LINEREL ( DX, DY )

DOTREL ( DX, DY )
```

where DX and DY are expressions with results of type integer and indicate relative X and Y displacements on the display.

Since these are automatically added to the original cursor coordinates, it is possible for the mathematical result to exceed the limits of Pascal integer values, which are -32768 and 32767. If the new X or Y coordinate of the cursor would exceed one of these limits, then the limit is used as the new coordinate value.

FILLPORT

The FILLPORT procedure takes no parameters. It fills every dot in the current viewport with the current fill color. The color at each point on the display may be modified by the color table or transfer option (see below). If the color table and transfer option are in their default states, the effect of FILLPORT is to erase everything in the current viewport.

XYCOLOR, XLOC, and YLOC

The XYCOLOR, XLOC, and YLOC functions return information about the current display. They take no parameters.

XYCOLOR returns an integer value, which is the ordinality of the color found in the current display buffer at the current cursor position.

XLOC and YLOC return integer values which are the X and Y coordinates of the current cursor position, respectively.

Another way to obtain the color of screen dots is to read from the graphics driver with UNITREAD. See last section of this appendix.

Text in Graphics

To put text onto a graphic display, simply output the text to the graphics driver. This can be done with WRITE if you declare a file variable of type INTERACTIVE and open it with REWRITE and the name 'GRAPHIC:' as in the following example:

```
VAR GSCREEN: INTERACTIVE;
...
REWRITE(GSCREEN, 'GRAPHIC:');
...
WRITELN(GSCREEN, 'Hello!');
```

The string 'Hello!' will be displayed. By default, the system character font is used but you can specify a different font as explained in a later section. The characters are drawn in the current pen color on a background of the current fill color (possibly modified by the color table and transfer option).

The string is positioned with the upper left corner of the first character at the current cursor position, and the cursor is moved into position for the next character to follow, in case there is one.

Another way to write text to the graphics driver is to use UNITWRITE with the value 3 as the "unitnum" parameter. To create the same effect as the previous example, you can declare a string variable and write its value out as follows:

```
VAR MSG: STRING;
...
UNITWRITE(3, MSG[1], LENGTH(MSG), Ø, 12 );
```

Note that the string must be subscripted, since UNITWRITE will treat it as a simple PACKED ARRAY OF CHAR with indices beginning at Ø; the characters of the string start at [1], not at the first byte of the string variable.

DRAWIMAGE

This is a high-speed procedure that draws a rectangular image, using data from a program variable as its input. The variable is typically a two-dimensional packed array of boolean, but can actually be any variable. It is considered as a sequence of bytes of data in memory. The bits within these bytes are mapped into dots on the display.

The sequence of bytes is considered to be broken into groups of a specified length, representing rows; then each of these rows is considered as a row of bits (eight bits for each byte). Thus DRAWIMAGE treats the variable as a two-dimensional array of bits.

Depending on the parameters, DRAWIMAGE can use the entire array of bits or it can select any rectangular subarray. This subarray is mapped to a rectangular area of the display buffer, using each bit to represent one dot. A "1" bit is considered to represent the current pen color, and a "Ø" bit is considered to represent the current fill color; these colors may be modified by the color table or transfer option.

The first row of bits in the array (or subarray) becomes the top row of the image, and the last row of bits becomes the bottom row of the image. Within a row, the first bit is at the left edge of the image and the last bit is at the right edge. The upper left corner of the image (corresponding to the first bit in the first row) is placed at the current cursor position. The cursor is not moved.

The type and size of the variable are not checked in any way. DRAWIMAGE simply starts with the first (least significant) byte of the variable and proceeds to use the bytes that follow it in memory. The number of bytes that DRAWIMAGE uses, and the way they are broken up into rows, are determined entirely by the parameters passed to DRAWIMAGE as explained below.

The form for calling DRAWIMAGE is

```
DRAWIMAGE ( SOURCE, ROWSIZE, XSKIP, YSKIP, WIDTH, HEIGHT )
```

where the parameters are as follows:

SOURCE is a reference to any program variable; its least significant byte is the starting point for DRAWIMAGE.

ROWSIZE is an expression with an integer value and specifies the size, in <u>bytes</u>, of each row in the array of bytes. ROWSIZE should always be an even value when using two-dimensional arrays, since Pascal aligns each row in the array on a word boundary.

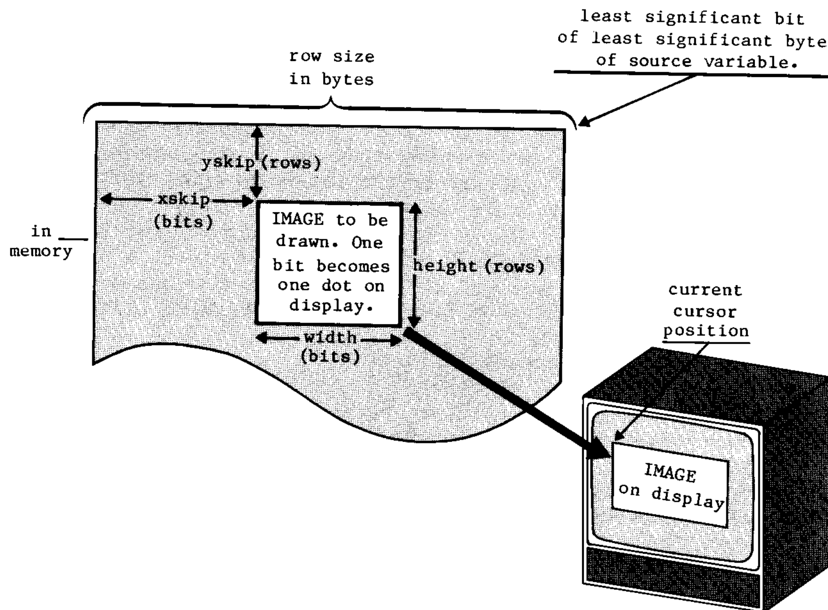
XSKIP is an expression with an integer value and specifies the number of <u>bits</u> to be skipped at the beginning of each row in the array. This parameter and the remaining parameters have the effect of selecting a subarray within the array.

YSKIP is an expression with an integer value and specifies the number of rows to be skipped in the array.

WIDTH is an expression with an integer value and specifies the number of <u>bits</u> to be taken from each row (starting at XSKIP).

HEIGHT is an expression with an integer value and specifies the number of rows to be taken from the array (starting at YSKIP).

The following diagram summarizes the meanings of the parameters:



For example, suppose that we have a 2Øx1Ø packed array of boolean values declared as follows:

```
VAR PIC: PACKED ARRAY[Ø..19, Ø..9] OF BOOLEAN;
```

Since the array is packed, each boolean value will occupy just one bit. Note two important points about the dimensions of this array:

- The first dimension specifies the number of rows in PIC, and the second specifies the number of values (bits) in each row. This means that in a subscripted reference to PIC, the first subscript is a Y coordinate and the second is an X coordinate. This is important to know when you write statements to put some pattern of bits into PIC.

- Although each boolean value is packed down to one bit, each row of 1Ø values is not packed down to 1Ø bits: each row is packed to an integral number of two-byte words. In the present case, each 1Ø-bit row is packed down to one word, or two bytes. In a two-dimensional packed array of boolean, you can calculate the number of bytes occupied by each row from the expression

  ```
  2*( (N+15) DIV 16 )
  ```

  where N is the number of packed boolean values in each row. The result of this expression is the "row size" to be specified to DRAWIMAGE.

Now suppose that a 2Øx1Ø image has been put into PIC, and we want to draw it on the display. The DRAWIMAGE call is

```
DRAWIMAGE(PIC, 2, Ø, Ø, 1Ø, 2Ø)
```

where we specify PIC as the SOURCE variable, 2 as the ROWSIZE, Ø as the XSKIP, Ø as the YSKIP, 1Ø as the WIDTH, and 2Ø as the HEIGHT. Each boolean value in PIC is transformed into a display dot, using the current pen color, color table, and transfer option. The upper left corner is placed at the current cursor position.

The Color Table

The SETCTAB procedure sets the color table. As previously mentioned, the color table is applied to the pen color or fill color whenever a display dot is plotted or filled; by default it has no effect. By changing the color table, you can cause the resulting color to depend on the source color (pen color or fill color) and the existing color of each display dot that is affected.

The form for calling SETCTAB is

```
SETCTAB ( SOURCE, OLD, RESULT )
```

where all three parameters are expressions with results of type SCREENCOLOR. The effect is that whenever a dot is plotted or filled with the specified SOURCE color, and the existing color of that dot is the specified OLD color, then the RESULT color is

used instead of the SOURCE color.

For example, suppose that we want to protect all yellow dots on the display from being drawn over by any other color. We have declared a variable of type SCREENCOLOR:

```
VAR COL: SCREENCOLOR;
```

Now we can protect all yellow dots as follows:

```
FOR COL := BLACK TO WHITE DO SETCTAB(COL, YELLOW, YELLOW)
```

This changes the color table so that for all possible SOURCE colors (from BLACK to WHITE), if the OLD color of a dot is YELLOW then the RESULT color is also YELLOW. Later we can change the color table again so that one color, ORANGE, can draw over a YELLOW dot:

```
SETCTAB(ORANGE, YELLOW, ORANGE)
```

More complicated effects can also be achieved. Suppose that we have an image on the display in GREEN and WHITE on an AQUA background. The color table is in its default state (no effect). Now we want to create a "night" effect by changing the display so that the background becomes BLACK, each GREEN dot becomes DARKGREEN, and each WHITE dot becomes GREY1. We can do this as follows:

```
SETCTAB(BLACK, GREEN, DARKGREEN);
SETCTAB(BLACK, WHITE, GREY1);
FILLCOLOR(BLACK);
FILLPORT
```

When the viewport is filled with BLACK, each AQUA dot is changed to BLACK because we made no changes to the color table for AQUA. Each GREEN dot is changed to DARKGREEN, and each WHITE dot is changed to GREY1.

The color table can always be set back to its default condition (no effect) by the INITGRAFIX procedure.

When the color table is in its default condition, all plotting and filling operations will run faster.

The Transfer Option

The XFROPTION procedure sets the transfer option. As previously mentioned, the transfer option is applied to the result from the color table whenever a display dot is plotted or filled; by default it has no effect.

By changing the transfer option, you can cause the resulting color to depend on the result from the color table and the existing color of each display dot that is plotted or filled. You can do this in any mode, but its main usefulness is in the black-and-white modes.

While the color table operates at the level of specified colors, the transfer option operates at the level of the 4-bit patterns that are used internally to represent colors. Each color is represented internally as a 4-bit pattern whose numeric value is the ordinality of the color. Thus the color BLACK has an ordinality of Ø, and is represented internally as ØØØØ; the color ORANGE has an ordinality of 9 and is represented internally as 1ØØ1, and so forth.

The transfer option specifies a bitwise logical operation upon the source color (pen color or fill color) and the old (existing) color of a display dot. For example, since ORANGE is represented as 1ØØ1 and AQUA is represented as 111Ø, we can say that the meaning of ORANGE AND AQUA is the color represented by 1ØØØ, namely BROWN. In a black-and-white mode, the only color values are ØØØØ (BLACK) and 1111 (WHITE). BLACK AND WHITE means BLACK, while BLACK OR WHITE means WHITE.

By default, the transfer option specifies that the 4-bit result is the same as the 4-bit representation of the source color. The other transfer options specify various 4-bit logical operations involving the source color and the old color; most of them are of greatest use when you are working in a black-and-white mode, but some are useful in full color.

The form for calling XFROPTION is

```
XFROPTION ( OPTION )
```

where OPTION is an expression with a result of type integer, in the range Ø..7. This value selects one of the 8 transfer options. Each of the options specifies a 4-bit logical operation as follows, given a SOURCE color (result from the color table) and an OLD color (existing on the display):

Ø: The default option; result is SOURCE, regardless of the OLD color on the display.

1: The "overlay" option; the result is SOURCE OR OLD. A white dot is unaffected when drawn over with either black or white; a black dot is changed to the SOURCE color. For effects in 16-color modes, see the Standard Device Drivers Handbook.

2: The "invert" option; result is SOURCE XOR OLD. This is a particularly useful option, which works with full color as well as with black and white. If you draw in any color, over any background, the result will probably be clearly visible; and if you subsequently repeat the drawing with the same color, the effect is to erase the drawing and restore the background to what it was originally.

3: The "erase" option; result is (NOT SOURCE) AND OLD. This also works with full color. If you draw something using the default option, and subsequently repeat the drawing with the "erase" option, the result is all black.

4: The "inverse replace" option; result is NOT SOURCE. Same as the default option, except that a "negative" image is produced. For effects in 16-color modes, see the Standard Device Drivers Handbook.

5: The "inverse overlay" option; result is (NOT SOURCE) OR OLD.

6: The "inverse invert" option; result is (NOT SOURCE) XOR OLD.

7: The "inverse erase" option; result is SOURCE AND OLD.

The transfer option can always be set back to its default state by the INITGRAFIX procedure.

NEWFONT and SYSFONT

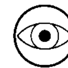
The NEWFONT and SYSFONT procedures are used to change the font used for displaying text as part of a graphic display. NEWFONT changes to a user-defined font, and SYSFONT changes back to the system font.

The form for calling NEWFONT is

```
NEWFONT ( FONT, CWIDTH, CHEIGHT )
```

where FONT is a variable reference; the variable should be an array in which you have created a font definition (see below). CWIDTH and CHEIGHT are expressions with integer values that specify the character height and width of the new font, in dots.

The effect is that character images that are drawn by writing text out to the graphics driver will be taken from the FONT variable. Also, the specified character width and height will be used in moving the cursor when a character is drawn.



NEWFONT has no effect on the font used by the console driver; it only affects text displayed in graphics mode.

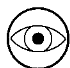
Defining a Font

The font variable's type and size are not checked in any way, and technically the font variable can be any program variable. The graphics procedures interpret the font variable as if it had a type defined as follows:

```
CONST CHEIGHT = {character height in the range Ø..255};
      CWIDTH  = {character width in the range Ø..255};

TYPE CIMAGE = PACKED ARRAY[1..CHEIGHT, 1..CWIDTH]
              OF BOOLEAN;

     FONT = PACKED ARRAY[Ø..127] OF CIMAGE;
```

where the values of CHEIGHT and CWIDTH are the values supplied in the NEWFONT call. In the following discussion we will assume that the program itself declares the type FONT as shown above.

A variable of type FONT is a packed array of 128 variables of type CIMAGE, and each variable of type CIMAGE is one character image. To display a particular character, the graphics driver uses the character's ASCII code as an index into the font variable.



Note that the font variable must contain space for the first 32 ASCII characters, even though they are control characters and cannot be displayed (if written to the graphics driver, they cause actions as described in the Standard Device Drivers Handbook). The displayable ASCII characters are ASCII 32 through ASCII 126, and their images must be in corresponding CIMAGE elements of the font variable.

To draw a character image, the graphics driver uses the DRAWIMAGE procedure described previously; thus each desired image can be set up as the contents of a CIMAGE variable by using the information given under DRAWIMAGE.

GSAVE and GLOAD

You can transfer the contents of the current graphics buffer to a diskette file by using GSAVE; subsequently, the same program or another one can retrieve the stored image into the current graphics buffer by using GLOAD.

The forms for calling GSAVE and GLOAD are

```
GSAVE ( pathname )

GLOAD ( pathname )
```

where the pathname is an ordinary pathname for a diskette file. The file is essentially a data file, but the file type is "Fotofile." Note that no file variable in your program is used for these operations. The specified diskette file is only accessed during execution of GSAVE or GLOAD.

Along with the graphics image, the graphics mode information is stored by GSAVE. GLOAD sets the current graphics mode to the mode that was stored by GSAVE.

The CP280 Mode

The CP28Ø mode is a byproduct of the 4Ø-column, 16-color text display mode of the console driver. It is not a very straightforward mode for graphics, but may be useful in some applications. When used carefully, CP28Ø mode allows you to draw certain kinds of images in full color with twice the horizontal resolution of the COL14Ø mode.

In the CP28Ø mode, the display is 192 dots high and 28Ø dots wide (same dimensions as the BW28Ø mode). The individual dots can be plotted and filled as in the BW28Ø mode, and all 16 colors can be used as in the COL14Ø mode—but with an important restriction as explained below.

Each row of 28Ø dots can be considered as a row of 4Ø cells, each containing 7 dots. Within any particular cell, there can only be two colors which are called the background and foreground colors of the cell.

In the other modes, each dot is considered to have a specific color at a given moment; in the CP28Ø mode it is useful instead to think of each dot as being ON or OFF. A dot that is ON appears in the foreground color of its 7-dot cell; a dot that is OFF appears in the background color of its cell.

The effect of drawing to a dot in the CP28Ø mode is this: the dot is turned ON and the foreground color for the cell that the dot is in is set to the current pen color (possibly altered by the color table and transfer option). This becomes the new color of this dot and of any other dots in the same cell that are ON. The processes that do this are any of the line and dot plotting procedures, the DRAWIMAGE procedure (when it uses a "1" bit in its source data), and text output via the graphics driver (which uses DRAWIMAGE).

The effect of filling a dot in the CP28Ø mode is this: the dot is turned OFF and the background color for the cell that the dot is in is set to the current fill color (possibly altered by the color table and transfer option). This becomes the new color of this dot and of any other dots in the same cell that are OFF. The processes that do this are the FILLPORT procedure, the DRAWIMAGE procedure (when it uses a "Ø" bit in its source data),

and text output via the graphics driver (which uses DRAWIMAGE and thus fills all the dots in the background of each character).

For the effect of transfer options other than the default option, and other information about the CP28Ø mode, see the Standard Device Drivers Handbook (section entitled "The Limited Color Mode" in the chapter on the graphics driver).

Reading from the Graphics Driver

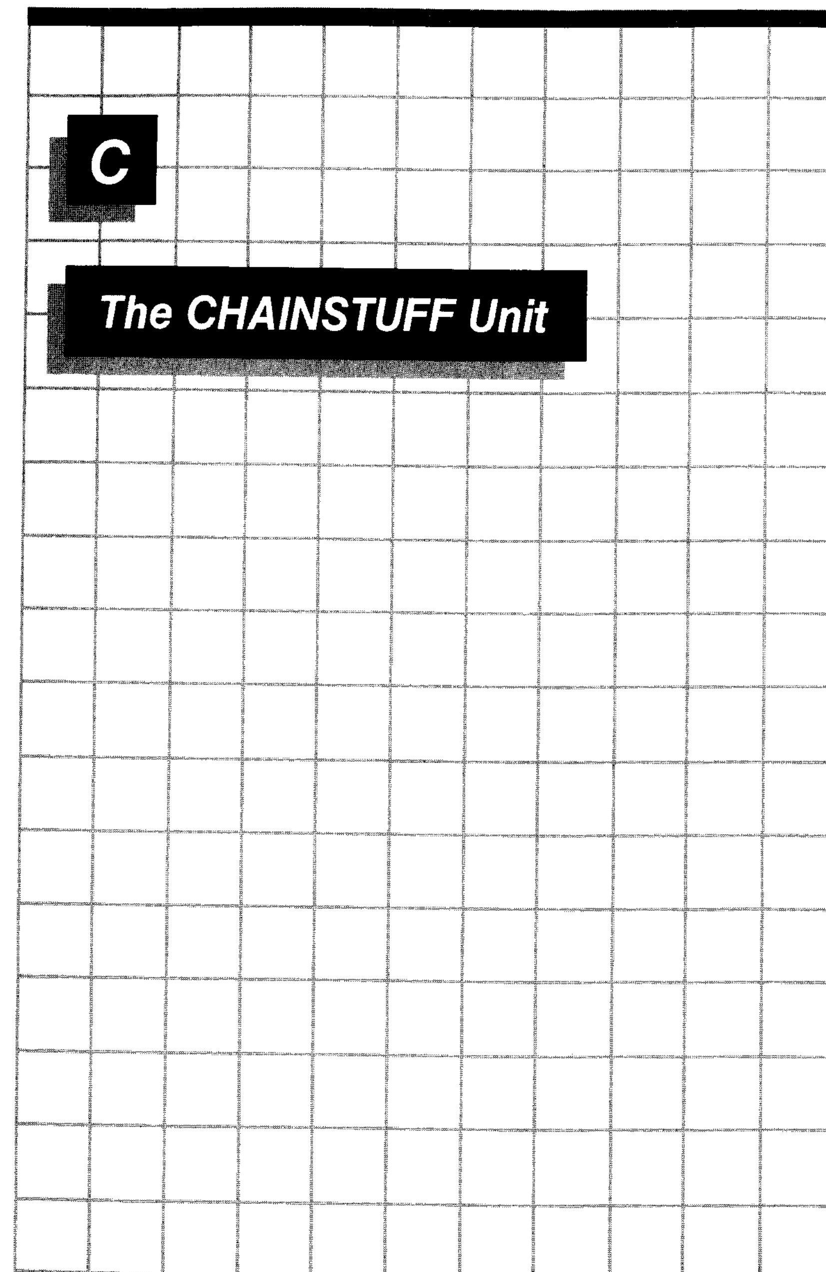
As explained in the Standard Device Drivers Handbook, successive bytes read from the graphics driver indicate the colors found on the display at successive dots, starting at the current cursor position and proceeding downward, upward, to the left, or to the right. The default direction is downward but this can be changed by means of a graphics driver command.

Reading from the graphics driver can be done most conveniently by using UNITREAD (see Chapter 12) with unit number 3 and mode 12.

The PGRAF Interface

```
UNIT PGRAF; INTRINSIC CODE 55 DATA 56;

(* VERSION 1.ØØA *)


INTERFACE

TYPE Screencolor= (Black,Magenta,DarkBlue,Purple,DarkGreen,
                   Grey1,MedBlue,LightBlue,Brown,Orange,
                   Grey2,Pink,Green,Yellow,Aqua,White);
     GMode =       (BW28Ø,CP28Ø,BW56Ø,COL14Ø);
     GBuf  =       1..2;
     XfrMode =     Ø..7;
     GSCBptr =     ^GSCB;
     GSCB =        PACKED RECORD
                     GHMode,GSMode:    CHAR;
                     GPX,GPY:          INTEGER;
                     GVL,GVR,GVB,GVT:  INTEGER;
                     GCF,GCB:          CHAR;
                     GFont:            PACKED ARRAY[Ø..3] OF CHAR;
                     CWidth,CHeight:   CHAR;
                     GColTab:          PACKED ARRAY[Ø..15,Ø..7]
                                         OF CHAR;
                   END;

VAR FotoFile:  FILE;

PROCEDURE GrafixMode(GrfxHMode: GMode; GrfxBuf: GBuf);
PROCEDURE GrafixOn;
PROCEDURE TextOn;
PROCEDURE FillPort;
PROCEDURE PenColor(Color: Screencolor);
PROCEDURE FillColor(Color: Screencolor);
PROCEDURE XfrOption(GrfxSMode: XfrMode);
PROCEDURE SetCTab(Ink,Pixelcolor,Newcolor: Screencolor);
PROCEDURE Viewport(Left,Right,Bottom,Top: INTEGER);
PROCEDURE MoveTo(X,Y: INTEGER);
PROCEDURE MoveRel(DX,DY: INTEGER);
PROCEDURE DotAt(X,Y: INTEGER);
PROCEDURE DotRel(DX,DY: INTEGER);
PROCEDURE LineTo(X,Y: INTEGER);
```

```
PROCEDURE LineRel(DX,DY: INTEGER);
PROCEDURE NewFont(VAR Font; ChrWidth,ChrHeight: INTEGER);
PROCEDURE SysFont;
PROCEDURE DrawImage(VAR Source; SRowSize,SXskip,SYskip,
                    Width,Height: INTEGER);
FUNCTION XYcolor: INTEGER;
FUNCTION Xloc: INTEGER;
FUNCTION Yloc: INTEGER;
PROCEDURE GSave(FName: STRING);
PROCEDURE GLoad(FName: STRING);
PROCEDURE InitGrafix;
```

CHAINSTUFF is an intrinsic unit in the SYSTEM.LIBRARY file. To compile or execute a program that uses CHAINSTUFF, this unit must be either in the SYSTEM.LIBRARY file on the system diskette, or in the program library (see Chapter 14).

This unit allows one program to "chain to" another program. This means that the first program specifies the second one by giving its filename; the system then executes the second program as soon as the first one terminates normally.

The CHAINSTUFF unit also allows the first program to pass a STRING value to the second program; note that this allows almost any information to be passed, since the string can be a filename and can thus specify a communications file containing almost anything.

CHAINSTUFF provides these capabilities in the form of three procedures named SETCHAIN, SETCVAL, and GETCVAL. To use these procedures, the program must have a USES declaration immediately after the program heading:

```
PROGRAM STARTER;
USES CHAINSTUFF;
...
```

The SETCHAIN Procedure

The SETCHAIN procedure call has the form

```
SETCHAIN ( NEXTFILE )
```

where NEXTFILE is a STRING value (up to 8Ø characters). It should be either the name of a code file, or the name of an exec file with the prefix EXEC//. As soon as the program terminates normally, the system will proceed to execute the file whose name is the value of NEXTFILE.

The file is executed exactly as if the X command had been used; thus it is not necessary to supply the suffix .CODE for a code file. Exec files are ASCII files; they do not have a standard suffix attached to their names.

If the program is halted because of any run-time error, the chaining does not occur. Note that this includes a halt caused

by the HALT procedure. However a termination caused by the EXIT procedure is considered a normal termination and the chaining will work.

The SETCVAL Procedure

The SETCVAL procedure call has the form

```
SETCVAL ( MESSAGE )
```

where MESSAGE is a STRING value (up to 8Ø characters). SETCVAL stores the MESSAGE in a system location called CVAL, where it can be picked up by another program.

The GETCVAL Procedure

The GETCVAL procedure call has the form

```
GETCVAL ( MESSAGE )
```

where MESSAGE is a STRING variable whose value is altered by GETCVAL. GETCVAL picks up the current value of CVAL from the system and stores it in the MESSAGE variable. Note that if CVAL has not been set by another program (using SETCVAL), then the value of CVAL is a zero-length string. Once CVAL has been set, it remains set to the same STRING value until it is changed or the system is reinitialized or rebooted.

If you execute a program from the main command line, CVAL is set to an empty string.

An Example of Chaining

Suppose that a diskette named /GAMES contains a collection of game programs whose code files have the following names:

```
CHESS.CODE
CHECKERS.CODE
BLASTOFF.CODE
GOMOKU.CODE
BACKGAMMON.CODE
BLACKJACK.CODE
HEARTS.CODE
SPROUTS.CODE
```

The user could use the Filer to display a list of filenames on the /GAMES diskette, then return to the Command level and use X to execute a selected program. Instead, however, you can write a "front-end" program to display a menu of all the available games; the user chooses one by typing a number, and the front-end program chains to the selected game program:

```
PROGRAM FRONT;
USES CHAINSTUFF;

VAR GAMENUM: INTEGER;

BEGIN
  {Display a greeting}
  WRITELN('Welcome to GAMES!');
  WRITELN;
  {Display the menu}
  WRITELN('Select game from list by typing its number:');
  WRITELN;
  WRITELN('1 -- Chess');
  WRITELN('2 -- Checkers');
  WRITELN('3 -- Blastoff');
  WRITELN('4 -- Gomoku');
  WRITELN('5 -- Backgammon');
  WRITELN('6 -- Blackjack');
  WRITELN('7 -- Hearts');
  WRITELN('8 -- Sprouts');
  WRITELN;
```



```
  {Get a number from the user}
  WRITE('Type a number from 1 to 8, then press RETURN: ');
  READLN(GAMENUM);
  {Make sure the number is valid}
  WHILE NOT (GAMENUM IN [1..8]) DO BEGIN
    WRITE('Number must be from 1 through 8--try again: ');
    READLN(GAMENUM)
  END;
  {Set chaining to filename of selected game}
  CASE GAMENUM OF
    1:  SETCHAIN('/GAMES/CHESS');
    2:  SETCHAIN('/GAMES/CHECKERS');
    3:  SETCHAIN('/GAMES/BLASTOFF');
    4:  SETCHAIN('/GAMES/GOMOKU');
    5:  SETCHAIN('/GAMES/BACKGAMMON');
    6:  SETCHAIN('/GAMES/BLACKJACK');
    7:  SETCHAIN('/GAMES/HEARTS');
    8:  SETCHAIN('/GAMES/SPROUTS')
  END
END.
```

There are several advantages to this. For one thing, the /GAMES diskette may have many other files besides the actual game programs, and this could be confusing to the user.

Many game programs ask the user to type in her name, so it can be used in messages and prompts from the program. You could also have the FRONT program get the user's name and pass it to the selected game program. To do this, the FRONT program can declare a STRING variable, NAME, and then include the following lines either just before or just after the CASE statement:

```
{Get user's name and store it in CVAL}
WRITE('Type your name, please:  ');
READLN(NAME);
SETCVAL(NAME)
```

Now a game program that uses the user's name can obtain it by having its own STRING variable named (for example) UNAME, and then calling GETCVAL:

```
GETCVAL(UNAME)
```

Note that if the FRONT program's codefile is placed on the system diskette and given the name SYSTEM.STARTUP , the FRONT program will be run automatically as soon as the system is booted.

D

The APPLESTUFF Unit

The APPLESTUFF unit is an intrinsic unit in the SYSTEM.LIBRARY file. To compile or execute a program that uses the APPLESTUFF unit, this unit must be either in SYSTEM.LIBRARY or in the program library (see Chapter 14).

The APPLESTUFF unit provides procedures and functions that allow you to do the following:

- Generate random numbers
- Test the keyboard to find out whether a key has been pressed
- Use the joystick inputs
- Generate sounds on the Apple III's speaker
- Read and set information in the Apple III system's internal date and time.

To use the facilities of the APPLESTUFF unit, the program must have a USES declaration containing the identifier APPLESTUFF, immediately after the program heading; for example,

```
PROGRAM MUMBLE;
USES GRAPHICS, APPLESTUFF;
...
```

The public procedures and functions of the APPLESTUFF unit are then available to the program.

The RANDOM Function

RANDOM is an integer function with no parameters. It returns a pseudo-random value uniformly distributed between Ø and 32767. If RANDOM is called repeatedly, the result is a pseudo-random sequence of integers. The statement

```
WRITELN (RANDOM)
```

will display an integer between the indicated limits.

Using the Random Function

A typical application of this function is to get a pseudo-random number, say, between LOW and HIGH inclusive. The expression

```
LOW + RANDOM MOD (HIGH-LOW+1)
```

is sometimes used where results are not critical, but the values formed by this expression are not evenly distributed over the range LOW through HIGH. If you want pseudo-random integers evenly distributed over a range, you can use the following function:

```
FUNCTION RAND (LOW, HIGH:INTEGER; VAR ERROR:BOOLEAN):INTEGER;
  VAR MAX, DIFF, TEMP: INTEGER;
  BEGIN
    ERROR := FALSE;
    IF LOW > HIGH THEN BEGIN
      TEMP := LOW;
      LOW := HIGH;
      HIGH := TEMP
    END;
    { LOW <= HIGH }
    IF LOW < Ø THEN ERROR := HIGH > MAXINT + LOW;
    IF ERROR THEN RAND := Ø { error exit }
    ELSE BEGIN
           DIFF := HIGH - LOW; { Ø <= DIFF <= MAXINT }
           IF DIFF = MAXINT THEN RAND := LOW + RANDOM
           ELSE BEGIN { Ø <= DIFF < MAXINT }
                  MAX := MAXINT -
                         (MAXINT - DIFF) MOD (DIFF + 1);
                  REPEAT TEMP := RANDOM UNTIL TEMP <= MAX;
                  RAND := LOW + TEMP MOD (DIFF + 1)
                END
         END
  END;
```

If HIGH is less than LOW, then the values of HIGH and LOW are exchanged. If the difference between HIGH and LOW exceeds MAXINT, then RAND returns Ø and sets the ERROR parameter to TRUE. Otherwise, RAND returns evenly distributed pseudo-random integer values between LOW and HIGH (inclusive).

Much of the complexity of the RAND function comes from the need to check that the values of HIGH and LOW are within the range

    Ø <= HIGH - LOW < MAXINT

The ARBITRARY function, listed below, is a simpler, faster version of the RAND function. The ARBITRARY function contains no error-checking; it must be called with parameter values within the appropriate range or it will not return a correct result.

```
FUNCTION ARBITRARY (LOW, HIGH:INTEGER):INTEGER;
  VAR MAX, RANGE, TEMP: INTEGER;
  BEGIN
    RANGE := HIGH - LOW + 1;
    MAX := MAXINT - (MAXINT - RANGE + 1) MOD RANGE;
    REPEAT TEMP := RANDOM UNTIL TEMP <= MAX;
    ARBITRARY := LOW + TEMP MOD RANGE
  END;
```

The RANDOMIZE Procedure

RANDOMIZE is a procedure with no parameters. Each time you run a given program using RANDOM, you will get the same random sequence unless you use RANDOMIZE.

RANDOMIZE uses a time-dependent location to generate a starting point for the random number generator.

The KEYPRESS Function

This function, which has no parameters, returns TRUE if there are any characters in the console type-ahead buffer. KEYPRESS does not read the character from CONSOLE or KEYBOARD or have any other effect on I/O. The following statement reads the next character in the type-ahead queue, if any. (CH is a CHAR variable.)

```
IF KEYPRESS THEN READ(KEYBOARD, CH)
```

This statement could be used to retrieve a character typed while the program was doing something else.

Once KEYPRESS becomes true it remains true (so that all characters will be read from the type-ahead buffer) until a GET, READ, or READLN accesses either the INPUT file or the KEYBOARD file, or until a UNITREAD or UNITCLEAR accesses the keyboard device.

The JOYSTICK Procedure

The JOYSTICK procedure has the form

```
JOYSTICK (SELECT, XCOORD, YCOORD, BØ, B1)
```

where SELECT is an integer treated modulo 2 to select one of the two joysticks numbered Ø and 1. XCOORD and YCOORD are integer variables which are used to return the coordinates of the selected joystick. The values returned in XCOORD and YCOORD are in the range Ø to 255. BØ and B1 are boolean variables which are used to return the status of the two buttons associated with the selected joystick. The values returned in BØ and B1 are TRUE for a button that is pressed or FALSE for a button that is not pressed.

The SOUND Procedure

The SOUND procedure has the form

```
SOUND (PITCH, DURATION, VOLUME)
```

where PITCH is an integer from Ø through 86, DURATION is an integer from Ø through 255, and VOLUME is an integer from Ø through 63.

A PITCH of Ø is used for a rest, and 2 through 86 yield a tempered (approximately) chromatic scale. DURATION is in arbitrary units of time, and VOLUME is in arbitrary units of loudness.

SOUND (1,1,63) gives a click.

A chromatic scale is played by the following program:

```
PROGRAM SCALE;

USES APPLESTUFF;
VAR PITCH, DURATION, VOLUME: INTEGER;

  BEGIN

    DURATION := 1ØØ;
    FOR PITCH := 12 TO 24 DO
      SOUND (PITCH, DURATION, 63)

  END.
```

Use of this procedure requires the standard device driver .AUDIO. See the Standard Device Drivers Handbook to find out how to add this driver to your SOS.DRIVER file.

The Internal Date and Time

The DATE, TIMEOFDAY, CLOCKINFO, and SETTIME procedures let you read and set information in the Apple III system's internal date and time.

The DATE procedure has the form

```
DATE ( D )
```

where D is a string variable to contain the information returned by DATE. The returned string has the format "YYYYMMDD", where YYYY is the year, MM is the month (as a two-digit number), and DD is the day of the month. For example, May 28th, 1945 would be represented as "1945Ø528".

The TIMEOFDAY procedure has the form

```
TIMEOFDAY ( T )
```

where T is a string variable to contain the information returned by TIMEOFDAY. The returned string has the format "HHMMSS", where HH is the hour (24-hour format), MM is the minute, and SS is the second. For example, the time 3:44:Ø6 PM would be represented as "1544Ø6".

The CLOCKINFO procedure has the form

```
CLOCKINFO ( YEAR, MON, DAY, DAYOFWK, HR, MIN, SEC, THOU )
```

where all of the parameters are integer variables used to contain the date and time information returned by CLOCKINFO. After a CLOCKINFO call, the variables have the following values:

YEAR — the current year.

MON — the number of the current month (1 for January to 12 for December).

DAY — the day of the month.

DAYOFWK — the number of the day of the week (1 for Sunday to 7 for Saturday).

HR — the hour in 24-hour format.

MIN — the minute.

SEC — the second.

THOU — the millisecond.

The SETTIME procedure lets you set the internal system date and time from a program.

The SETTIME procedure has the form

```
SETTIME ( T )
```

where T is a string of eighteen characters representing the date and time to be recorded internally by the system. T has the format

```
YYYYMMDDWHHNNSSUUU
```

where the fields are defined as

Field	Description	Range
YYYY	year	ØØØØ - 9999
MM	month	Ø1 - 12 (Jan - Dec)
DD	date of the month	Ø1 - 31
W	day of the week	1 - 7 (Sun - Sat)
HH	hour in 24-hour format	ØØ - 23 (midnight to 11 pm)
NN	minute	ØØ - 59
SS	second	ØØ - 59
UUU	millisecond	ØØØ - 999

SETTIME checks that each field is in range, but it does not check consistency between fields. For example, it does not recognize February 31 as an erroneous date because Ø2 is a valid month and 31 is a valid date. Any field that is out of range will be set to zero by SETTIME.

PADDLE, BUTTON, and NOTE

The PADDLE and BUTTON functions and the NOTE procedure provide compatibility with Apple II Pascal programs. The PADDLE function has the form

```
PADDLE (SELECT)
```

where SELECT is an integer treated modulo 4 to select one of the four paddle inputs numbered Ø, 1, 2, and 3. PADDLE returns an integer in the range Ø to 255 which represents the position of the selected paddle.

The BUTTON function has the form

```
BUTTON (SELECT)
```

where SELECT is an integer treated modulo 4 to select one of the four button inputs numbered Ø, 1, 2 and 3. The BUTTON function returns a BOOLEAN value of TRUE if the selected game-control button is pressed, and FALSE otherwise.

The NOTE procedure has the form

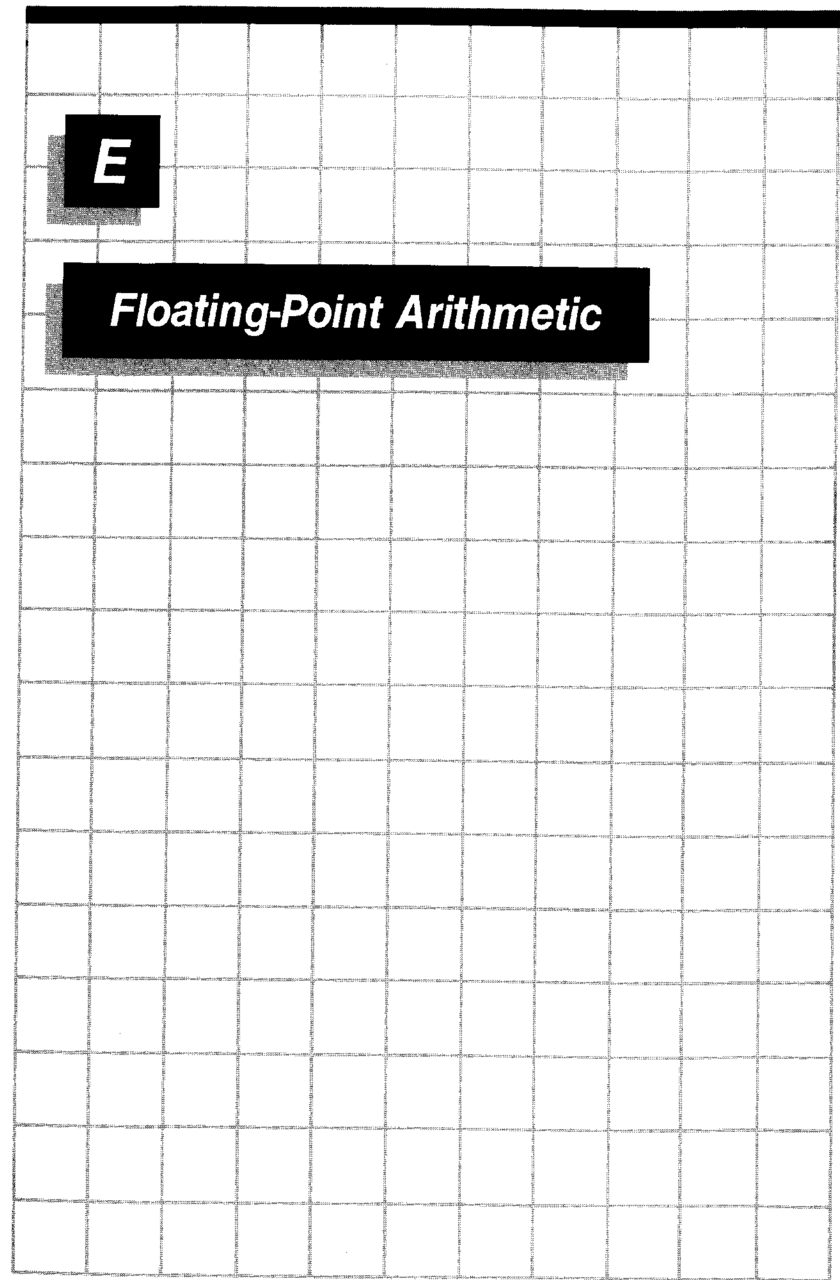
```
NOTE (PITCH, DURATION)
```


where PITCH is an integer from Ø through 86 and DURATION is an integer from Ø through 255.

A PITCH of Ø is used for a rest, and 2 through 86 yield a tempered (approximately) chromatic scale. DURATION is in arbitrary units of time.

NOTE (1,1) gives a click.

To use the NOTE procedure you must have the driver .AUDIO in your SOS.DRIVER file. (See the <u>Standard Device Drivers Handbook</u> for details.)



E

Floating-Point Arithmetic

Introduction

This appendix describes the Apple III Pascal implementation of the IEEE floating-point proposal. The information in this appendix is needed only if you are concerned with the best possible accuracy in floating-point calculations.

By default, the improved arithmetic will not do anything unexpected except that it provides "gradual underflow" of arithmetic results. Most Pascal programs written for other UCSD-based systems and recompiled for the Apple III will produce the same arithmetic results, except for improved performance in cases where underflow occurs.

The arithmetic also eliminates artificially imposed limitations on the development of algorithms, providing numerical analysts with additional tools.

This appendix includes two main sections: a description of the floating-point system and an explanation of the functions and procedures that enable you to configure the arithmetic environment. At the end of this appendix is a table that summarizes the Apple III Pascal floating-point system.

IEEE Floating-Point Standard

Recently, the IEEE appointed a committee to study issues of floating-point arithmetic such as the problems of limited precision, finite range, and varying treatment by different computers of overflow, underflow, and division by zero. The committee created a Proposed Standard for Binary Floating-Point Arithmetic (Draft 8.Ø of IEEE Task P754). The floating-point operations available in Apple III Pascal conform to this standard (single precision only).

The standard specifies

- accuracy of arithmetic, I/O conversions, remainder, and square root;
- internal number formats;
- rounding modes;
- special optional treatment of exceptions like overflow and division by zero; and
- configurable arithmetic under program control.

The purpose of the standard is to

- allow programs written in many different environments to run on computers that adhere to the standard; and
- improve diagnostics and error handling.

Definitions

The following definitions are for terms used by the IEEE floating-point standard.

Binary Floating-Point Number. A 32-bit string characterized by three components: a sign, a signed exponent, and a significand. Its numeric value, if any, is the signed product of its significand and 2 raised to the power of its exponent.

Exponent. The component of a binary floating-point number that normally signifies the power to which 2 is raised in determining the value of the represented number. Occasionally, the exponent is called the signed or unbiased exponent.

Biased Exponent. Exponents are stored as values that range from Ø to 255. For normalized numbers, the biased exponent equals the unbiased exponent plus 127.

Significand. The 24-bit component of a binary floating-point number that consists of the implicit bit to the left of the binary point and the fraction field to the right of the binary point. The implicit bit is not stored.

Fraction. The 23-bit field of the significand that lies to the right of its implied binary point.

NaN. Not a Number. Special 32-bit quantities that are generated automatically when the result of an arithmetic operation could not otherwise be specified (for example, Ø/Ø). The internal format of the NaN contains a code that describes the circumstances in which it was generated. If a NaN is an argument of an arithmetic operation, the result will be a NaN. This allows programs to run to completion when they would otherwise be forced to abort.

Exceptions. The IEEE Proposed Standard for Binary Floating-Point Arithmetic specifies a list of exceptions--special cases in arithmetic, comparisons, remainder, and square root. The response to these exceptions is specified to insure a uniform arithmetic environment.

Exception Signal. Associated with each exception is a signal that can be set, cleared, and tested. It is set whenever a given exception occurs and stays set until it is cleared.

Rounding. When the result of an arithmetic operation cannot be represented exactly as a binary floating-point number (for example, 1/3 or 1/1Ø), a decision of how to round the result must be made. There are four rounding methods that can be selected. These methods are described in a later section of this appendix.

Infinities. Infinities are signed quantities that behave like very large numbers. They are generated by overflows and division by zero, and can be arguments in arithmetic operations and comparisons.

Normalized Numbers. The storage format of all binary floating-point numbers except infinities, NaNs, and denormalized numbers (certain values at the underflow threshold). Normalized numbers are characterized by the assumption of a leading 1 in the significand.

Denormalized Numbers. A special treatment of underflow may produce a nonzero number when an Apple II program would have aborted. These numbers are characterized by a special format that is not normalized (the leading bit of the significand is Ø and the exponent is $^{-126}$).

Exceptions

The IEEE standard lists the following set of exceptional conditions in floating-point arithmetic:

Overflow
Underflow
Division by zero
Inexact result
Invalid operation

Apple III Pascal adds a sixth exception to this list:

Integer conversion

The integer conversion exception is signaled by TRUNC and ROUND if their arguments exceed the bounds of the predeclared type INTEGER.

Associated with each exception is a "sticky" signal, which is set each time the exception occurs and is only cleared by an explicitly programmed call on the SETXCPN procedure.

Each exception can cause the program to halt. The programmer controls whether or not the occurrence of an exception halts the program. The section on exception handling describes how to choose halt or continue for each exception, and what the result is for each exception when it occurs.

To test whether or not an exception occurred during the evaluation of an expression or procedure, be sure to clear the signal before evaluating the expression or procedure, then test the signal after evaluating the expression or procedure.

Overflow

The overflow exception occurs when a correctly rounded result is larger than the largest normalized single-precision real number. The table below shows that number.

Storage Format	Meaning	Decimal Value
Sign= Ø,1 Exponent= 254 Fraction= all ones	$2^{254-127}$ *1.11..1	3.4Ø2823E38

The default response to an overflow is a program halt.

Underflow

The standard specifies a treatment of underflow called "gradual underflow". On many computers, a single-precision result less than the underflow threshold (2^{-126} on Apple II Pascal) will cause a zero result. For example,

if A = 2^{-126} then A/2 = Ø

The standard specifies that if the exponent of a number is smaller than -126, the significand is right-shifted (denormalized) until the number is correctly represented. For example, representing the significand in binary,

A=1.x2^{-126}

A/2=Ø.1x2^{-126}

A/4=Ø.Ø1x2^{-126}

A/2^N=Ø.Ø..Ø1x2^{-126} (with N leading zeros)

If N is greater than 23, A is set to Ø.

This procedure is called gradual underflow, and it reduces the impact of underflow to be comparable to rounding errors. The default response to an underflow exception is to continue.

The underflow exception occurs when the magnitude of a nonzero result is:

- in normalizing mode, less than 1.1754944 x 1Ø$^{-38}$ (2^{-126}) (result is not necessarily Ø); or

- in warning mode, less than 1.1754944 x 1Ø$^{-38}$ (2^{-126}) and further denormalized. For example,

  2^{-140} * 2 is not an underflow, but

  2^{-140} / 2 is an underflow
  (the result will not necessarily be Ø.)

Division by Zero

The division by zero exception occurs in a division operation when the divisor is Ø and the dividend is a finite non-zero number (for example, 2/Ø). The default is to halt the program. If continue is set, the result is infinity with the proper sign.

Division of Ø by Ø is a special case, covered in the section on invalid operations.

Inexact Result

The inexact exception occurs when a result has been rounded or has overflowed. The default response to an inexact result is to continue.

Invalid Operations

This exception arises in a variety of arithmetic operations. Any exception other than overflow, underflow, division by zero, and inexact result falls in the category of invalid operations. Invalid operations are exceptions that do not occur frequently enough to deserve special classification.

The following events are invalid operations:

- if the argument of a function is a NaN that causes an invalid operation signal (see NaN section);

- addition or subtraction of infinities in Projective mode; or (+infinity)-(+infinity) or (+infinity)+(-infinity) in Affine mode;

- multiplication of Ø * infinity;
- division of:
  - zero by zero,
  - Infinity/Infinity, or
  - in warning mode, A/X where A is finite and not zero, and X is denormalized;
- Remainder x REM y, where:
  - y is zero or denormalized in warning mode, or
  - x is infinite;
- Square Root if the operand is :
  - less than zero, or
  - infinity in the projective mode, or
  - denormalized in warning mode;
- conversion of a single-precision real to an integer when overflow or infinity make a correct conversion impossible;
- comparisons using <,<= , >=, or > when the relation is unordered.
- SCALB in warning mode, when its operand is denormalized and the result would be normalized.

The default result of an invalid operation is to halt the program.

Floating-Point Format

This section describes the format of the numbers used by the floating-point system. A normalized, single-precision number has the form

$$X = +/- \ 2^{E-127} * (1.F)$$

The number X above is represented in storage by the bit string

byte 3 (high addr)		byte 2	byte 1	byte Ø (low addr)
S	exponent	fraction		
31	3Ø 23	22		Ø

Where:

+/- = sign bit (+ is Ø, - is 1),

E = exponent, and

F = X's 23-bit fraction that, together with an implicit leading 1, is the significand. The significand ranges between 1.ØØ..Ø and 1.11..1 (since the leading bit is always 1, it is not stored.)

These numbers offer the same precision (slightly more than 7 significant decimal places) as the DEC PDP-11 format and slightly more than the IBM 37Ø short format.

In addition to normalized numbers certain other special symbols are required. These are +/- Ø (the sign is only regarded in division), infinities (the sign bit is sometimes ignored), NaNs, and denormalized numbers (used to cope with underflow).

The following table presents the format for normalized numbers, Ø, denormalized numbers, infinities, and NaNs. For each type of number, the table gives the range for the sign, exponent, and fraction. It also gives an interpretation for each type of number.

SPECIFIC NUMBER FORMATS

Type	Sign	Exponent	Fraction	Interpretation
Normalized Number	Ø,1	1 to 254	Any	$(-1)^S * 2^{E-127} * (1.F)$
Zeros	Ø,1	Ø	Ø	$(-1)^S * Ø$
Denormalized	Ø,1	Ø	Non-Zero	$(-1) * 2^{S-126} * (Ø.F)$
Infinities	Ø,1	255	Ø	+/- Infinity
NANs	Ø,1	255	Non-Zero	(See NaN table)

Arithmetic with Denormalized Numbers

Because gradual underflow and denormalized numbers are not generally familiar, the standard allows arithmetic operations on denormalized numbers in two modes, normalizing mode and warning mode.

Normalizing mode (the default) is recommended for most application development.

Warning mode is intended for use when numerical programs are to be transported from a non-IEEE floating-point system. Warning mode prevents promotion of denormalized numbers to normalized numbers. In Warning mode, if a multiplication or division would promote a denormalized operand to a normalized result, the program signals an invalid operation and returns a NaN instead. For example, the operation

$$2^{-145} * 2^{100} = 2^{-45}$$

signals the invalid operation exception and produces a NaN

instead of 2^{-45}. This is done to prevent an undetected loss of precision caused by denormalization. Note that this cannot occur in addition or subtraction. (In Normalizing mode, the answer is produced and no warning is given.)

In Warning mode, underflow is signaled only if "further denormalization" occurs. For example,

$2 * 2^{-145}$ would not cause an underflow signal, but

$2^{-145} / 2$ would.

(Note that in normalizing mode, both operations would cause the underflow signal.)

Infinity Arithmetic and Comparisons

On other systems, overflows or division by zero may cause unpredictable results and sometimes halt the program. An alternative is provided by the standard in this floating-point system. The halt switches can be turned off (see the section on halts) and infinities can be created whenever overflow or division by zero occurs. Infinities may be positive or negative.

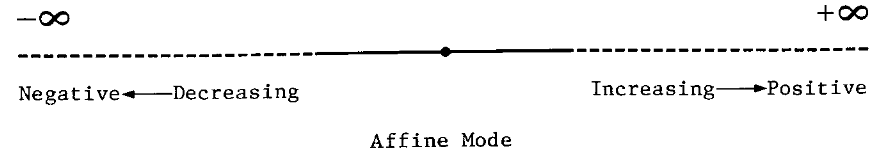
Affine and Projective Modes

Arithmetic operations and comparisons of the infinities are done in either of two modes:

Affine (the default), or
Projective.

Affine mode creates a real number system with two infinities, one positive and one negative:

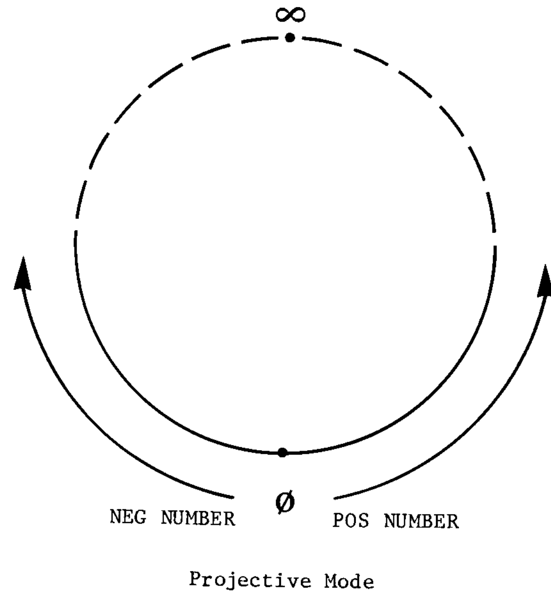
Linear Infinities



Affine Mode

Affine mode is recommended for nearly all applications of infinity arithmetic.

Projective mode creates a real number system with one infinity. That number system has the real line as a circle with the negative and positive real axes meeting at infinity:



Projective Mode

In arithmetic operations, the only difference between Affine and Projective mode is when both arguments are infinity. In that case, in Projective mode additions and subtractions always set the invalid operation exception signal. There are more significant differences in comparisons.

Rules for Infinity Arithmetic

The two tables below show the results of arithmetic operations on infinities, with halts disabled to permit a program to continue after division by zero, invalid operations, overflow, and underflow. The appropriate exceptions are always signaled. For multiplication and division, the sign of the result is + if the operands have the same sign, and - if they have different signs. Any operation involving a NaN produces a NaN.

Table of rules for arithmetic involving infinities

RESULTS OF ARITHMETIC WITH INFINITIES

		operand 2		
	operand 1	- infinity	number	+ infinity
Addition	- infinity number + infinity	- infinity* - infinity NaN	- infinity number[a] + infinity	NaN + infinity + infinity*
Subtraction	- infinity number + infinity	NaN + infinity + infinity*	- infinity number[a] + infinity	- infinity* - infinity NaN

		operand 2		
	operand 1	Ø	number	infinity
Multiplication	Ø number infinity	Ø Ø NaN	Ø number[a] infinity	NaN infinity infinity
Division	Ø number infinity	NaN infinity infinity	Ø number[a] infinity	Ø Ø NaN

* This value is a NaN in projective mode
[a] This value can be an infinity if the operation overflows

Rules for Comparisons

The rules for comparison operations are summarized below. In traditional computers when two operands a and b are compared, there are only three possible outcomes:

a = b
a < b
a > b

In Apple III Pascal, there is a fourth possibility:

unordered

- A NaN is unordered with respect to all real values, including other NaNs and itself.

- In Projective mode, infinity is unordered with respect to all finite values (+/- infinity is equal to +/-infinity).

The comparison of two unordered values by means of a relational operator (>, <, =, >=, <=) will always yield "false" as a result. Every comparison other than equals signals an invalid operation.

Input and Output of Infinities

When an infinite value is output via WRITE or WRITELN, it is always represented by two or more contiguous "+" signs if it is positive, or by two or more "-" signs if it is negative.

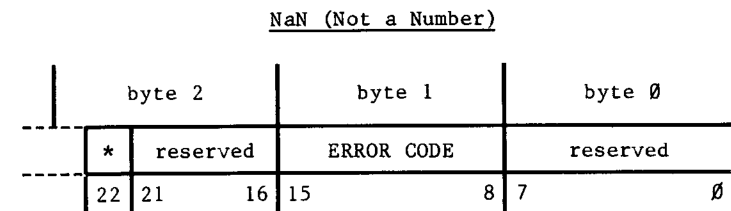
The written value is always preceded by at least one space. If the width specification is greater than 2, the field is filled with "+" or "-" signs. If there is a "decimal places" specification, it is ignored.

An infinite value written out in this manner can be read back correctly by READ or READLN.

NaNs

To provide for the handling of exceptional conditions, the standard specifies 32-bit quantities that are used as diagnostics. These diagnostics, called NaNs ("Not a Number"), may be generated by the programmer or as the result of an exceptional condition. In either case, they will be propagated when arithmetic operations are performed upon them unless an invalid operation causes a halt. The figure below shows the internal format for a NaN.

NaN (Not a Number)



*Invalid Operation bit

The subfields in the figure are interpreted as follows:

- Sign bit: ignored

- Signaling Bit: determined by the class of the NaN. NaNs are divided into two classes. One (a signaling NaN) signals an invalid operation when an arithmetic operation is performed upon it; the other (a propagating NaN) does not. A signaling NaN has a 1 in the Signaling Bit. A propagating NaN has a Ø.

- Error Code: This value (bits 8-15) indicates the circumstance in which the NaN was generated. The codes are explained below.

- Reserved Bit: Bits 1-7 and 16-22 are reserved for future use.

An arithmetic operation on a signaling NaN signals an invalid operation and results in a propagating NaN with the same error code. If no program halt occurs, the Signaling Bit is turned off, and the NaN propagates.

If an operation involves one NaN, the result will be that NaN. If an operation involves two NaNs, then the result will be the NaN with the greater error code.

The following table presents the error codes and their meanings.

NAN ERROR CODES

Error Code	Meaning
ØØ	Reserved
Ø1	Invalid square root
Ø2	Invalid affine addition of infinities
Ø4	Invalid division, such as Ø/Ø
Ø7	Invalid projective addition of infinities
Ø8	Invalid multiplication, such as Ø*infinity
Ø9	Invalid remainder or modulo
ØA	Invalid parameter to base conversion routine
ØC	Warning mode: normalized result from denormalized operand
11	Invalid decimal to binary conversion: syntax
12	Decimal to binary conversion: NaN in source
13	Decimal to binary conversion: unrepresentable value
21	Trig radian argument reduction error: argument exceeds capacity of conversion routine
FF	Reserved

NaNs appear in a Pascal program in the following situations:

- as a result of invalid operations when no exception halt occurs (see the section on halts). These are always propagating NaNs.

- as the result returned by the MAKENAN function in the REALMODES unit. These NaNs can (at the programmer's choice) cause the invalid operation exception to occur and may be used for debugging.

Accuracy

The following sections describe these aspects of the floating-point system that affect the accuracy of arithmetic operations: rounding modes, the inexact signal, and input/output conversions.

Accuracy of Arithmetic Operations

The standard requires the following arithmetic operations: +, -, *, /, remainder, round-to-integer, and square root. The standard specifies performance down to the last bit. Remainders that do not underflow are computed without rounding error.

Rounding Modes and the Inexact Signal

If the result of an arithmetic operation is exactly representable in the single-precision format, that result will be returned. Otherwise, the result will be rounded. There are four rounding modes:

- round to nearest value, with ties going to the even value (this is the default);

- round toward zero (truncate);

- round up; and

- round down.

Here is an example showing the rounding modes. Assume Z cannot be represented as a single-precision real. If Z is the exact result of an arithmetic operation and X1 and X2 are the closest single-precision real values for which X1 < Z <X2, then the rounding modes function as follows:

- Round to nearest (Z) = the nearer of X1 and X2 to Z. In the case of a tie, choose the one that has a Ø in its least significant bit. Ties round to even.
- Round toward zero (Z) = smaller of X1 and X2 in magnitude.
- Round up (Z) = X2.
- Round down (Z) = X1.

A rounding operation during an arithmetic operation sets the inexact signal.

Input/Output Conversions

The use of floating-point arithmetic requires the conversion of numbers from decimal to binary on input and from binary to decimal on output. The error that occurs in these conversions will be less than 1 unit of the destination's least significant digit. The I/O conversions are used by:

READ and READLN
WRITE and WRITELN

This section describes how real values may be written to and read from text files, using the built-in procedures READ and WRITE. READLN and WRITELN work similarly to READ and WRITE, respectively. Since text files represent numbers in decimal notation, and the computer uses a binary representation internally, such input and output require number base conversions from decimal to binary and binary to decimal. Base conversions have rarely been done accurately in a way that permits simple error bounds to be put on the results.

The proposed IEEE standard for Binary Floating-Point Arithmetic specifies accuracy and other desirable properties of decimal <---> binary conversions, which Apple III Pascal follows. In addition, several Pascal standards groups (IEEE, ANSI, and ISO) have tentatively agreed on some cosmetic details of READ and WRITE, that will make it easier to format reports and predict what your output will look like. We have tied to follow their suggestions, too.

Reals appear as character strings in two different contexts: as source code submitted to the Compiler (real constants), and

as text files written and read by Pascal programs. The syntax of real numbers applies in both cases. The Compiler converts character strings to numbers differently than READ and WRITE do. The main differences are:

- the Compiler can't use infinities and NaN's (to get these into your program, use the REALMODES functions described below); and
- the Compiler uses a simpler and less accurate method of decimal-to-binary conversion than READ and WRITE.

For READ and WRITE, positive infinity is represented by a string of at least two plus signs, and negative infinity by a string of at least two minus signs. NaNs are represented by the characters "NaN", with an optional leading sign, and an optional trailing quoted string of characters, as follows:

-NaN'ØØ1.ff.5'

The character string will be used in future versions to provide diagnostic data.

Input: Decimal to Binary

When READ expects a real number, it searches for the first nonblank character, which is assumed to be the first character of the real. READ throws away any blanks it finds in the meantime (for this purpose, carriage returns are counted as blanks). All subsequent characters, up to but not including the first character failing to satisfy the syntax of a real number, are assumed to belong to the real. The file's window variable is left pointing at the delimiting character.

If the first non-blank characters can't be interpreted as a real number, or if an end-of-file (eof) was encountered before a real could be found, a syntax error results. This signals the invalid operation exception, and returns a syntax error NaN (see the table of NaN error codes below). An eof may delimit the last real in the file, acting as the "first character failing the syntax".

When reading a real number, digits and decimal points are interpreted in the usual way. "E" can be read (roughly) as "times ten to the -- power". Any number of digits can be read, and all of them will contribute to the conversion. The exponent

part has a range of Ø to 99. Conversion of NaNs and infinities raises no exception.

The decimal-to-binary conversion signals underflow whenever a nonzero input produces a zero or denormalized result. It signals overflow whenever a finite input exceeds the largest representable number. Both underflow and overflow are checked after rounding according to the current rounding mode, which is the only element of the numerical environment that governs the conversion. The directed roundings "round up" and "round down" guarantee, in addition to standard accuracy, that the binary number returned is an upper or lower bound, respectively, of the decimal number input.

Output: Binary to Decimal

For writing real values, WRITE statements take parameters of three forms:

```
REXP:E1
REXP:E1:E2
REXP
```

where REXP is an expression of type REAL, and E1 and E2 are expressions of type INTEGER. E1 is called the "width expression", and gives a minimum number of characters to be written. E2 is called the "decimal places" expression, and asks for a specific number of digits to appear to the right of the decimal point.

REXP:E1 asks for the value "REXP" to be written as

```
-x.xxxxxxxE+yy
```

In this "floating" form, the signs will vary, but the form will always include one digit to the left of the point and a two-digit exponent with a sign. The value of "REXP" is rounded (according to the current mode) to the number of digits needed to fill up the field width given by E1.

REXP:E1:E2 asks for the value "REXP" to be written as

```
bbbbb-xxxx.yyy
```

In this "fixed-point" form, there is no exponent, and the value "REXP" is rounded (according to the current mode) to E2 decimal

places to the right of the point. The number of digits to the left of the decimal point are implied by the magnitude of REXP. Enough blanks are padded on the left to fill out the field width given by E1. If the width E1 is insufficient, it is ignored and as many characters are written as are needed to represent the value of REXP with E2 decimal places.

The parameter REXP (without E1 and E2) asks for the default "floating" form with E1 set to 12. This gives six significant digits of precision.

If E2 is missing, any E1 less than 8 is increased to 8. If E2 is present, any E1 less than E2 + 3 is increased to E2 + 3. Then, if E1 is greater than 8Ø, E1 is decreased to 8Ø, and if E2 is present, it is decreased by an equal amount.

A zero value is always written "Ø.Ø", regardless of E1 and E2. It is padded with blanks left and right to fill the given field width and to keep its decimal point aligned with those of other values written with the same WRITE parameters.

Expert's Corner

In Warning mode, denormalized numbers are written as described above, except that in the "floating" format, at least one leading zero digit is written to indicate the possible loss of precision due to denormalization. Furthermore, the exponent written in Warning mode is not allowed to fall below -38, which makes the number of leading zeros a rough indicator of the amount of denormalization.

A final note on accuracy, and the relation between the input and output base conversions: It is a curiosity of the mathematics of base conversion that 9 is the minimum number of decimal digits required to distinguish different binary values, although 1Ø^9 > 2^24. Thus, an interval of 9-digit decimal values is mapped into a single binary value by the input routine. This guarantees that there are (9-digit) decimal values d for which dec(bin(d)) = d fails. Our implementation keeps the conversion errors small enough that for all representable binary values b, bin(dec(b)) = b if the rounding mode is "round to nearest". For length(d) <= 6 digits, dec(bin(d)) = d as well.

Real Arithmetic Environments

The settings of the exception signals, the halt switches, and the arithmetic mode settings for rounding, closure, and handling of denormalized numbers define the arithmetic environment. The REALMODES unit, described in Appendix A, contains functions and procedures that enable you to:

- set or check an exception signal,
- arm or disarm halting on exceptions,
- set and check the arithmetic modes for rounding, closure on infinities, and denormalized numbers,
- save and restore a numeric environment, and
- use supporting functions (such as ISNAN and SCALB).

REALMODES Unit

Using procedures contained in the REALMODES unit, a program can check or modify the mode settings or the response to exceptions from the default settings. To use any of these procedures, a program must have a USES declaration containing the identifier REALMODES immediately after the program heading; for example,

```
PROGRAM DESIGN;
USES REALMODES
```

The public functions of the REALMODES unit are then available to the program.

The REALMODES unit defines five data types that can be used in declaring program variables. They are declared as follows:

```
XCPN     = (UNDERFL, OVERFL, DIVØ, CVTOVFL, INXACT, INVOP)
RMODE    = (RNEAR, RPOS, RNEG, RZERO)
CLOSURE  = (PROJ, AFFINE)
DENORM   = (WARNING, NORMALIZING)
NUMENV   = Array [Ø..2] OF Integer
```


Exception Handling

The XCPN type provides identifiers for the six kinds of exception:

```
OVERFL     -- The overflow exception
UNDERFL    -- The underflow exception
DIVØ       -- The division by zero exception
CVTOVFL    -- The integer conversion overflow exception
INXACT     -- The inexact result exception
INVOP      -- The invalid operation exception
```

These types (or a variable or expression of type XCPN) are used as parameters for the SETXCPN, GETXCPN, SETHALT, and GETHALT procedures. Associated with each type is a boolean signal and a boolean halt switch. At the end of this section is a table summarizing the exception halts and switches.

Signals

A boolean signal is set true by an occurrence of its corresponding floating-point exception or by an explicitly programmed call on SETXCPN. It is set false only by a call on SETXCPN. GETXCPN returns the current state of the signal asked for.

Halts

If an exception halt switch is true, the corresponding exception causes a program halt. SETHALT arms or disarms the halt switch (i.e., sets it true or false). GETHALT returns the current state of the halt switch.

The occurrence of an exception, when its corresponding halt switch is armed, causes the program to halt with a diagnostic message on the screen. (The section on diagnostics in the main text of this manual tells you how to use the diagnostic message to find where in your program the error occurred.)

Pressing the spacebar reinitializes the Pascal operating system and loses the current state of your program.

Arithmetic Modes

Using the functions and procedures described below, a programmer controls the way in which floating-point arithmetic

- rounds inexact results of arithmetic operations,
- includes infinities in the number system (through closure), and
- handles denormalized numbers.

Rounding

The RMODE type names the four rounding modes:

RNEAR -- round to nearest representable value; in case of a tie, use the value that has a Ø in the least significant bit.

RPOS -- round in the positive direction.

RNEG -- round in the negative direction.

RZERO -- round toward Ø.

A call on the procedure, SETROUND (R), sets the rounding mode to R, where R is of type RMODE.

The GETROUND function returns a result of type RMODE, that is, the current round mode.

Infinities and CLOSURE Modes

The CLOSURE type names the two ways of including infinity in the number system:

AFFINE -- affine closure, and
PROJ -- projective closure.

A call on the procedure, SETCLOS (C), sets the closure mode to C, where C is of type CLOSURE.

The GETCLOS function returns a result of type CLOSURE, that is, the current mode.

Handling Denormalized Arithmetic

The DENORM type names the two modes for handling denormalized operands:

NORMALIZING -- normalizing mode, and
WARNING -- warning mode.

A call on the procedure, SETDNORM (D), sets the indicated mode, where D is of type DENORM.

The GETDNORM function returns a result of type DENORM, that is, the current mode setting.

Numeric Environment

The current settings of the exception signals, the halt switches, and the arithmetic mode settings for rounding, closure, and handling of denormalized numbers all together define the current state of a numeric environment. Since it takes 15 procedure calls to set up a complete numeric environment, making many changes to a numeric environment can be tedious, especially if you plan to restore the old settings.

For example, a subroutine from a library may require certain mode settings, and, at the same time, a programmer may want to protect the exception signals from the actions of the library routine. For this purpose, the type NUMENV provides a place to store (in encoded form) all the data necessary to set up a numeric environment.

The purpose of this type is to permit entire environments to be saved and restored by calls on the procedures:

SAVENV (E)
RESTENV (E)

A call on the procedure, SAVENV (E), stores the current state of the numeric enviroment in the variable E, using the code. (Note that E must be a variable of type NUMENV, and not an expression.) RESTENV sets up the numeric environment according to the code in E, a variable of type NUMENV. RESTENV (E), when E hasn't previously been loaded by a call on SAVENV (E), can

produce very strange results.

Don't try to set signals or switches, or control arithmetic modes, by changing the contents of a NUMENV variable. The codes will be changing from version to version, and you will encounter the strange results just mentioned.

The IEEE proposed standard for binary floating-point arithmetic specifies the following modes as the defaults:

Round to nearest
Warning mode
Projective mode
No halts on exceptions

To make the numeric environment conform to the IEEE settings, embed the following code in your program:

```
USES REALMODES;

VAR  E: XCPN;

BEGIN

     FOR E:=INVOP TO INXACT DO SETHALT (E, FALSE);
     SETCLOS (PROJ);
     SETDNORM (WARNING);

END;
```

(There are many ways to embed this code in a program; choose the one that best fits your situation.)

Supporting Functions

In addition to the procedures and functions for controlling the numeric environment, REALMODES provides a set of useful functions related to the special capabilities of Apple III Pascal arithmetic. In all of the following descriptions, X and Y represent any values of type real (including infinite and NaN values).

INTEGRAL (X:REAL):BOOLEAN returns TRUE if the value of X is an integer (in the mathematical sense, not the Pascal sense.) It returns FALSE otherwise.

FINITE (X:REAL):BOOLEAN returns TRUE if X is a finite numeric value, FALSE if it is a NaN or an infinity.

ISNAN (X:REAL):BOOLEAN returns TRUE if X is a NaN, FALSE otherwise.

NEXTAFTER (X, Y:REAL):REAL returns, in general, the nearest representable number to X, in the direction of Y. Since Y serves only to indicate direction, Y can be infinite if the closure mode is affine (the default). If X is a NaN or infinite, or Y is infinite in projective mode, then NEXTAFTER (X,Y) returns X.

UNORDERED (X, Y:REAL):BOOLEAN returns TRUE if X and Y are unordered with respect to each other, FALSE otherwise.

INFINITY:REAL returns the positive infinity value.

MAXREAL:REAL returns the largest representable positive real value, which is about 3.4Ø2823466E38 in Pascal notation.

MINNORM:REAL returns the smallest positive normalized value, which is about 1.175494351E-38 in Pascal notation.

MINREAL:REAL returns the smallest representable positive real value (denormalized), which is about 1.4Ø1298464E-45 in Pascal notation.

MAKENAN (SIGNALING:BOOLEAN):REAL returns a "programmer's NaN" value, which contains the error code ØE in bits 8-15. If its parameter is true, then using the NaN in a subsequent arithmetic operation signals an invalid operation exception. If MAKENAN's parameter is false, the returned NaN will be a propagating NaN.

COPYSIGN (X, Y:REAL):REAL returns a number whose absolute value is that of X, but whose sign is that of Y.

LOGB (X:REAL):REAL returns, in general, the "binary order of magnitude" of X (i.e., the power of 2 represented in the exponent field of X). If X is Ø, (-infinity) is returned. If X is infinite, (+infinity) is returned. If X is a NaN, the value of X is returned.

SCALB (X:REAL; N:INTEGER):REAL scales X by a power of 2. If X is a numeric value, the value returned is

$$X * 2^{N}$$

This high-speed function allows you to scale a real value by a power of 2 which may be greater than the maximum representable real value. No rounding errors occur since SCALB affects only the exponent field unless the result would be denormalized. LOGB and SCALB are inverses in the sense that

1 < SCALB (X, -LOGB (X)) < 2.

SCALB may cause underflow or overflow. If overflow occurs, the value returned is infinity with X's sign. If X is infinite or a propagating NaN, the value of X is returned. If X is a signaling NaN, the invalid operation exception is signaled, and the value returned is the same NaN converted to a propagating NaN.

REM (X, Y), where X and Y are REAL values, returns a REAL value which is the remainder when X is divided by Y. This is computed, in principle, by normalizing X and taking the integer quotient Q, where Q is the integral value nearest to the mathematical REAL value X/Y. If X/Y is exactly halfway between two integral values, then Q is the even one. The sign of Q is determined by normal arithmetic rules. The value returned is X-(Q*Y).

If X is Ø and Y is nonzero, or X is finite and Y is infinite, then the value returned is X.

If X is infinite or Y is Ø, the invalid operation exception is signaled, and the value returned is a NaN.

The following figure summarizes the capabilities of the Apple III Pascal floating-point system:

Summary of the Floating-Point System

The Apple III Pascal floating-point system supports the Proposed Standard for Binary Floating-Point Arithmetic (Draft 8.Ø of IEEE Task P754).

Data Format

Real data are stored in 4 bytes (32 bits). The format is that specified in the standard, with least significant byte at lowest address.

Operations

Operations governed by the standard are supplied by the P-code interpreter and by three library units:

- interpreter: +, -, *, /, TRUNC, ROUND, and comparison
- PASCALIO: decimal to binary and binary to decimal conversions
- REALMODES: exception signals, halt & arithmetic mode switches, and miscellaneous functions
- TRANSCEND: remainder and square root

Floating-Point Exceptions

Arithmetic Exception	Default Response	Response if Continue
Overflow	Halt	+/- Infinity
Underflow	Continue	Denormalized or Ø
Divison by Zero	Halt	+/- Infinity
Conversion Overflow	Halt	(see standard)
Inexact Result	Continue	Round
Invalid Operation	Halt	NaN

Rounding Modes

Round-to-nearest (default)
Round-toward-Ø
Round up
Round down

Infinity Arithmetic

Affine Mode (default):	- Infinity < Finite Numbers < + Infinity
Projective Mode:	- Infinity = + Infinity

Denormalized Arithmetic

Normalizing Mode (default)
Warning Mode

Procedures and Functions That Check or Modify Settings

Procedures/Functions	Type of Signal, Switch, or Mode
SETXCPN(E:XCPN; B:BOOLEAN) GETXCPN(E:XCPN): BOOLEAN	Exceptions
SETHALT(H:XCPN; B:BOOLEAN) GETHALT(H:XCPN): BOOLEAN	Halts
SETCLOS(C:CLOSURE) GETCLOS: CLOSURE	Infinity closure
SETDNORM(D:DENORM) GETDNORM: DENORM	Underflow
SETROUND(R:RMODE) GETROUND: RMODE	Rounding
SAVENV(VAR V:NUMENV) RESTENV(V:NUMENV)	Whole environment

Special Functions and Predicates

COPYSIGN(x,y:REAL): REAL	FINITE(x:REAL): BOOLEAN
LOGB(x:REAL): REAL	ISNAN(x:REAL): BOOLEAN
SCALB(x:REAL; n:INTEGER): REAL	UNORDERED(x,y:REAL): BOOLEAN
NEXTAFTER(x,y:REAL): REAL	INTEGRAL(x:REAL): BOOLEAN
INFINITY: REAL	MINNORM: REAL
MAXREAL: REAL	MINREAL: REAL
MAKENAN(halting:BOOLEAN): REAL	

Bibliography

The following articles contain detailed information and discussion of the proposed IEEE floating-point standard. (Articles are listed in order of importance.)

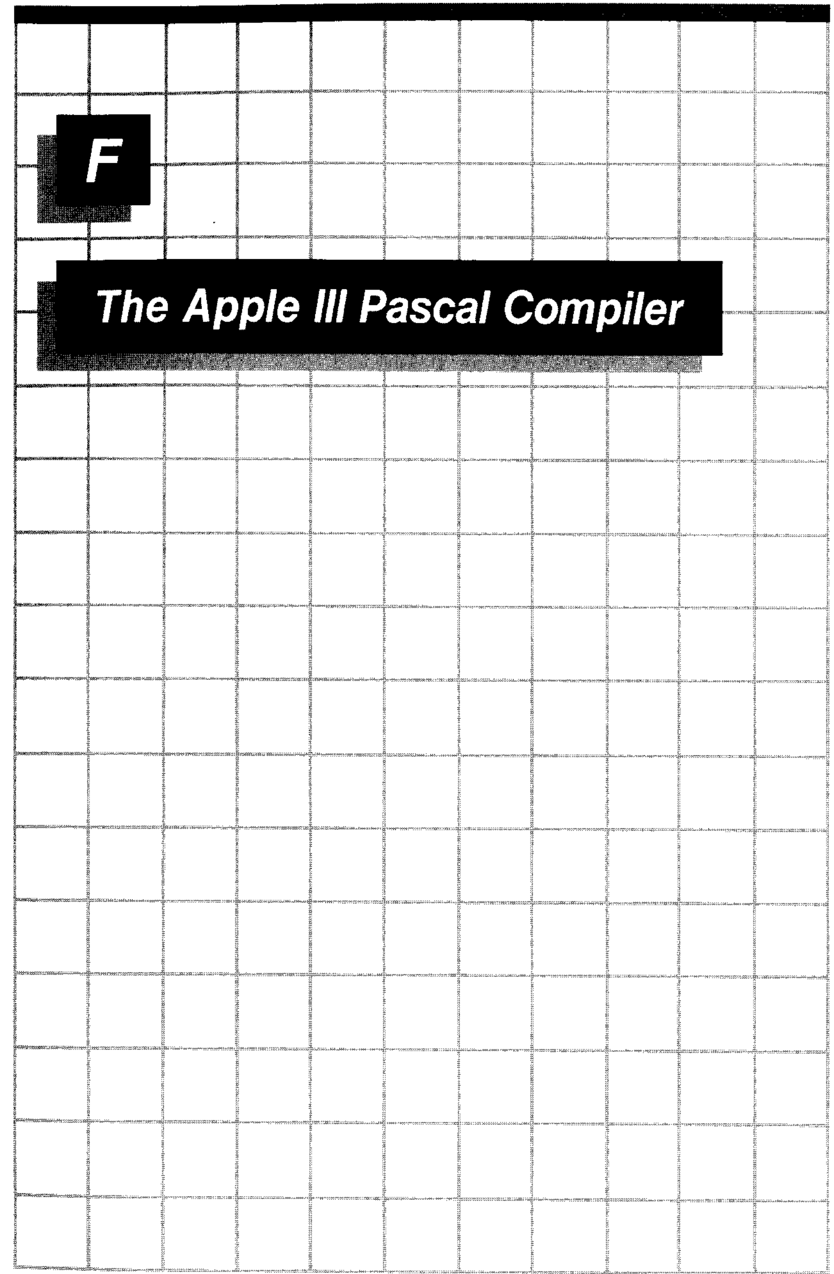
"A Proposed Standard for Binary Floating-Point Arithmetic", IEEE Computer, Vol. 14, No. 3, March 1981.

Coonen, J.: "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic", IEEE Computer, Vol. 13, No. 1, January 198Ø.

ACM SIGNUM Newsletter, special issue devoted to the proposed IEEE floating-point standard, October 1979. In particular, see article by Kahan and Palmer.

Coonen, J.: "Underflow and the Denormalized Numbers", IEEE Computer, Vol. 14, No. 3, March 1981.

Coonen, J.: "Binary <--> Decimal Conversion in KCS Arithmetic", (unpublished ms.), July 1Ø, 198Ø.



F

The Apple III Pascal Compiler

Introduction

The Apple III Pascal Compiler translates the source textfile of a Pascal program into a codefile. The codefile contains P-code, which is the "machine language" of the Pascal interpreter or "pseudo-machine."

For a simple program, the codefile can be executed immediately. However, if the program contains any external references, the Linker must be used to link external code into the codefile before it can be executed, as described in the Program Preparation Tools manual. This is required in the following cases:

- If the program contains any procedures or functions that are declared EXTERNAL (ie., assembly code), it requires linking.

- If the program uses any regular units, it requires linking. Note that the library packages supplied with the system are intrinsic units and do not require linking.

Diskette Files Needed

To operate the Pascal Compiler, you need the following diskette files:

- Your source file — Any diskette, any drive; default is the main system diskette's text workfile SYSTEM.WRK.TEXT , any drive.

- SYSTEM.COMPILER — Any diskette, any drive.

- SYSTEM.LIBRARY — System diskette, any drive; required only if any of the units in the system library are used by the program.

- Other Libraries — Any diskette, any drive; required if any units not in the system library are used by the program being compiled. In this case you will need to use the Compiler's USING option, described further on in this appendix.

- SYSTEM.EDITOR — Any diskette, any drive; optional; to fix errors found by the Compiler.

- SYSTEM.SYNTAX — Any diskette, any drive; optional; contains error messages given on entering the Editor with an error from the Compiler.

In addition to the above files, the following files may be needed if you are invoking the Compiler automatically via the Run command:

SYSTEM.LINKER
SYSTEM.PASCAL

Using The Compiler

The Compiler is invoked by typing C for Compile or R for Run from the outermost Command level of the system. The difference between these commands is that Compile simply compiles the source file, while Run has three stages: First it compiles the source file if no codefile is found; then it automatically runs the Linker if the program has any external references; and finally it automatically executes the program.

When the Linker is run automatically under the Run command, it will only link in external code from the SYSTEM.LIBRARY file. If your program uses any external code that is in a different library file, you must use the Compile command and then explicitly run the Linker via the Link command. To compile, the program must also contain the USING option, described further on in this appendix.

If you use Compile instead of Run, it is up to you to run the Linker if necessary and to execute the program by means of the Execute command.

By default, the Compiler takes the text workfile as its input, and places its output in the code workfile. The Compiler always does this if the text workfile exists—even if it doesn't contain valid Pascal source text, in which case the Compiler will soon detect an error and terminate. If you don't want to compile the workfile, use the Save and New commands of the Filer to save and clear the workfile.

If the text workfile exists, the screen immediately shows the message

```
Compiling...
```

If no text workfile exists, you are prompted for a source filename:

```
Compile what text?
```

You should respond by typing the name of the text file that you wish to have compiled. If you do not type a suffix, the suffix .TEXT is automatically supplied by the Compiler. If you want to prevent this from happening, add a period to the end of your filename. (If you want to return to the main menu without compiling, press RETURN or press ESC followed by RETURN.)

Next, if there is no text workfile, you will be asked for the name of the file where you wish to save the compiled version of your program:

```
To what codefile?
```

If you press ESC followed by RETURN, the command will be terminated. However, if you simply press the RETURN key, the command will not be terminated, as you might expect. Instead, the source file will be compiled and the compiled version of your program will be saved on the code workfile named SYSTEM.WRK.CODE on the system diskette.

If you want the codefile to have the same name as the source textfile (with the suffix .CODE instead of .TEXT), just type a dollar sign and press the RETURN key. The dollar sign ($) repeats your entire source file specification, including the volume identifier, so do not specify the volume identifier before typing the dollar sign.

If you want your codefile to have a different name, type the desired pathname. If you do not type the suffix .CODE, that suffix is automatically supplied by the Compiler.

If you are using Apple II compatible disks, the following information may be needed. By default, the Compiler places the code file at the beginning of the largest unused space on the disk. To override this, you can give a size attribute with the pathname. In this case you must type the suffix .CODE, followed by the number of blocks in square brackets, followed by a period:

```
To what codefile? myprog.code[8].
```

The period at the end prevents the system from adding the .CODE prefix after the size attribute. The size attribute [8] causes the code file to be placed in the first location on the disk where at least 8 blocks are available.

While the Compiler is running, messages on the screen show the progress of the compilation as in the following example:

```
Apple /// Pascal Compiler [A3/1.Ø]
<   Ø>......
MYPROG  [  2334 WORDS]
<   6>........
14 Lines
Smallest available space = 2334 words
```

The numbers in square brackets in the first line identify a particular version of the Compiler, and may not be as shown here.

The identifiers appearing on the screen are the identifiers of the program and its procedures. The identifier for a procedure is displayed at the moment when compilation of the procedure body is started.

The numbers within [ ] indicate the number of 16-bit words available for symbol table storage and Compiler execution at that point in the compilation. If this number falls too low, the Compiler may fail with a "stack overflow" message. You must then put the swapping option (described below) into your program and try again.

The numbers enclosed within < > are the current line numbers in the source file. Each dot on the screen represents one source line compiled.

If the Compiler detects an error in your program, the screen will show the text preceding the error, an error number, and a marker <<<< pointing to the symbol in the source where the error was detected. The following is an example:

```
IF I=2 THEN I:=Ø;
ELSE <<<<
Line 9, error 6:  <sp>(continue), <esc>(terminate), E(dit
```

This shows that the word ELSE is an illegal symbol at this point in the program. You have three options when you see a message like this.

- Pressing the spacebar instructs the Compiler to continue the compilation. If the error is not fatal (ie., if the error number is less than 4ØØ), the Compiler will attempt to recover and continue compilation without generating a codefile. Note that further error messages may appear as a consequence of the first error.

- Pressing the ESC key causes termination of the compilation and return to the Command level.

- Typing E sends you to the Editor, which automatically reads in the workfile, ready for editing. If you were not compiling the workfile, the Editor requests the name of the file you were compiling. You should respond by typing the filename of the file you were compiling, and that file will then be read into the Editor. When the correct file has been read into the Editor, the top line of the screen displays the error message (or number, if SYSTEM.SYNTAX is not on line) and the cursor is placed at the symbol where the error was detected.

  If SYSTEM.SYNTAX is not available, you can look up the error in Table 5 (see Appendix J). You may wish to delete the file SYSTEM.SYNTAX on your backup copies to obtain more diskette space.

Compiler Option Syntax

Compiler options are placed in the text to be compiled; the option takes effect when the Compiler arrives at that place in the text during compilation. A Compiler option looks like a special kind of comment and takes the following form using either the { } or (* *) comment delimiters:

```
{$option}

  or

(*$option*)
```

where "option" consists of a keyword followed by any arguments that a particular option may require. (In this appendix, use of { } is assumed.) Typical arguments are "+" (meaning "ON"), "-" (meaning "OFF"), or a pathname. For most options, the keyword can be given in full or abbreviated to a single letter; see the individual option descriptions below. Also, all keywords can be either upper or lower-case.

As shown below, there must be no spaces on either side of the $ character:

```
{$GOTO-}           This is a compiler option.
{ $GOTO-}          This is an ordinary comment.
{$ GOTO-}          This is an ordinary comment.
```

Several options can be combined in one set of {$...} brackets, by separating the options with commas; spaces are not allowed after the comma.

```
{$option,option,...}    Example:  {$IOCHECK-,SWAP+,GOTO-}
```

Some options can appear anywhere in a source file; others must appear before the program heading; and still others must appear at specific points within the source file. See the individual descriptions below.

Some options require a text string (pathname, identifier, number, etc.) immediately following the option letter, instead of the usual + or -. In this case, all characters between the option

letter and the next comma or } are taken as the string, except that blanks preceding or following the string are ignored. Therefore, the string must be the last item before the comma or }.

If the first character of a string is + or -, you must place a blank between the option letter and the string.

& HAND Within an option that does not take a text string, you can embed text after the argument and before the comma or } that ends the option. This text will be ignored; for example, the two options

{$GOTO+}

{$GOTO+ GOTO statements allowed from here on}

are exactly equivalent; the explanatory text in the second one is ignored.

Options That Do Not Affect Program Code

The SWAP, LIST, PAGE, COMMENT, and QUIET options have no effect on the code, run-time loading, or execution of the program. They are provided as conveniences. The most important of these are SWAP and LIST.

The SWAP Option

This option determines whether or not the Compiler operates in "swapping" mode.

There are two main parts of the Compiler: one processes declarations; the other handles statements. In the swapping (S+ OR SWAP+) mode, only one of these parts is in main memory at a time. This makes about 53ØØ additional words available for symbol-table storage at the cost of slower compilation speed (because of the overhead of swapping the Compiler segments in from disk). This option must occur before the program heading, or it will have no effect.

Default option: {$SWAP-}

{$SWAP+} Puts the Compiler in swapping mode.

{$SWAP-} Puts the Compiler in non-swapping mode.

{$SWAP++} The Compiler does even more swapping than with the S+ option. The program compiles still more slowly, but still more room is left in memory for symbol-table storage (about 15ØØ more words).

The LIST Option

This option consists of the keyword LIST or the letter L followed by a +, -, or pathname argument. A program listing is a text file that contains the source text plus annotations indicating how the resulting code is related to the source text. This is useful for debugging purposes. LIST controls whether the Compiler will generate a program listing, which parts of the program will be listed, and where the listing will be written.

The LIST option is most often placed before the program heading to generate a complete listing, but it can be placed anywhere in the source text. Only one listing file can be produced.

Default option: {$LIST-}

{$LIST pathname} Tells the Compiler to start listing to the specified file.

{$LIST+} Tells the Compiler to turn on listing of the following source text. If a pathname has not been specified with {$LIST pathname}, then the listing goes to the file SYSTEM.LST.TEXT on the system diskette.

{$LIST-} Tells the Compiler to temporarily halt listing.

For example, the following will cause the compiled listing to be sent to a diskfile called DEMO1.TEXT on the diskette named /MYDISK

{$LIST /MYDISK/DEMO1.TEXT }.

Note that a diskette listing file may be edited just like any other text file, provided that it is not too big and that page options have not been used. A listing file is approximately twice as large as a source file.

In the compiled listing, the Compiler places next to each source line the line number, segment number, procedure number, and the number of bytes or words (bytes for code, words for data) required by that procedure's declarations or code to that point. The Compiler also indicates whether the line lies within the actual code to be executed or is a part of the declarations for that procedure by printing a "D" for declaration, an integer Ø..9 to designate the lexical level (the level of statement nesting within the code part), or an "S" to indicate skipping of text due to conditional compilation. All of these indications are as of the end of the line.

Here is a sample listing, to which column headings have been added:

```
                 Proc#:
Line#   Seg#   Lex.lvl   Byte#  Program Text

    1    1      1:D        1 {$LIST /PRESCR/DOCTORLIST.TEXT}
    2    1      1:D        1 PROGRAM DOCTOR;
    3    1      1:D        3  VAR WEEK: 1..52;
    4    1      1:D        4
    5    1      2:D        1    PROCEDURE DOSE;
    6    1      2:Ø        Ø    BEGIN
    7    1      2:1        Ø      WRITE('1 APPLE/DAY');
    8    1      2:1       23      WRITELN(' AND ')
    9    1      2:Ø       48    END;
   1Ø    1      2:Ø       6Ø
   11    1      3:D        1    PROCEDURE WEEKTREAT;
   12    1      3:D        1     VAR DAY: 1..7;
   13    1      3:Ø        Ø    BEGIN
   14    1      3:1        Ø      FOR DAY := 1 TO 7 DO BEGIN
   15    1      3:3       17         DOSE
   16    1      3:2       17      END
   17    1      3:Ø       19    END;
   18    1      3:Ø       4Ø
```



```
   19    1      1:Ø        Ø BEGIN
   2Ø    1      1:Ø        Ø    {intentional value range error follows}
   21    1      1:1        Ø    FOR WEEK := Ø TO 52 DO BEGIN
   22    1      1:3       19       WEEKTREAT
   23    1      1:2       19    END;
   24    1      1:1       28    WRITELN('THAT KEEPS THE DOCTOR AWAY')
   25    1      1:Ø       74 END.
```

The information given in the compiled listing can be very valuable for debugging a large program. A run-time error message will normally indicate the segment number, the procedure number, and the byte number within the current procedure where the error occurred.

Here is a sample run-time error message:

```
Exec Err # 1
S# 1, P# 1, I# 5
Type <space> to continue
```

where S# is the segment number, P# is the procedure number, and I# is the byte number in that procedure where the error occurred. In this example, you could find the Pascal statement where the error occurred by finding Segment 1 in the second column of the listing, then Procedure 1 in the third column. Then look in the fourth column for the largest byte number that is less than 5. This is the starting byte number of the statement that contains Byte 5 of Procedure 1 of Segment 1, and this is the statement where the error occurred.

The PAGE Option

This option consists of the keyword PAGE or the letter P, with no arguments. If a listing is being produced, the PAGE option causes one form-feed character (ASCII 12) to be inserted into the text of the listing, just before the line containing the PAGE option. If your program contains the line

```
{$PAGE}
```

that line will appear at the top of a new page when you print the program's compiled listing. Before editing a listing file containing form feed characters, use the Replace command of the Editor to remove all form feeds.

The COMMENT Option

This option consists of the keyword COMMENT or the letter C and a line of text. The text is placed, character for character, in Block Ø of the codefile bytes 432 to 511 (where it will not affect program operation). The purpose of this is to allow a copyright notice or other comment to be embedded in the codefile. Example:

```
{$COMMENT COPYRIGHT ALLUVIAL O. FANSOME 1981}
```

The COMMENT option can appear anywhere in the program. Note that the line of text cannot contain a comma.

The QUIET Option

This option consists of the keyword QUIET or the letter Q followed by a + or - argument. It can be used to suppress the screen messages that tell the procedure names and line numbers and detail the progress of the compilation.

Default option: {$QUIET-}

{$QUIET+} Causes the Compiler to suppress output to the screen.

{$QUIET-} Causes the Compiler to send procedure name and line number messages to the screen.

Error Checking Options

The IOCHECK, RANGECHECK, VARSTRING, and GOTO options control four different error-checking features. IOCHECK and RANGECHECK are options for run-time error checking; VARSTRING and GOTO are options for compile-time error checking.

Note that the Compiler provides for other types of error checking besides the types controlled by these options.

The IOCHECK Option

This option consists of the keyword IOCHECK or the letter I and a + or - argument. It tells the Compiler whether or not to

create automatic error-checking code after each structured file I/O statement (not the block or device I/O statements). If the automatic error-checking detects an I/O error, it halts the program with a run-time error message.

Default option: {$IOCHECK+}

{$IOCHECK+} Instructs the Compiler to generate code after each statement that performs any I/O, in order to check that the I/O operation was accomplished successfully. In the case of an unsuccessful I/O operation, the program will be terminated with a run-time error.

{$IOCHECK-} Instructs the Compiler not to generate any I/O-checking code. In the case of an unsuccessful I/O operation, the program is not terminated with a run-time error. The program can then use the IORESULT function to detect and report I/O errors. (See Chapter 1Ø.)

The IOCHECK option can appear anywhere in the program.

The RANGECHECK Option

This option consists of the keyword RANGECHECK or the letter R, followed by a + or - argument. With the {$RANGECHECK+} option, the Compiler will produce code that checks on array and string subscripts and on assignments to variables of subrange and string types. The checking code will halt the program with a run-time error message if a subscript or assignment is out of the range specified in the program's declarations.

Default option: {$RANGECHECK+}

{$RANGECHECK+} Turns range checking on.

{$RANGECHECK-} Turns range checking off.

The RANGECHECK option can appear anywhere in the program. Note that programs compiled with the {$RANGECHECK-} option selected will run slightly faster. However if an invalid index occurs or an invalid assignment is made, the program will not be halted. Use {$RANGECHECK-} only when speed or code size is critical.

The VARSTRING Option

This option consists of the keyword VARSTRING or the letter V followed by a + or - argument. When a procedure or function has a VAR parameter of type STRING, the actual parameter in each call to the procedure or function can be checked at compile time to make sure that its declared maximum length is not less than the declared maximum length of the formal parameter. This checking is controlled by the VARSTRING option:

Default option: {$VARSTRING+}

{$VARSTRING+} Turns checking on.

{$VARSTRING-} Turns checking off.

Note that if checking is off and the actual length of the actual parameter is less than the maximum length of the formal parameter, it is possible for the procedure or function to alter bytes of data that are beyond the end of the actual parameter variable. If VARSTRING checking is off and RANGECHECKing is on, then the range checked is the length of the formal parameter, not the length of the actual parameter. This does not cause a run-time error, but does cause unpredictable results.

The GOTO Option

This option consists of the keyword GOTO or the letter G and a + or - argument. It tells the Compiler whether to allow or forbid the use of the Pascal GOTO statement within a program.

Default option: {$GOTO-}

{$GOTO+} Allows the use of the GOTO statement.

{$GOTO-} Causes the Compiler to treat a GOTO as an error.

The GOTO option can appear anywhere in the program.

Control of Segments and Libraries

The NEXTSEG, NOLOAD, and RESIDENT options control the way that segments of a program are loaded for execution. Full

explanations and examples of the use of these options are in Chapter 15. The USING option is used to select a library file other than SYSTEM.LIBRARY as the file to search for units referred to in a USES declaration.

The NEXTSEG Option

The NEXTSEG option consists of the keyword NEXTSEG or the letters NS followed by an unsigned integer which should be in the range 1..63. This option specifies the segment number to be associated with the next code segment produced by the Compiler. This option can appear anywhere in the program but is ignored in certain cases; see Chapter 15 for details.

The NOLOAD Option

This option consists of the keyword NOLOAD or the letter N followed by a + or - argument. It prevents the code of any units used by the program from being loaded automatically when the program is executed. Instead, each unit's code is in memory only when some portion of it is active, or unless specified as resident by the RESIDENT option.

Default option: {$NOLOAD-}

{$NOLOAD+} Unit code will be loaded only when active.

{$NOLOAD-} Unit code will be loaded as soon as program begins executing.

The {$NOLOAD+} option should be placed at the beginning of the main program body (after the BEGIN). Note that use of the {$NOLOAD+} option does not prevent the initialization portion of a unit from being initially executed. For more information see Chapter 15.

The RESIDENT Option

This option consists of the keyword RESIDENT or the letter R followed by either an identifier or an unsigned number.

If an identifier is used, it must be the identifier of a unit or of a SEGMENT procedure or function. If a number is used, it should be the segment number of a unit or of a SEGMENT procedure or function.

The RESIDENT option should be placed at the beginning of a procedure or function body (after the BEGIN and before any statements). It causes the code of the specified segment to be kept in memory, for as long as the procedure or function that contains the option is executing. See Chapter 15 for details.

The USING Option

This option consists of the keyword USING or the letter U followed by the pathname of a library file. The USING option causes the Compiler to seek units in subsequent USES declarations in the named library file instead of in SYSTEM.LIBRARY.

Note that the USING option applies only during compilation. If intrinsic units are used, then at execution time the system will still look for them first in the program library (if there is one) and then in SYSTEM.LIBRARY .

The specified pathname is used exactly as typed. No suffix is added.

The following is an example of a valid USES declaration employing the USING option:

```
USES UNIT1,UNIT2, {Found in SYSTEM.LIBRARY}
{$USING MYDISK:A.CODE } UNIT3,
{$USING MYDISK:B.LIBRARY } UNIT4,UNIT5;
```

The INCLUDE Option

This option consists of the keyword INCLUDE or the letter I followed by a pathname. It causes the contents of another file of Pascal source text to be compiled at that point in processing the source file. Thus you can compile a large program without having the entire source in one large file. The syntax is

```
  {$I pathname }
or {$INCLUDE pathname}
```

If the INCLUDE option is in a series of options separated by commas (within a single pair of comment delimiters), then it must be the last option in the series.

Apart from this one restriction, the INCLUDE option can appear anywhere in the source file. The contents of the specified file are inserted into the compilation at the point where the option is encountered by the Compiler.

The Compiler expands the pathname according to the normal rules when expecting a textfile. If the attempt to open the file fails, or if some I/O error occurs while reading the file, the Compiler responds with a fatal error message and terminates its operation.

If the INCLUDE option occurs within the declarations section of a program or procedure (i.e., before the BEGIN), then the Compiler will allow further declarations out of order. For example, suppose that a program contains TYPE declarations and VAR declarations, and then an INCLUDE option. The included file is allowed to contain further TYPE and VAR declarations, and can also contain USES, LABEL, and CONSTANT declarations.

If the INCLUDE option occurs within the body of a procedure or program (i.e., after the BEGIN), the included file must not start with any declarations. If it does, a syntax error is generated because declarations are not allowed in a program or procedure body.

The Compiler cannot keep track of nested INCLUDE options; i.e., an included file must not contain an INCLUDE option. This results in a fatal Compiler error.

Special Compilation Mode

The USER option controls a special compilation mode. It is not used in normal programming with the standard Apple III Pascal system.

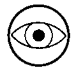
The USER Option

This option consists of the keyword USER or the letter U followed by a + or - argument. It determines whether this compilation is a user program compilation, or a compilation at the system level.

Default option: {$USER+}

{$USER+} Informs the Compiler that this compilation is to take place on the user program lexical level.

{$USER-} Tells the Compiler to compile the program at the system lexical level. Also sets certain other options as follows: RANGECHECK-, GOTO+, IOCHECK-.



Compilation at the system level will produce meaningful results only if the program was written with knowledge of the operating system code structure. Do not attempt system-level compilation unless you have this knowledge.

Conditional Compilation

The conditional compilation capability of the Apple III Pascal Compiler allows sections of the source text to be skipped. The skipping is controlled by the IFC, ELSEC, and ENDC options, which are used to bracket sections of source text. A fourth option, SETC, is used to create "compile-time variables" and assign values to them.

Compile-Time Variables

IFC, like the Pascal IF statement, makes a decision based on a boolean value which it obtains by evaluating an expression. The expression can contain compile-time variables. These variables are completely independent of program variables; even if a compile-time variable and a program variable have the same identifier and appear in the same procedure, they can never be confused by the Compiler.

A compile-time variable is "declared" when it appears for the first time on the left-hand side of a SETC assignment; for example, the option

```
{$SETC LIBVERSION := 5}
```

declares the compile-time variable LIBVERSION (if it has not appeared previously) and assigns the value 5 to it. Since 5 is an INTEGER value, LIBVERSION is a variable of type INTEGER (the SETC option is explained in detail below). Now suppose that later in the compilation the Compiler finds

```
...
{$IFC PROGVERSION >= LIBVERSION}
K := KVAL1(DATA+INDAT);
{$ELSEC}
K := KVAL2(DATA+CPINDAT^);
{$ENDC}
WRITELN(K);
...
```

where PROGVERSION is another compile-time variable. If the value of PROGVERSION is greater than or equal to 5 (the value of LIBVERSION), then the statement K := KVAL1(DATA+INDAT) is compiled, and the statement K := KVAL2(DATA+CPINDAT^) is skipped.

But if the value of PROGVERSION is less than the value of LIBVERSION, then the first statement is skipped, and the second statement is compiled.

In either case, the WRITELN(K) statement is compiled because the conditional construction ends with the {$ENDC} option.

Note the following points about compile-time variables:

- The constants TRUE and FALSE are pre-declared.

- A compile-time variable is declared when it appears for the first time on the left-hand side of the assignment in a SETC option.

- All compile-time variables must be declared before the end of the declarations section of the main program. In other words, a SETC option that declares a new compile-time variable must precede the main program's

procedure and function definitions (if any), and must precede the BEGIN of the main program body; otherwise a compilation error will be generated.

- At any time, a compile-time variable can have a new value assigned to it by a SETC option.
- The type of a compile-time variable is that of the most recent value assigned to it in a SETC option. Only built-in scalar types are allowed. Therefore the only possible types are INTEGER, BOOLEAN, and CHAR.
- There is no scope for compile-time variables; once declared, a compile-time variable is known throughout the compilation. As already mentioned, compile-time variable identifiers are completely independent of identifiers used by the program.

One compile-time variable, APPLE, is pre-declared; it is used to specify whether Apple III code or Apple II code is to be produced by the Compiler. Details are given further on in this appendix; do not use the identifier APPLE as a compile-time variable identifier for any other purpose.

Compile-Time Expressions

Compile-time expressions appear in the SETC option (on the right-hand side of an assignment) and in the IFC option. The only operands allowed in a compile-time expression are compile-time variables and constants of the types INTEGER, BOOLEAN, and CHAR. Function calls, set constructors, pointer references, or references to program variables are not allowed.

The IN operator is not allowed, but all of the other operators that can be used in Pascal expressions are allowed; since there are no compile-time REAL values, the / operator is automatically replaced by DIV. The Compiler evaluates a compile-time expression as soon as it is encountered in the text, according to the usual rules for evaluating Pascal expressions.

The SETC Option

The keyword SETC cannot be abbreviated. The SETC option has the form

```
{$SETC ID := EXPR}
```

where ID is the identifier of a compile-time variable and EXPR is a compile-time expression. EXPR is evaluated immediately. If ID has not yet been found in a SETC option, it is declared at this time. In any case, the value and type of EXPR are assigned to ID.

THE IFC, ELSEC, AND ENDC OPTIONS (@)

The keywords IFC, ELSEC, and ENDC cannot be abbreviated. The ELSEC and ENDC options take no arguments. The IFC option has the form

```
{$IFC EXPR}
```

where EXPR is a compile-time expression with a boolean value.

These three options form constructions similar to the Pascal IF statement, except that the ENDC option is always needed at the end of the IFC construction. In other words, there are two ways of using IFC. The first is without an ELSEC option:

```
...
{$IFC compile-time expression}
SOURCE TEXT A
{$ENDC}
SOURCE TEXT B
...
```

If the compile-time expression has the value TRUE, then both SOURCE TEXT A and SOURCE TEXT B are compiled as if the options were not there. If the compile-time expression has the value FALSE, then SOURCE TEXT A is skipped and compilation continues with SOURCE TEXT B.

The second form uses ELSEC:

```
...
{$IFC compile-time expression}
SOURCE TEXT A
{$ELSEC}
SOURCE TEXT B
{$ENDC}
SOURCE TEXT C
...
```

If the compile-time expression has the value TRUE, then SOURCE TEXT A is compiled and SOURCE TEXT B is skipped. If the compile-time expression has the value FALSE, then SOURCE TEXT A is skipped and SOURCE TEXT B is compiled. In either case, compilation continues with SOURCE TEXT C.

IFC constructions can be nested within each other to 5 levels. Every IFC must have a matching ENDC.

When the Compiler is skipping source text during a conditional compilation, all options are ignored except the following:

ELSEC
ENDC
IFC (so that ENDC's can be matched properly)
SETC
INCLUDE (text is scanned even if it is being skipped, in case it contains ELSEC, ENDC, IFC, or SETC options).

All Pascal program text is ignored during skipping. If a listing is produced, each source line that is skipped is marked with the letter S as its "lex level."

Compiling Apple II Code

The compile-time variable APPLE is predeclared with a value of 3. This default causes the Compiler to produce Apple III code. If the value of APPLE is changed to 2 by a SETC option before the program header,

```
{$SETC APPLE := 2}
```

then the Compiler will produce an Apple II codefile.

Compiler Option Summary

Note: in the following summary, "pn" stands for "pathname."

COMMENT	(C)	Following string is placed directly into codefile.
GOTO+	(G+)	Allows GOTO statements.
GOTO-	(G-)	Forbids GOTO statements (default).
IOCHECK+	(I+)	Generates I/O-checking code (default).
IOCHECK-	(I-)	No I/O checking.
INCLUDE pn	(I pn)	Includes named source file in compilation.
LIST+	(L+)	Sends listing to SYSTEM.LST.TEXT on system disk.
LIST-	(L-)	Makes no compiled listing (default).
LIST pn	(L pn)	Sends compiled listing to named file.
NOLOAD+	(N+)	Prevents units from being loaded until activated.
NOLOAD-	(N-)	Loads units immediately when program runs (default).
NEXTSEG num	(NS num)	Specifies number of next segment.
PAGE	(P)	Inserts a form feed into listing.
QUIET+	(Q+)	Suppresses screen messages.
QUIET-	(Q-)	Sends messages to screen (default)
RANGECHECK+	(R+)	Generates range-checking code (default).
RANGECHECK-	(R-)	No range checking.
RESIDENT name	(R name)	Keeps named segment loaded while current procedure is active.
RESIDENT num	(R num)	Keeps segment number loaded while current procedure is active.
SWAP+	(S+)	Puts Compiler in swapping mode.
SWAP++	(S++)	Compiler does even more swapping.
SWAP-	(S-)	Non-swapping mode.
USER+	(U+)	Compiles user program (default).
USER-	(U-)	Compiles system program.
USING pn	(U pn)	Specifies name of library file for finding units.

VARSTRING+	(V+)	Checks VAR parameters of type STRING (default).
VARSTRING-	(V-)	No checking of VAR parameters of type STRING.
SETC	(SETC)	Assigns value to compile-time variable.
IFC	(IFC)	Conditional compilation.
ELSEC	(ELSEC)	Conditional compilation.
ENDC	(ENDC)	Conditional compilation.

G

Special Techniques

Introduction

This appendix discusses areas where Apple III Pascal goes beyond the fundamental ideas of Pascal as originally defined by Jensen and Wirth. These techniques are intended for the experienced programmer when faced with problems for which the concepts of Pascal are not adequate; if you use them, you abandon some of the safety features of the language and it will be easier to create incomprehensible bugs in your program. Thus use them carefully if at all.

Strong Typing

A major assumption of Pascal philosophy is strong typing; this means that any stored value represents data of one type only and cannot be directly interpreted as if it were of another type. Jensen and Wirth did provide a mechanism for circumventing strong typing, namely the free union record variant. This mechanism is explained here; in addition, Apple III Pascal provides other mechanisms to circumvent strong typing.

Boolean Logic

Another assumption of Pascal is that there are only two boolean values, represented by the built-in constants TRUE and FALSE. But in this implementation of Pascal, a boolean value can actually be any pattern of 16 bits and is interpreted as either "true" or "false" depending on its least significant bit. The boolean operators AND, OR, and NOT are usually assumed to do single operations on values of TRUE and FALSE; but actually they are bitwise logical operators with 16-bit operands. This permits some uses of boolean values and operations that are not normally considered part of Pascal.

Representation of Scalar Values

In order to understand and apply these special techniques, you need to know how some of the data types are represented. The following sections give details on how data are stored in memory, for each simple (i.e. single-valued) scalar data type. A later section deals with arrays.

The basic unit of storage is a 16-bit binary word, consisting of two contiguous 8-bit bytes. The least significant byte comes first (at the lower of the two byte addresses). A word can be visualized as follows:

high byte								low byte							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

most significant ... least significant

Every scalar value is stored in one word.

Integers

An integer value is stored in one word, least significant byte first. Two's-complement notation is used to represent negative integers.

Characters

A char value is represented by its ASCII code, stored in one word. Since ASCII codes are in the range 0..255, they only require one byte; the char code is stored in the least significant byte of the word (i.e. the first byte). The most significant byte contains 0's.

Booleans

A boolean value is stored in one word. The logical "true" or "false" value is represented by the least significant bit of the least significant byte; for most purposes, this is the only meaningful bit and all the other bits are ignored. However, see the section below on the ORD and ODD functions.

User-Defined Scalars

When a user-defined scalar type is declared, each of its value identifiers is associated with an ORD value: the first one declared has an ORD value of 0, the next has an ORD value of 1, and so forth. For example, the declaration

```
VAR DAY: (MON, TUES, WED, THURS, FRI, SAT, SUN);
```

creates a variable DAY whose possible values (at the source program level) are MON..SUN. These have the ORD values Ø..6; thus ORD(MON) is Ø, ORD(TUES) is 1, and ORD(SUN) is 6. Now if DAY is assigned a particular value, such as

```
DAY := WED
```

the value is represented in memory as the integer 2—because ORD(WED) is 2. Every user-defined scalar value is represented in one word in memory as its binary value—its ordinality.

Implications

By combining this information on representation of scalars with the following facts about the ORD and ODD functions, you can use some special techniques.

ORD and ODD

The familiar ORD function accepts any scalar value as its parameter, and returns the ordinality of that value. This is done in a strikingly simple way: ORD merely returns the very same value that was passed to it; since ORD is by definition an integer function, the returned value is now interpreted as an integer. This works because every scalar value is stored in the same way: as a binary value. The numerical value of this word is the ordinality of the scalar value.

The ODD function accepts any integer as its argument; it returns "true" if the integer is odd, and "false" if the integer is even. We saw above that "true" and "false" depend only on the least significant bit of a boolean value; now notice that "odd" and "even" depend only on the least significant bit of an integer value. What ODD actually does is to return the same value that was passed to it; since ODD is by definition a boolean function, the returned value is now interpreted as a boolean value.

This implies that any scalar value can be interpreted according to its original type, or as an integer, or as a boolean value:

- To interpret the value of any non-integer scalar S as an integer, use ORD(S).

- To interpret the value of any integer N as a boolean, use ODD(N).
- To interpret the value of any non-integer scalar X as a boolean, use ODD(ORD(X)).

Here is a simple example of the application of these ideas. It uses the concept of a bit mask as a way of controlling one bit within a char variable—namely Bit 5, which is the bit that distinguishes between capital and lower-case letters.

```
PROGRAM MASKER;
{A program to read a string and convert each lower-case
 letter to the corresponding upper-case letter by masking
 off Bit 5 }

VAR
    ST:STRING;      {A variable to contain the string }
    MASK:BOOLEAN;   {Contains a bit pattern with a Ø in
                     Bit 5 and all 1's in the other bits }
    LOWERCASE:SET OF CHAR;  {Contains the lower-case letters }
    I:INTEGER;      {An integer index variable }
```

```
BEGIN
  MASK:=NOT ODD(32);
  {The integer value 32 has a 1 in Bit 5 and all Ø's in the
   other bits. We take ODD(32) which is the same bits but
   considered as a boolean value; the NOT operation
   complements all 16 bits, resulting in a Ø in Bit 5 and
   all 1's in the other bits: }

  {Initialize LOWERCASE with the lower-case letters: }
  LOWERCASE:=['a'..'z'];
  {Prompt for a string, and read it into ST: }
  WRITE('Type a string: ');
  READLN(ST);
  {Scan the string: }
  FOR I:=1 TO LENGTH(ST) DO

    {If the character is lower-case letter, then... }
    IF ST[I] IN LOWERCASE THEN
      {AND it with MASK. The char code is a bit pattern.
       Take ORD of it, which returns the same bits as an
       integer, then take ODD which returns the same bits as
       a boolean. Now AND it with MASK, thus masking off
       Bit 5. To get the bits back to being a char value,
       take ORD and then CHR: }
      ST[I]:=CHR(ORD(MASK AND ODD(ORD(ST[I]))));

  {Write out the string: }
  WRITELN(ST)
END.
```

There are simpler ways to do case conversion on char values; the above program is presented merely to illustrate the technique of manipulating individual bits by using ODD, ORD, and the boolean operators.

"Normal" Uses of Booleans

As we have seen, there are other boolean values besides FALSE and TRUE; for example ODD(3) is a boolean "true" value but ORD(ODD(3)) is 3, not 1. However, note that the boolean constants FALSE and TRUE are always represented as the integer values Ø and 1, respectively, since ORD(FALSE) is Ø and ORD(TRUE) is 1.

It might appear that these extensions would interfere with the "normal" uses of booleans--that is, the uses described or implied

by Jensen and Wirth. However, in Apple III Pascal there is no problem. The "normal" uses of boolean values work normally because they consider only the least significant bit of a boolean value. Thus every boolean value appears to be equal to either TRUE or FALSE in all the following cases:

- Boolean value used to control an IF, WHILE, REPEAT, CASE, or FOR statement.

- Comparison of two boolean values. The comparison expression ODD(3)=ODD(5), for example, gives the result TRUE.

- Array index of type boolean.

- Testing a boolean value for membership in a set of boolean.

- Putting a boolean value into a set of boolean.

The only "normal" use of a boolean that gives an "abnormal" result is taking the ORD of a boolean, as described above: the result may be a value other than Ø or 1. However there is generally no reason to do this except when you want the "abnormal" result; if you do want the "normal" value, write ORD(B)=TRUE where B is the boolean value.

Representation of Arrays

A non-packed array of scalar values is represented simply as a sequence of words, with each word containing one scalar value as described previously.

When the array is packed, each value does not necessarily take up one word. The word is still the unit of storage, but each word can contain more than one value if it has enough bits. For example, consider the declaration

```
VAR OCTAL: ARRAY[Ø..63] OF Ø..7;
```

which creates an array OCTAL of 64 elements. Each element is an integer value in the range Ø..7, and requires three bits. Since a word contains 16 bits, 5 array elements can be packed into a word. The elements are right-justified in the word: that is,

the first element in each word is in bits Ø..2, the second is in bits 3..5, and so on to the fifth element in bits 12..14. Bit 15 is unused. The next element goes in bits Ø..2 of the next word.

The following specific cases are of particular interest:

- A char value requires 8 bits; in a packed array of char, each word of storage contains two char values; the first one is in Bits Ø..7 and the second in Bits 8..15.
- A value of the subrange type Ø..255 also requires 8 bits and can be thought of as a "byte" type value. Storage in a packed array of Ø..255 is the same as for packed char values.
- A boolean value requires only one bit; in a packed array of boolean, each word contains 16 values. The first value is in Bit Ø, and the last is in Bit 15.

The above only applies as long as the variables remain packed. Whenever a value is unpacked from a packed variable, it is expanded to occupy a full word.

Representation of Real Values

Complete details on the representation of real values are given in Appendix E. Briefly, each real value is represented in two words, or 32 bits. The most significant bit is the sign bit, the next 8 bits are the exponent field, and the 23 least significant bits are the fraction field. An example program in the next section shows how to access individual bits in a real value.

Free Union Variants

In an ordinary variant record (as discussed in Chapter 8) a tag field value is stored as part of the record, and is normally used by the program to determine how to interpret the variant data. This is useful when the data in each variant field of a particular record is of a specific type; the presence of the tag field is a safeguard against misinterpreting the variant data.

But here we are interested in ways of purposely interpreting the same data in more than one way. This is possible with an ordinary variant record: merely ignore the tag field. By using a free union variant, you can eliminate the tag field; this saves a little memory and also makes the maneuver more convenient.

A free union variant looks like an ordinary variant, except that the tag field identifier is omitted. A tag type is still required, and case labels are still required. For example:

```
VAR FOXY: RECORD CASE BOOLEAN OF
                   FALSE: (INT: INTEGER);
                   TRUE:  (BOOL: BOOLEAN)
                 END;
```

Now FOXY.INT refers to a value of type integer, and FOXY.BOOL refers to a value of type boolean. Both refer to the same actual word of data. The labels FALSE and TRUE, corresponding to the tag type BOOLEAN, are a matter of convenience; you could use any tag type that has enough possible values to use as case labels. In the example below, you will see a user-defined scalar type declared solely for use as a tag type for a free union that has three cases.

Earlier, we needed to clear Bit 5 of a boolean variable, and did so by first assigning the value NOT ODD(32) to the MASK and then ANDing the MASK with the variable. In the following program we will use a free union as a more powerful way of accessing individual bits of a boolean value—and more. This will be a three-way free union that allows the same word of data to be treated as an integer, as a boolean value, or as a packed array of 16 boolean values.

```
PROGRAM BINARY;
{This program takes an integer value from the keyboard and
 displays its value as a 16-bit binary number by treating it
 as a packed array of 16 one-bit boolean values.  Then it
 treats the value as one 16-bit boolean value, complements
 it, and again displays the result as a 16-bit binary
 number:  }

TYPE
     {A type to use as tag type in a 3-way free union: }
     THREEWAY=(A,B,C);
     {An index type for 16-element arrays: }
     BITINDEX=Ø..15;
     {An array type of 16 booleans, each represented as a bit: }
     BITARRAY=PACKED ARRAY[BITINDEX] OF BOOLEAN;
     {A free union record type, which can represent an integer,
      or a bit array, or a boolean; same 16 bits in all cases: }
     THREETYPES=RECORD CASE THREEWAY OF
                       A:  (INT:INTEGER);
                       B:  (BITS:BITARRAY);
                       C:  (BOOL:BOOLEAN)
                     END;

VAR
   VALUE:THREETYPES;    {A variable of the free union type }

{A procedure which takes a parameter of free union type,
 treats it as a bit array, and writes the 16 bits out as 1's
 and Ø's:  }
PROCEDURE BINOUT(NUM:THREETYPES);
  VAR K:BITINDEX;  {An index variable }
  BEGIN
    {Scan the 16 bits, most significant first: }
    FOR K:=15 DOWNTO Ø DO
      {If the bit is true, write a 1;
       if it's false, write a Ø: }
      CASE NUM.BITS[K] OF
        TRUE: WRITE('1');
        FALSE:WRITE('Ø')
      END
  END;
```



```
{Main program: }
BEGIN
  {Prompt the user for a decimal integer: }
  WRITE('Type Number: ');
  {Store it as an integer value: }
  READLN(VALUE.INT);
  {Write it as a binary integer: }
  BINOUT(VALUE);
  WRITELN;
  {Complement the value as a 16-bit boolean: }
  VALUE.BOOL:=NOT VALUE.BOOL;
  {Write it as a binary integer: }
  BINOUT(VALUE);
  WRITELN;WRITELN
END.
```

The next example uses a similar technique to access individual bits of a real value. It takes any real value from the keyboard and displays the bit values.

```
PROGRAM REALBITS;
{This program takes a real value from the keyboard and
 displays the value of its sign, exponent, and fraction
 fields by treating it as a packed array of 32 one-bit
 boolean values:  }

TYPE
     {A type to use as tag type in a 2-way free union: }
     TWOWAY=(A,B);
     {An index type for 32-element arrays: }
     BITINDEX=Ø..31;
     {An array type of 32 booleans,
      each one represented as a bit: }
     BITARRAY=PACKED ARRAY[BITINDEX] OF BOOLEAN;
     {A free union record type, which can represent a
      real value or a bit array; same 32 bits in all cases: }
     TWOTYPES=RECORD CASE TWOWAY OF
                       A:  (REALVAL:REAL);
                       B:  (BITS:BITARRAY)
                     END;

VAR
    VALUE:TWOTYPES;    {A variable of the free union type }
```

```
{A procedure which takes a parameter of free union type,
 treats it as a bit array, and writes the 32 bits out as 1's
 and Ø's with spaces to separate the sign, exponent, and
 fraction fields: }
PROCEDURE BINOUT(NUM:TWOTYPES);
  VAR K:BITINDEX;     {An index variable }

  BEGIN
    {Write a 1 or Ø for the sign bit, then a space:}
    IF NUM.BITS[31] THEN WRITE('1') ELSE WRITE('Ø');
    WRITE(' ');
    {Scan the 8 bits of the exponent field, most
     significant first, then a space: }
    FOR K:=3Ø DOWNTO 23 DO
      CASE NUM.BITS[K] OF
        TRUE: WRITE('1');
        FALSE:WRITE('Ø')
      END;
    WRITE(' ');
    {Scan the 23 bits of the fraction field, most
     significant first: }
    FOR K:=22 DOWNTO Ø DO
      CASE NUM.BITS[K] OF
        TRUE: WRITE('1');
        FALSE:WRITE('Ø')
      END
  END;

{Main program: }
BEGIN
  {Prompt the user for a real number: }
  WRITE('Type Number: ');
  {Store it as a real value: }
  READLN(VALUE.REALVAL);
  {Display the bits: }
  BINOUT(VALUE);
  WRITELN;WRITELN
END.
```

Byte-Oriented Built-Ins Revisited

Chapter 13 describes the byte-oriented routines FILLCHAR, MOVELEFT, MOVERIGHT, SCAN, and SIZEOF. It mentions that the parameters for these routines are not type-checked; note that

these procedures provide yet another way to defeat strong typing. For example, if you have the declarations

```
VAR BIT: PACKED ARRAY[Ø..15] OF BOOLEAN;
    BOOL: BOOLEAN;
```

then you can transfer the value of BOOL into the bit array BIT by means of the statement

```
MOVELEFT (BOOL, BIT, 2)
```

which moves two contiguous bytes (one word) without checking data type.

Special Uses of UNITSTATUS

Chapter 12 describes most functions of the UNITSTATUS procedure. The form is

```
UNITSTATUS ( UNITNUM, DATA, OPTION )
```

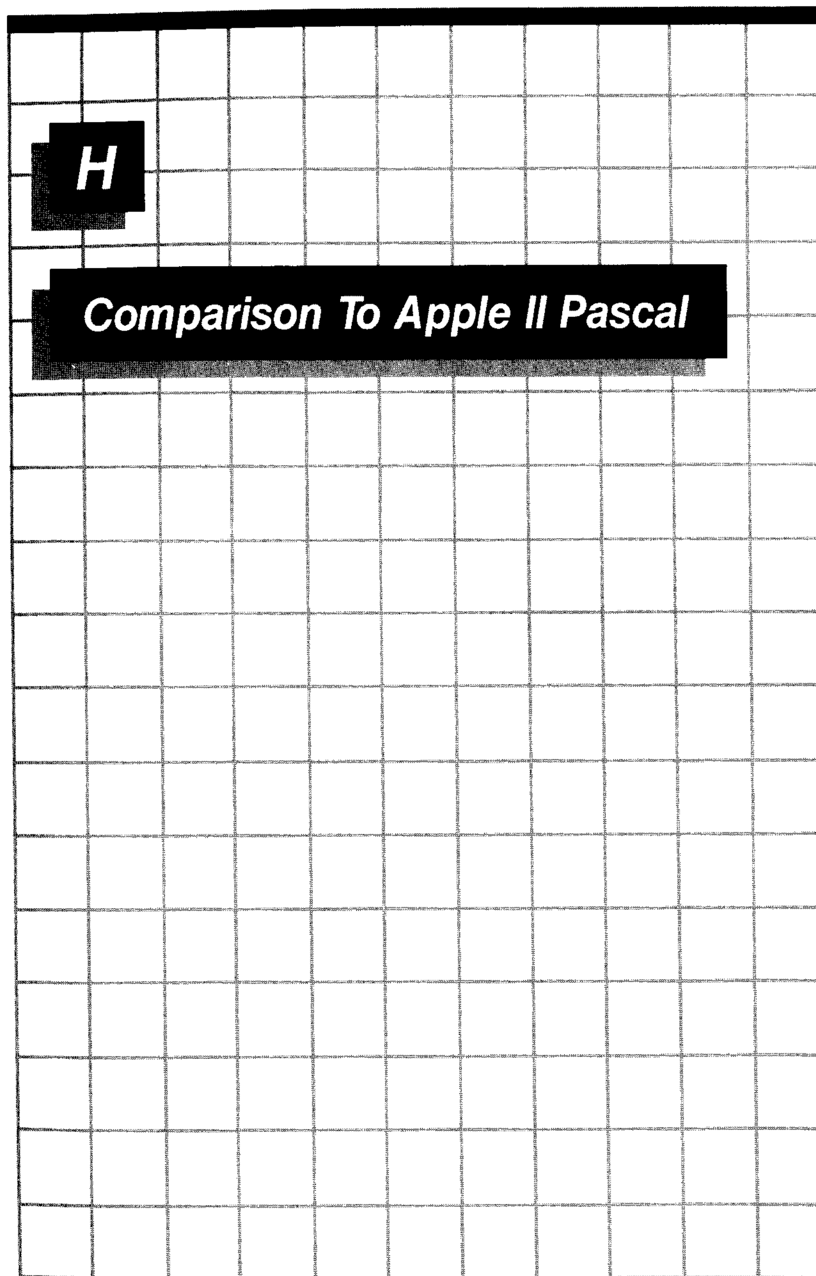
In Chapter 12, UNITNUM is required to be the unit number of an I/O device. However, special operations are invoked if UNITNUM has value Ø (which is not the unit number of any Pascal unit).

If UNITNUM = Ø and OPTION = Ø, UNITSTATUS returns a structure containing SOS device numbers corresponding to the Pascal I/O units numbered 1-2Ø and 128-147. In this case, DATA should be a record declared as follows:

```
DATA : PACKED RECORD
         REGULARUNITS : PACKED ARRAY [1..2Ø] OF Ø..255;
         USERUNITS    : PACKED ARRAY [128..147] OF Ø..255
       END;
```

Array elements corresponding to non-existent units are set to Ø.

If UNITNUM = Ø and OPTION = 1, UNITSTATUS reinitializes the console to the same state it was initialized to at boot time. This provides a means for the user to restore "normal" console operation after having altered it for some purpose. In this case, DATA may be of any type; it is not used.



H

Comparison To Apple II Pascal

The great majority of Apple II Pascal programs can be recompiled and executed on the Apple III without modification.

The following is a summary of significant differences between Apple II Pascal and Apple III Pascal.

OTHERWISE Clause in CASE Statement

Apple III Pascal provides an OTHERWISE clause in the CASE statement. The OTHERWISE clause, if present, contains a statement that is executed if none of the cases in the CASE statement are executed. See Chapter 5.

SOS Pathnames

SOS pathnames are different from the Pascal filenames used on the Apple II. Apple III Pascal supports both kinds of names, as explained in the Introduction, Filer, and Editor manual.

SOS Device Driver Support

All SOS device drivers are supported by Apple III Pascal as "I/O units." See Chapters 1Ø-12 and the Standard Device Drivers Handbook.

Graphics

The Apple III screen graphics modes differ significantly from the Apple II, and the graphics screen is driven through the SOS graphics driver (see Standard Device Drivers Handbook). Therefore, a new unit named PGRAF is supplied as a high-level interface to the graphics driver.

TURTLEGRAPHICS is available only for compatibility with Apple II—see Appendix K.

New Procedures

DATE, TIMEOFDAY, CLOCKINFO, and SETTIME are procedures provided in the APPLESTUFF unit for reading and setting the Apple III system's internal date and time. See Appendix D.

JOYSTICK and SOUND are procedures contained in the APPLESTUFF unit, to support the joystick device and the Apple III built-in sound generator. Note that PADDLE, BUTTON, and NOTE are also supported. See Appendix D.

New Data Types

The BYTESTREAM and WORDSTREAM types are provided for use as types of VAR parameters in procedure and function definitions. See Chapter 13.

Real Arithmetic

For operations on values of type REAL, Apple III Pascal conforms to the proposed IEEE floating-point standard. Under default conditions, the difference between this and the arithmetic of Apple II Pascal is invisible unless the program peforms operations with exceptional results (such as division by zero). See Appendix E for complete details.

Library Files and Units

In addition to the SYSTEM.LIBRARY file, each codefile under Apple III Pascal can have a "program library" file associated with it. This makes the use of library units more convenient and allows a program to have up to 48 segments at run time.

When compiling a unit, it is no longer necessary in all cases to use the Compiler's swapping option.

Memory Organization

The memory organization of the Apple III under SOS and Pascal is different from the memory organization of the Apple II under Pascal. This makes no difference to most programs. However, the amount of memory available for a user program will be somewhat greater on the 128K Apple III than on the Apple II.

Memory organization might adversely affect an Apple II program if it depends on pointer values created while running on an Apple II and stored in a diskette file. Any such values will of course be incorrect on the Apple III.

Similarly, an Apple II program that depends on explicit Apple II hardware addresses will not work on the Apple III. This might affect Apple II Pascal programs that are designed to drive the Silentype printer; such programs should be revised to use the Apple III Silentype driver described in the Standard Device Drivers Handbook.

The UNITSTATUS Procedure

For device-oriented I/O, the UNITSTATUS procedure is supported. See Chapter 12.

Runtime Segment Table

The runtime segment table allows for 64 segments instead of 32. See Chapter 15.

Conditional Compilation

The Apple III Pascal Compiler allows conditional compilation. See Appendix F.

The CHAINSTUFF Unit

Since Apple III Pascal has no "system swapping" mode, the SWAPON and SWAPOFF procedures are absent from the CHAINSTUFF unit.

Compiling Apple II Code

The Apple III Pascal Compiler can compile code to run on the Apple II. See Appendix F.

File Variable Size

Every declared file in an active procedure requires 1,1ØØ bytes of memory.

Compiler Options

Option names can be spelled out.

Because Compiler options always end with a comma, they can be chained together (except for the INCLUDE option). Therefore, the COMMENT option cannot contain a comma, and the RESIDENT option does not accept a list.

Procedure Complexity

A more complex procedure may be compiled on the Apple III than on the Apple II because of the Apple III's larger memory.

System Globals

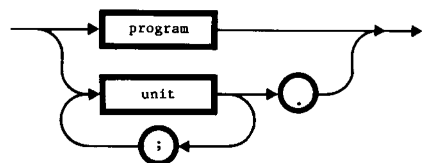
Users of the {$USER-} option may find that their programs are not portable.

I

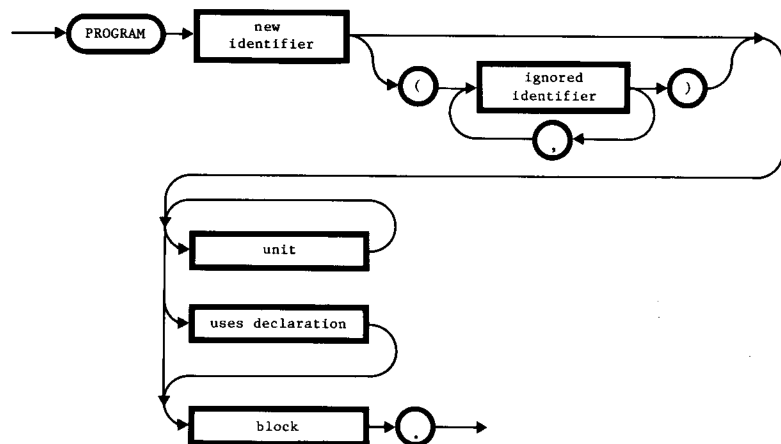
Syntax Diagrams

These diagrams give the formal syntax for all Apple III Pascal constructions.

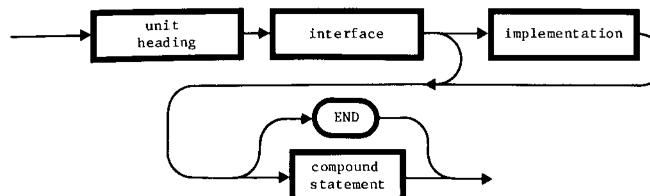
compilation



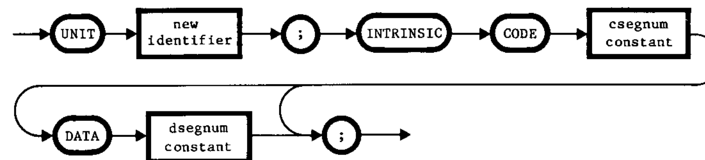
program



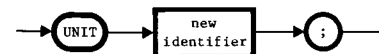
unit



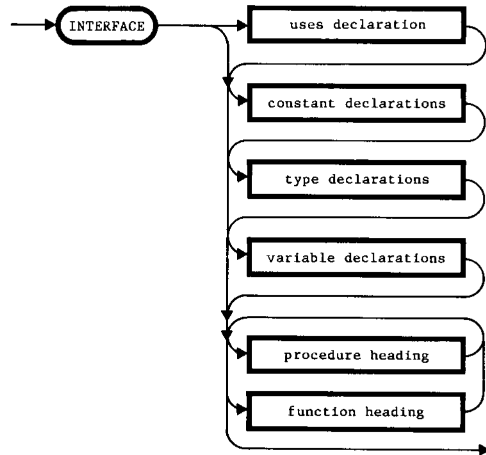
intrinsic unit heading



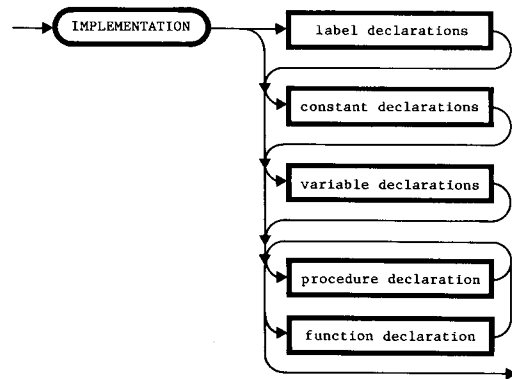
regular unit heading



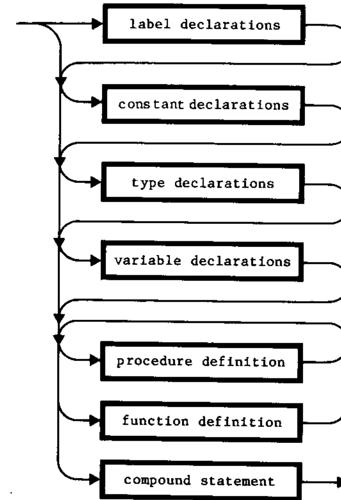
interface



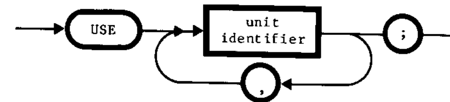
implementation



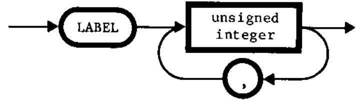
block



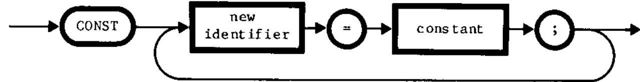
uses declarations



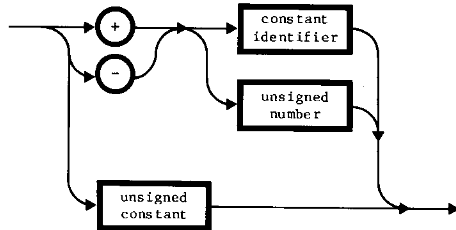
label declarations



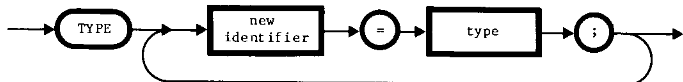
constant declarations



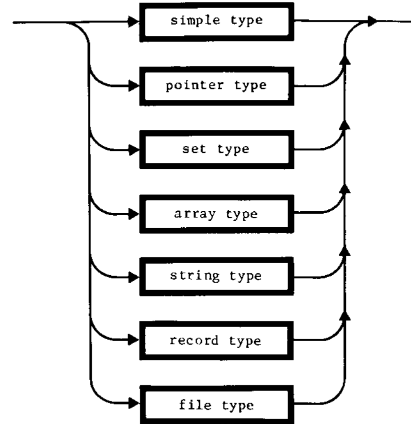
constant



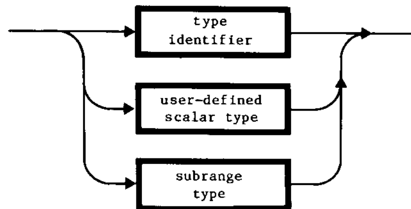
type declarations



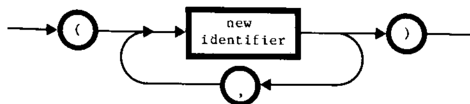
type



simple type

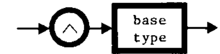


user-defined scalar type

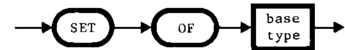


subrange type

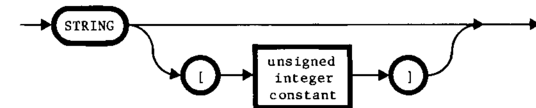
pointer type



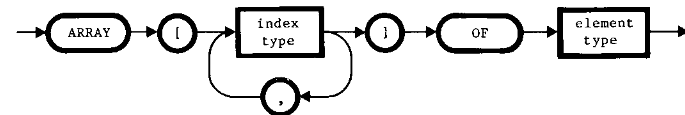
set type



string type

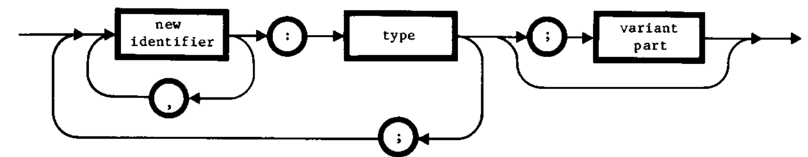


array type

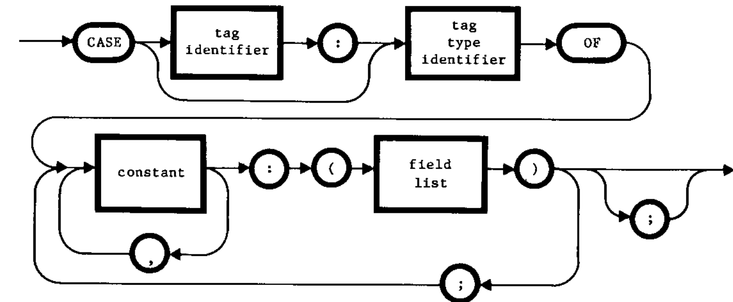


record type

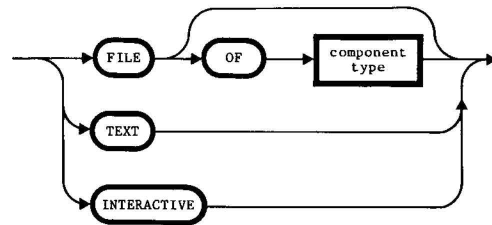
field list



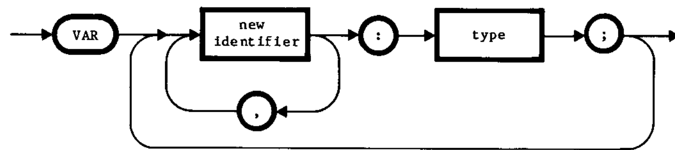
variant part



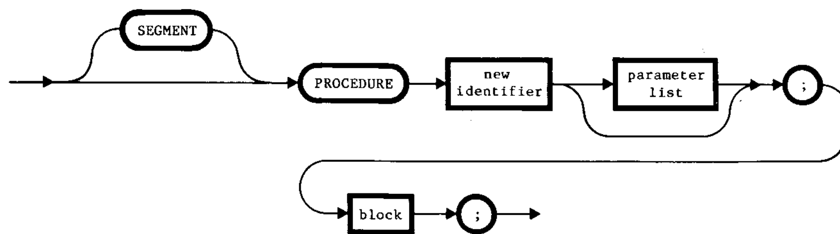
file type



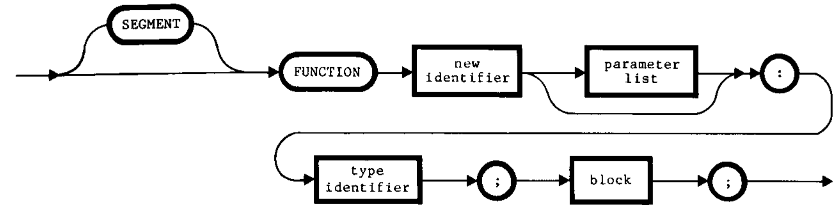
variable declarations



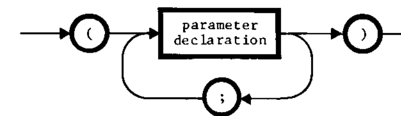
procedure definition



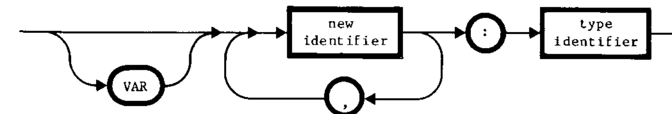
function definition



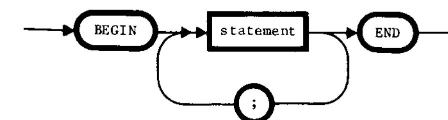
parameter list



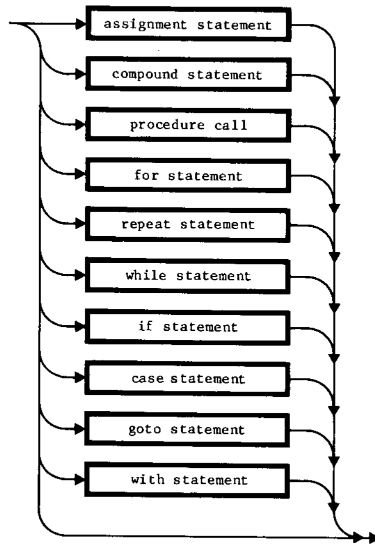
parameter declaration



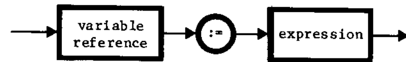
compound statement



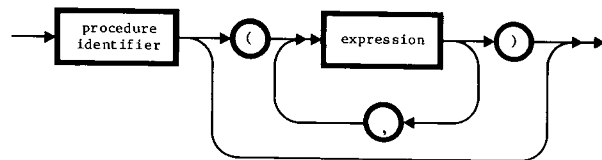
statement



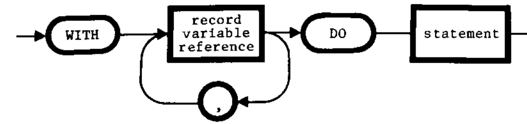
assignment statement



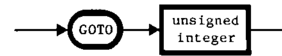
procedure call



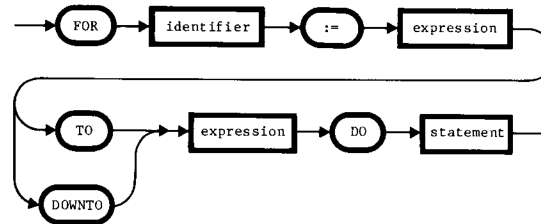
with statement



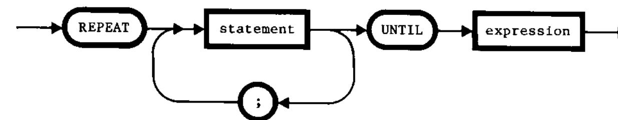
goto statement



for statement

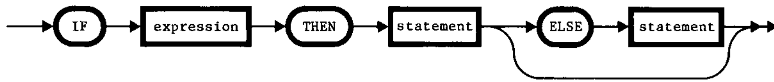


repeat statement

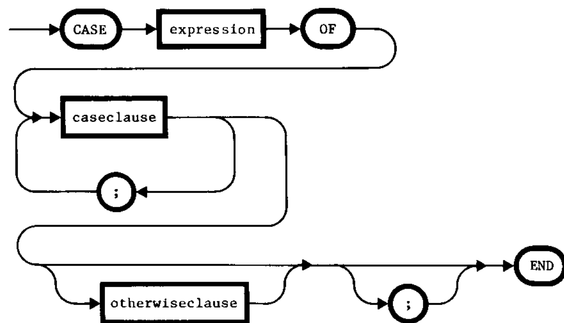


while statement

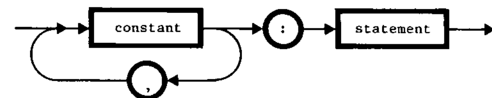
if statement



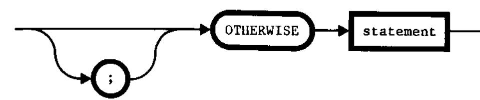
case statement



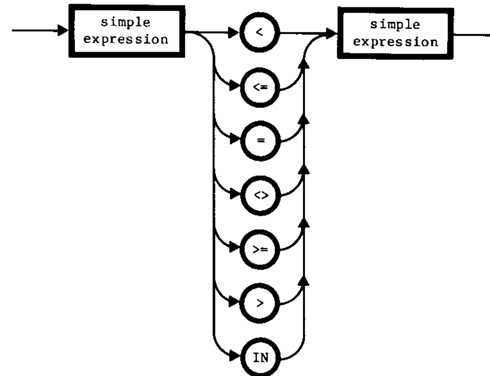
case clause



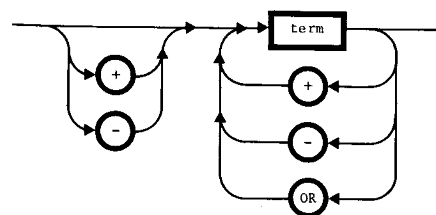
otherwise clause



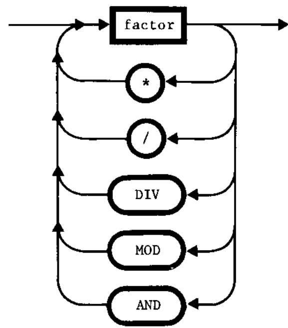
expression



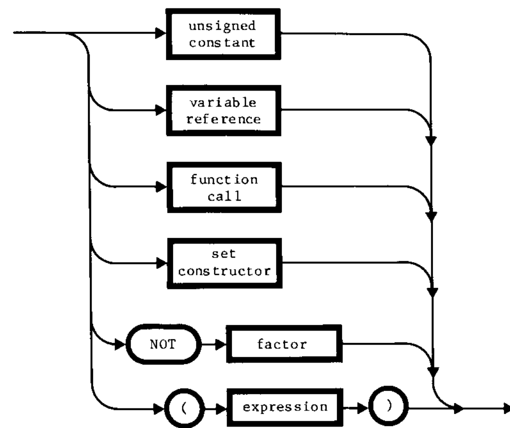
simple expression



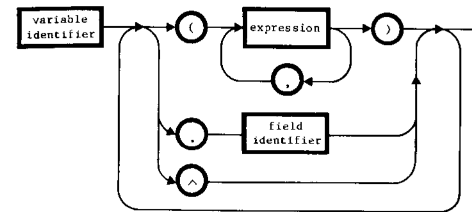
term



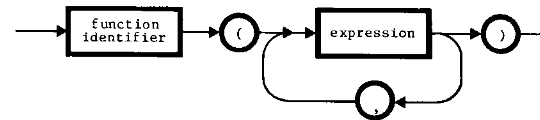
factor



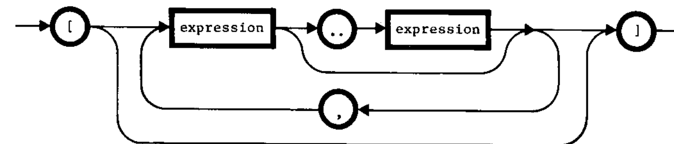
variable reference



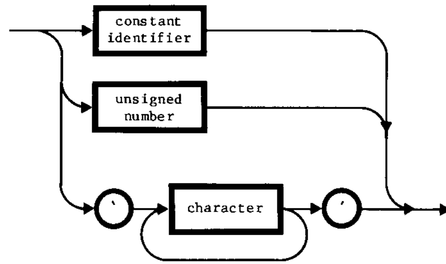
function call



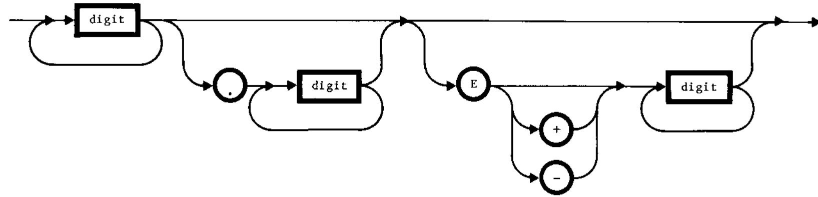
set constructor



unsigned constant

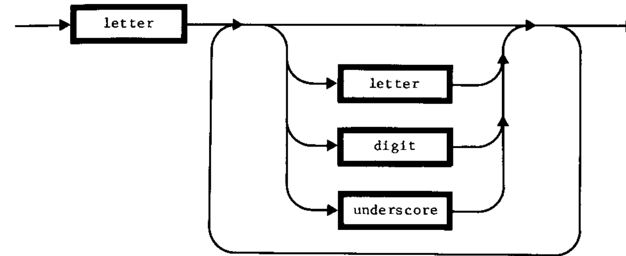


unsigned number



unsigned integer

identifier



J

Tables

Table 1: Execution Errors

1	Value range error
2	No procedure in segment table
3	Exit from uncalled procedure
4	Stack overflow
5	Integer overflow
6	Divide by zero
7	NIL pointer reference
8	Program interrupted by user
9	System I/O error
1Ø	User I/O error
11	Unimplemented instruction
12	Floating point error
13	String overflow
14	Programmed HALT
15	Programmed break-point

When one of these errors occurs, a message is displayed giving the error number followed by a segment number, a procedure number, and a byte number. These numbers relate to the program listing; see the description of the "Listing" option in Appendix F.

Table 2: I/O Errors

This table lists all the error numbers that can possibly be returned by the IORESULT function (see Chapter 1Ø). The notation (SOS) in the table indicates an error reported by SOS; some of the SOS errors are unlikely under the Pascal system, and are included here for completeness only.

Ø	No error; normal I/O completion
2	Bad unit number
3	Illegal operation (e.g., read from PRINTER:)
5	Lost unit -- no longer on line
6	Lost file -- file is no longer in directory
7	Illegal pathname
8	No room -- insufficient space on diskette
9	No unit -- unit is not on line
1Ø	No such file in specified directory
11	Duplicate pathname
12	Attempt to open an already open file
13	Attempt to access a closed file
14	Bad input format -- error in reading number
15	Ring buffer overflow -- input arriving too fast
16	Write-protect error -- diskette is protected
19	Too many files open for system to handle
32	(SOS) Invalid request code
34	(SOS) Invalid control parameter list
35	(SOS) Character device not open
36	(SOS) Device not available
37	(SOS) Resource not available
44	(SOS) Invalid byte count
45	(SOS) Invalid block number
48..63	(SOS) Device-specific error
64	Device error -- bad address or data on diskette
65	(SOS) Too many character files open
66	(SOS) Too many block files open
67	(SOS) Invalid file reference number
73	(SOS) Directory full
74	(SOS) Incompatible file format
75	(SOS) Unsupported storage type
76	(SOS) Attempted read past end of file
77	(SOS) File position out of range
78	(SOS) Illegal access attempted
79	(SOS) User's buffer too small

8∅	(SOS) File busy
81	(SOS) Volume format neither SOS nor Apple II
83	(SOS) Invalid value in list parameter
84	(SOS) Out of memory for SOS system buffer
85	(SOS) Buffer table full
86	(SOS) Invalid system buffer parameter
87	(SOS) Duplicate volume error
123..127	(SOS) System call error

These errors result in a run-time halt unless the Compiler option {$IOCHECK-} is used to turn off I/O checking. With I/O checking off, the I/O error number can be returned to the program by the built-in function IORESULT (see Chapter 1∅).

Table 3: Reserved Words

These are words that have fixed meanings in Pascal. You can never use them as identifiers without causing a Compiler error. The next table lists some more words you should not use as identifiers.

Standard Pascal Reserved Words

AND
ARRAY
BEGIN
CASE
CONST
DIV
DO
DOWNTO
ELSE
END
FILE
FOR
FORWARD
FUNCTION
GOTO
IF
IN
LABEL
MOD
NIL
NOT
OF
OR
PACKED
PROCEDURE
PROGRAM
RECORD
REPEAT
SET
THEN
TO
TYPE
UNTIL
VAR
WHILE
WITH

Additional Apple III Pascal Reserved Words

EXTERNAL
IMPLEMENTATION
INTERFACE
OTHERWISE
SEGMENT
UNIT
USES

Table 4: Predefined Identifiers

These are the identifiers of the built-in procedures and functions and the predefined types and variables of Apple III Pascal. The list does not include those identifiers that are declared or defined in the standard library units. If you declare or define one of these identifiers in your program, no error will result but you will lose the capability of the corresponding built-in or predefined entity.

With each identifier, a code is shown to the left to indicate what kind of object the identifier represents. The codes are

Code	Meaning	Code	Meaning
p	procedure	i	integer function
b	boolean function	r	real function
t	type	c	char function
k	constant	f	file
s	string function	-	other

Code	Identifier	Code	Identifier	Code	Identifier
r	ABS	t	INTERACTIVE	p	REWRITE
i	BLOCKREAD	i	IORESULT	i	ROUND
i	BLOCKWRITE	f	KEYBOARD	i	SCAN
t	BOOLEAN	i	LENGTH	p	SEEK
t	BYTESTREAM	p	MARK	i	SIZEOF
t	CHAR	k	MAXINT	r	SQR
c	CHR	i	MEMAVAIL	s	STR
p	CLOSE	p	MOVELEFT	t	STRING
s	CONCAT	p	MOVERIGHT	-	SUCC
s	COPY	p	NEW	t	TEXT
p	DELETE	b	ODD	i	TREESEARCH
b	EOF	i	ORD	k	TRUE
b	EOLN	f	OUTPUT	i	TRUNC
p	EXIT	p	PAGE	b	UNITBUSY
k	FALSE	i	POS	p	UNITCLEAR
p	FILLCHAR	-	PRED	p	UNITREAD
p	GET	p	PUT	p	UNITSTATUS
p	GOTOXY	r	PWROFTEN	p	UNITWAIT
p	HALT	p	READ	p	UNITWRITE
p	IDSEARCH	p	READLN	t	WORDSTREAM
f	INPUT	t	REAL	p	WRITE
p	INSERT	p	RELEASE	p	WRITELN
t	INTEGER	p	RESET		

Table 5: Compiler Error Messages

When the Pascal Compiler discovers an error in your program, it reports that error immediately, by error number. If you then enter the Editor to fix that error, a more complete error message is given, taken from the system diskette file SYSTEM.SYNTAX . If you remove the file SYSTEM.SYNTAX from the system diskette, errors will be reported by number only.

The Pascal Compiler error message corresponding to each error number is given in the table below. Some people will prefer to gain some additional space on their system diskette, by removing SYSTEM.SYNTAX and using this table instead. You can also print your own copy of this table by transferring the file SYSTEM.SYNTAX to a printer.

Some additional helpful information is provided in Table 5, enclosed in square brackets [ ]. This information is not part of the file SYSTEM.SYNTAX ; it cannot be printed, and it will not appear on your screen.

1: Error in simple type
2: Identifier expected
3: 'PROGRAM' expected
4: ')' expected
5: ':' expected
6: Illegal symbol (maybe missing or extra ';' on line above)
7: Error in parameter list
8: 'OF' expected
9: '(' expected
1Ø: Error in type
11: '[' expected
12: ']' expected
13: 'END' expected
14: ';' expected (possibly on line above)
15: Integer expected
16: '=' expected
17: 'BEGIN' expected
18: Error in declaration part
19: Error in field-list
2Ø: ',' expected
21: '.' expected
22: 'Interface' expected

23: 'Implementation' expected
24: 'CODE' expected
50: Error in constant
51: ':=' expected
52: 'THEN' expected
53: 'UNTIL' expected
54: 'DO' expected
55: 'TO' or 'DOWNTO' expected in FOR statement
58: Error in factor (bad expression)
59: Error in variable
101: Identifier declared twice
102: Low bound exceeds high bound
103: Identifier is not of the appropriate class
[Maybe a packed variable is being used where an unpacked variable is required.]
104: Undeclared identifier
105: Sign not allowed
106: Number expected
107: Incompatible subrange types
108: File not allowed here
[A file may not be part of a record or an array; a file may not be the object of a pointer.]
109: Type must not be real
110: Tagfield type must be scalar or subrange
111: Incompatible with tagfield part
113: Index type must be a scalar or a subrange
114: Base type must not be real
115: Base type must be a scalar or a subrange
117: Unsatisfied forward reference
119: Re-specified params not OK for a forward declared procedure
120: Function result type must be scalar, subrange or pointer
121: File value parameter not allowed
122: The result type of a forward declared function cannot be re-specified
123: Missing result type in function declaration
125: Error in type of standard procedure parameter
126: Number of parameters does not agree with declaration
128: Illegal operation for this file type
[BLOCKREAD and BLOCKWRITE must be to untyped files; other operations must be to typed files. Certain operations are restricted to character files. See Chapter 12.]
129: Type conflict of operands
130: Expression is not of set type
131: Only tests on equality are allowed
132: Strict inclusion not allowed

133: File comparison not allowed
134: Illegal type of operand(s)
135: Type of operand must be boolean
136: Set element type must be scalar or subrange
137: Set element types must be compatible
138: Type of variable is not array
139: Index type is not compatible with the declaration
140: Type of variable is not record
141: Type of variable must be file or pointer
142: Illegal actual parameter
143: Illegal type of loop control variable
144: Illegal type of expression
145: Type conflict
146: Assignment of files not allowed
147: Label type incompatible with selecting expression
148: Subrange bounds must be scalar
149: Index type must not be integer
150: Assignment to standard function is not allowed
152: No such field in this record
154: Actual parameter must be a variable
155: Control variable cannot be formal or non-local
156: Multidefined case label
158: No such variant in this record
159: Real or string tagfields not allowed
160: Previous declaration was not forward
161: Forward declared twice
162: Parameter size must be constant
[Optional parameters in NEW must be constants.]
165: Multidefined label
166: Multideclared label
167: Undeclared label
168: Undefined label
[165-168: In order to "declare" a label you must include it in the LABEL declaration section; in order to "define" a label you must specify it before the statement to which it refers in the body of the procedure. A label must be declared and defined exactly once.]
169: Base type of set too large
175: Actual parameter max string length < formal max length
182: Nested units not allowed
183: External declaration not allowed at this nesting level
184: External declaration not allowed in interface section
185: Segment declaration not allowed in unit
186: Labels not allowed in interface section
187: Attempt to open library unsuccessful
188: Unit not declared in previous uses declaration

189: 'Uses' not allowed at this nesting level
190: Unit not in library
191: No private files in unit
192: 'Uses' must be in interface section
195: Unit not importable (interface text not available)
201: Error in real number--digit expected
202: String constant must not exceed source line
203: Integer constant exceeds range
250: Too many scopes of nested identifiers
251: Too many nested procedures or functions
253: Procedure too long
[A procedure is too long when it overflows the internal code buffer used by the Compiler. Solution: break off one or more nested procedures.]
254: Procedure too complex
[A procedure is too complex when it generates too many long jumps (i.e., too many control structures).]
260: New compile-time variable must be declared at global level
261: Undefined compile-time variable
262: Error in compile-time expression
263: Conditional compilation options nested too deeply
264: Unmatched ELSEC
265: Unmatched ENDC
266: Error in SETC
267: Unterminated conditional compilation option
271: Comment must appear at top of program
[Certain Compiler options must appear before the word PROGRAM.]
272: Invalid symbol in Compiler option
273: No such unit or segment
274: Invalid segment number
275: Include must be last option
276: Invalid code version type
301: Not enough room for case jump table
[When a case statement is compiled, a jump table is generated with one entry for each value between the minimum and maximum case selectors. A short case statement with a wide range between minimum and maximum selectors may result in a very large piece of code.]
350: No data segment allocated
[An intrinsic unit with global variables in either the INTERFACE or the IMPLEMENTATION requires a data segment. The data segment must not be declared unless it is used.]
352: No code segment allocated
353: Non-intrinsic unit called from intrinsic unit

354: Too many segments for segment dictionary
355: Data segment empty
399: Implementation restriction
[May be one of the following:
- subrange of real is not allowed;
- segment procedures, segment functions, and units must be declared before regular procedures;
- the word SEGMENT must apply to PROCEDURE or FUNCTION only;
- the defining text of a forward-declared segment must repeat the word SEGMENT;
- no external segments are allowed.]

400: Illegal character in text
401: Unexpected end of input
[Possible causes include:
- mismatched BEGIN and END;
- omitting the period after the final END;
- unterminated comment;
- unterminated conditional compilation;
- procedure without a body.]

402: Error in write to code file, maybe not enough room on disk
403: Error while opening or reading include file
404: Bad open, read, or write to Linker file SYSTEM.INFO
[See Program Preparation Tools manual.]
405: Error while reading library
406: Include file not legal in interface nor while including
408: General Compiler error

Table 6: ASCII Character Codes

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
Ø	ØØ	NUL	32	2Ø	SP	64	4Ø	@	96	6Ø	`
1	Ø1	SOH	33	21	!	65	41	A	97	61	a
2	Ø2	STX	34	22	"	66	42	B	98	62	b
3	Ø3	ETX	35	23	#	67	43	C	99	63	c
4	Ø4	EOT	36	24	$	68	44	D	1ØØ	64	d
5	Ø5	ENQ	37	25	%	69	45	E	1Ø1	65	e
6	Ø6	ACK	38	26	&	7Ø	46	F	1Ø2	66	f
7	Ø7	BEL	39	27	'	71	47	G	1Ø3	67	g
8	Ø8	BS	4Ø	28	(	72	48	H	1Ø4	68	h
9	Ø9	HT	41	29	)	73	49	I	1Ø5	69	i
1Ø	ØA	LF	42	2A	*	74	4A	J	1Ø6	6A	j
11	ØB	VT	43	2B	+	75	4B	K	1Ø7	6B	k
12	ØC	FF	44	2C	,	76	4C	L	1Ø8	6C	l
13	ØD	CR	45	2D	-	77	4D	M	1Ø9	6D	m
14	ØE	SO	46	2E	.	78	4E	N	11Ø	6E	n
15	ØF	SI	47	2F	/	79	4F	O	111	6F	o
16	1Ø	DLE	48	3Ø	Ø	8Ø	5Ø	P	112	7Ø	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	5Ø	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
2Ø	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	12Ø	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	9Ø	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	6Ø	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
3Ø	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

The codes in the range 128..255 are not assigned to specific characters, but are nevertheless usable as ASCII code values; see Chapter 3.

Table 7: Standard I/O Devices

The Pascal system identifies each peripheral device by a unit number and a unit name. SOS device names may also be used. The standard unit names and numbers are

SOS DEVICE NAME	PASCAL UNIT #	PASCAL UNIT NAME
.CONSOLE	1	CONSOLE:
.CONSOLE	2	SYSTERM:
.GRAFIX	3	GRAPHIC:
.D1	4	(volume name)
.D2	5	(volume name)
.PRINTER	6	PRINTER:
.RS232	7	REMIN:
.RS232	8	REMOUT:
.D3	9	(volume name)
.D4	1Ø	(volume name)

The distinction between units 1 and 2 (CONSOLE: and SYSTERM:) is that I/O operations using SYSTERM: (unit 2) do not cause typed characters to be echoed on the screen. The built-in Pascal file identifier KEYBOARD is associated with the SYSTERM: unit, and the built-in Pascal file identifiers INPUT and OUTPUT are associated with the CONSOLE: unit.

The distinction between units 7 and 8 (REMIN: and REMOUT:) is that unit 7 is used for input and unit 8 is used for output. Both refer to the SOS device .RS232 .

Note that if there is a printer-like drive present (e.g., .SILENTYPE) and no .PRINTER , the printer-like driver is assigned the Pascal unit name PRINTER: and its unit number is 6.

Non-standard devices are assigned sequential unit numbers in the range 128 through 255 according to their order in SOS.DRIVER . (See the Standard Device Drivers Handbook.)

Table 8: Size Limitations

Maximum size of any one procedure (including main program): approximately 12ØØ bytes of compiled code (see note below)

Default maximum length of the STRING value in a variable declared without explicit size: 8Ø

Maximum size that can be declared for a STRING variable: 255

Maximum number of elements in a set: 512

Maximum number of segments in a codefile: 16

Maximum number of segments in a program: 48

Maximum number of procedures and/or functions within a segment: 149

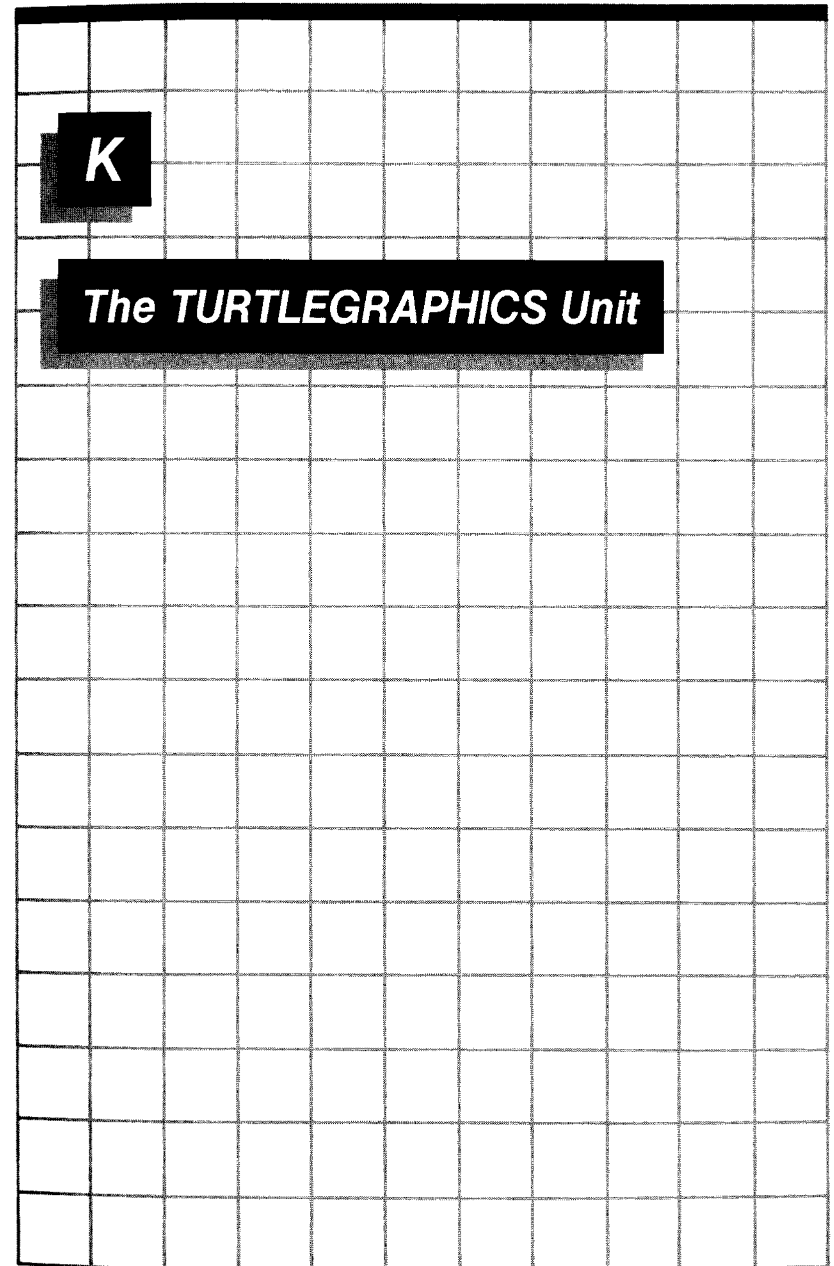
Maximum representable integer value: 32767

Minimum representable integer value: -32768

Maximum representable absolute real value: 3.4Ø2823466E38

Minimum representable absolute non-zero real value: 1.4Ø1298464E-45

The Compiler error message "Procedure too long" means either that the procedure's code exceeds the limit of about 12ØØ bytes, or that the procedure has too much complexity in its control structure. The remedy is described at the end of Chapter 6.



Using Apple II TURTLEGRAPHICS with the Apple III

The TURTLEGRAPHICS unit is available for compatibility with Apple II. TURTLEGRAPHICS is described in detail in the Apple II Pascal Language Reference Manual.

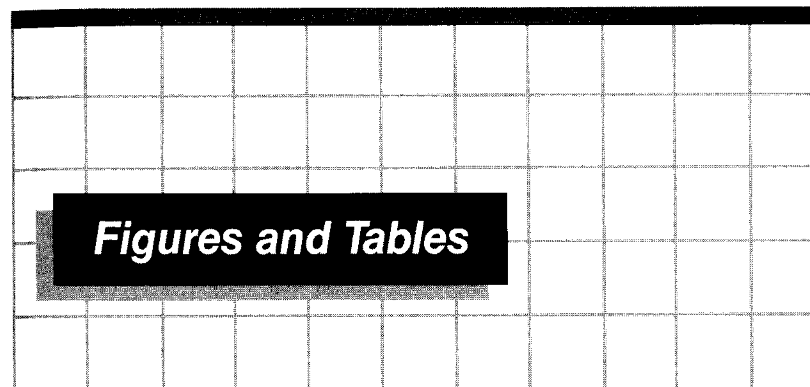
If you have an Apple II program which uses the TURTLEGRAPHICS unit, you can run it on your Apple III without having to change the graphics code. However, you will notice the following important differences:

- Because a different method of color generation is used by the Apple III, TURTLEGRAPHICS uses only the black-and-white mode. Any screencolor other than black is shown as white except in the case of the INVERSE screencolor.
- Use of the screencolor REVERSE is significantly slower, especially when the FILLSCREEN procedure is used.
- The DRAWBLOCK and CHARTYPE routines are available only for modes 4,5,6,8,10,13, and 14. Any other mode is treated by the Apple III as if it were mode 10.
- In text mode, 80 columns are displayed on the Apple III. (The Apple II displays 40 columns.)

Recommended use of TURTLEGRAPHICS on Apple III is limited to cases where

- you want to execute an Apple II program which uses TURTLEGRAPHICS without modifying the code; or
- you want to use your Apple III to develop graphics applications which will run on an Apple II.

We suggest that the PGRAF unit be used for all new Apple III graphics applications. (See Appendix B.)



Figures and Tables

Volume 1—Chapters

1 What is Apple III Pascal? 1

6 Sample Program: FIRSTEXAMPLE

2 Overview of Pascal 7

10 Identifier Syntax
12 Delimiter Characters
23 Arithmetic Operators
23 Comparison Operators
23 Logical Operators
23 Set Operators
30 Sample Program: FIRSTEXAMPLE

3 Simple Data Types 33

36 Constant Declarations Syntax
38 Variable Declarations Syntax
39 Floating-point Number Syntax
40 Exponent Syntax
41 Non-floating-point Number Syntax
46 User-defined Scalar Type Syntax
47 Subrange Type Syntax

4 Expressions and Assignments 51

52 Assignment Statement Syntax
52 Variable Reference Syntax
54 Precedence of Operators
56 Arithmetic Operators
57 Type Results of Multiplication
57 Type Results of Division
58 Type Results of Integer Division
59 Type Results of Addition
59 Type Results of Subtraction
60 Relational Operators
62 Logical Operators with Boolean Operands
63 Relational Operators with Boolean Operands
63 Summary of Type Results
64 Legal Assignments for Non-structured Variables

5 The Flow of Control 65

66 Statement Syntax
68 Compound Statement Syntax
68 Procedure Call Syntax
69 FOR Statement Syntax
72 REPEAT Statement Syntax
72 WHILE Statement Syntax
74 IF Statement Syntax
77 CASE Statement Syntax
77 Caseclause Syntax
78 OTHERWISE Clause Syntax
80 EXIT Procedure Syntax
81 GOTO Statement Syntax

6 Procedures and Functions 83

85 Procedure Definition Syntax
85 Parameter List Syntax
86 Parameter Declaration Syntax
87 Block Syntax
91 Function Definition Syntax
92 Function Call Syntax
98 Nested Program Structure

7 Arrays, Sets, and Strings 103

105 Array Type Syntax
115 Set Type Syntax
117 Set Constructor Syntax
121 String Constant Syntax
122 STRING Type Syntax
125 CONCAT Function Call Syntax

8 Records 129

130 Record Type Syntax
130 Field List Syntax
133 Variant Part Syntax
136 WITH Statement Syntax

9 Pointers and Dynamic Variables 141

144 Pointer Type Syntax
147 NEW Procedure Syntax

10 Introduction to Files and I/O 155

158 File Type Syntax (For Typed File)
163 RESET Procedure Syntax
165 CLOSE Procedure Syntax
167 Effect of CLOSE Options (Opened with REWRITE)
167 Effect of CLOSE Options (Opened with RESET)
171 IORESULT Function Status Codes
177 Sample Program: RANDOMACCESS

11 Text I/O 181

186 EOLN Function Syntax
186 READ Procedure Call Syntax
190 READLN Procedure Call Syntax
192 WRITE Procedure Call Syntax
192 "Value Specifier" Syntax
195 WRITELN Procedure Call Syntax
195 Sample Program: ASCIITABLE
195 Sample Program: FLUSHPERIODS

12 Block File I/O and Device I/O 201

203 BLOCKREAD Function Call Syntax
204 BLOCKWRITE Function Call Syntax
205 Sample Program: FILECOPY
207 Standard Device Numbers and Names
208 UNITREAD and UNITWRITE Procedure Call Syntax
211 UNITSTATUS Procedure
213 UNITSTATUS Procedure: OPTION=21

13 Special-Purpose Built-Ins 217

223 Sample Procedure: UPPERCASE
227 IDSEARCH Declarations

14 Library Units 231

234 Regular Unit
235 Intrinsic Unit
236 Compilation Syntax
236 Unit Syntax
237 Regular Unit Heading Syntax
238 Intrinsic Unit Heading Syntax
241 INTERFACE Syntax
242 IMPLEMENTATION Syntax
243 Sample Unit: OPENS
245 Sample Unit: USEIT

Volume II—Appendices

A TRANSCEND and REALMODES Units 1

8 Square Root, Remainder, and Transcendental Functions: Summary of Special Values and Results

B The PGRAF Unit 11

15 Color Identifiers and Ordinalities
16 Color Transformations
18 Memory Usage
19 Summary of PGRAF Routines
20 Graphics Driver Defaults
28 DRAWIMAGE Parameters
32 Transfer Options
37 PGRAF INTERFACE

C The CHAINSTUFF Unit 39

42 Sample Program Using CHAINSTUFF

D The APPLESTUFF Unit 45

47 Sample Function: Generate Pseudo-Random Integers
52 SETTIME Procedure Fields

E Floating-Point Arithmetic 55

62 Floating-Point Format
63 Specific Number Formats
65 Linear Infinities
66 Circular Infinities
67 Results of Arithmetic with Infinities
69 NaN Format
70 NaN Error Codes
83 Summary of Floating-Point System

F The Apple III Pascal Compiler 87

96 Sample Compilation
110 Compiler Option Summary

G Special Techniques 113

115 16-Bit Binary Word Structure
117 Sample Program: MASKER (Convert to Upper Case)
122 Sample Program: BINARY (Display Integer as Boolean Values)
123 Sample Program: REALBITS (Display Fields of Real Value)

H Comparison To Apple II Pascal 127

I Syntax Diagrams 133

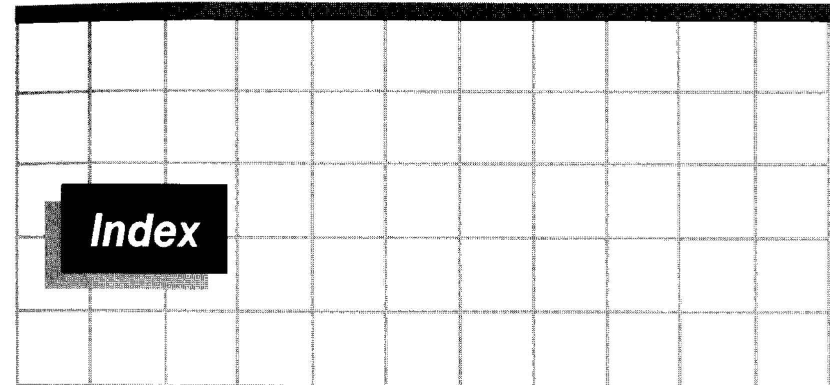
134 Compilation
134 Program
135 Unit
135 Intrinsic Unit heading
135 Regular Unit heading
136 Interface
136 Implementation
137 Block
137 Uses Declarations
138 Label Declarations
138 Constant Declarations
138 Constant
138 Type Declarations
139 Type
139 Simple Type
139 User-defined Scalar Type
140 Subrange Type
140 Pointer Type
140 Set Type
140 String Type
140 Array Type
141 Record Type
141 Field List
141 Variant Part
142 File Type
142 Variable Declarations
142 Procedure Definition
143 Function Definition
143 Parameter List
143 Parameter Declaration
143 Compound Statement
144 Statement
144 Assignment Statement
144 Procedure Call
145 With Statement
145 Goto Statement
145 For Statement
145 Repeat Statement
146 While Statement
146 If Statement
146 Case Statement
146 Case Clause
147 Otherwise Clause
147 Expression
147 Simple Expression

148 Term
148 Factor
149 Variable reference
149 Function Call
149 Set Constructor
150 Unsigned Constant
150 Unsigned Number
150 Unsigned Integer
151 Identifier

J Tables 153

154 Table 1: Execution Errors
155 Table 2: I/O Errors
157 Table 3: Reserved Words
158 Table 4: Predefined Identifiers
159 Table 5: Compiler Error Messages
164 Table 6: ASCII Character Codes
165 Table 7: Standard I/O Devices
166 Table 8: Size Limitations

K The TURTLEGRAPHICS Unit 167



The page numbers in this index do not refer to every occurrence of a word or phrase in the text. Instead, they refer to the locations of significant information on the topic related to the word or phrase.

References in Volume II are shown in square brackets [ ].

A

ABS function 5∅
actual parameter 88
addition 58
address [115]
affine mode [4,65-66]
Algol 2
allocation of memory 132, 142, 148, 151-153, [18]
AND operator 62, [114]
apostrophe 11, 43, 121
APPLE compile-time variable [1∅6]
Apple II formatted diskettes [91]
Apple II Pascal [52,1∅9, 128-131]
Apple III Pascal 2
Apple III Pascal System xv, [ix]
APPLESTUFF unit [46-53]
ARBITRARY function [48]
arithmetic operation accuracy [71]
arithmetic operators 22, 56-6∅
array assignment 15, 11∅
array comparison 11∅-111, 114
array definition 1∅4
array element 1∅4-1∅5
array of indefinite length 222-223
array parameter 84-9∅, 1∅4-1∅5
array representation [119-12∅]
array types 15, 1∅5-11∅, [14∅]
array variable 1∅4-1∅5
ASCII code 43-44, 219, [115, 164]
Asciifile structure 215-216
Asciifile type 183, 213-214
Assembler xv, 1∅∅, [ix]

assembly language 100
assignment operator 52, 64
assignment statement 18, 52, 64, [144]
asterisk 8
ATAN function [6]
audio [50,53]
automatic line feeds 209-210
automatic rounding [58, 71, 78]
automatic type conversion 38

B

base type of pointer 145
base type of set 46, 115, 117
BASIC 3
BASIC text files 183, 215
BCD 42
BEGIN 68, 72
biased exponent [57]
binary floating-point number [57]
binary search 225
binary to decimal conversion [74-75]
bit 111-113, 115-116, 139-140, [17,26,31,115]
block 26-28, 84, 87, 204, [137]
block file declarations 202-203
block file I/O 202-206
block structure 27, 87
block-structured device 157, 202-206, 214
BLOCKREAD function 203-204
BLOCKWRITE function 204-205
boolean logic [114]
BOOLEAN type 14, 45, [115]
boolean values [115]
bubbles xvii, [xi]
buffer variable 159, 170, 183-184, 197-200, 203
built-in 34, 47, [158]
built-in files 184, [158]
built-in procedures and functions 26, 47, [158]
buttons (on joystick) [52]
BW280 graphics mode [13]
BW560 graphics mode [13]
byte 111-113, 218-223, [26,115]
byte-oriented features 218-223, [124]
BYTESTREAM type 222-223

C

case clause [146]
CASE statement 20, 76-79, [146]
chaining [40-43]
CHAINSTUFF unit [40-43]
CHAR type 14, 43, 182-183
character constants 11, 43
character device 156-157, 184, 209
character file 182, 214
character input 184
character output 194
character set 43
CHR function 44
clearing the screen 197
CLOCKINFO procedure [51]
CLOSE options 166-168
CLOSE procedure 160, 164-166
closure [78]
code segment 238, 254-255
code swapping 260-261
codefile 250, 254-255, [88]
COL140 graphics mode [14]
color table [29]
columns in array 106-107
COMMENT compiler option [98]
comments 8
comparison of sets 120
comparison operators 23, 60-62
compilation 236, [134]
Compile command [89]
compile-time error checking [92]
compile-time expressions [106]

compile-time variables [104-106]
Compiler 2, 79, 81, 132, 183, 254, [88-11,159]
Compiler error [92,159-163]
Compiler options [93-104]
compiling Apple II code [109]
compound statement 19, 67, [143]
CONCAT function 125
conditional compilation [104]
conditional statements 73
congruent type 105, 108, 138
conjunction 62
console 156-157, 180
CONSOLE: [165]
CONST 12, 35-36
constant [138]
constant declarations 12, 35-37, 39, 41-47, [138]
control 66-82
control characters 178-180
control variable 21, 69
control-C character 179-180, 187, 194, 209
convert overflow exception [59-60,83]
converting char to string 123
COPY function 126
COS function [6]
CP280 graphics mode [14,35]
CR character 178-179
CRUNCH 165
cursor 224

D

DATA 238
data segment 238, 254-255
data types 13, 34-50
Datafile type 213
date and time [50]
decimal places 193
decimal point 38, 40, 190, 193
decimal to binary conversion [73-74]
declarations 3, 12, 35-47, [96]
DELETE procedure 126
delimiters 11
denormalized number [58, 63-64]
device 156, [165]
device driver 157, 179, 194, 206-207, 211-212
device driver control 210-212
device driver status 207-213
device I/O 206-213
difference operator 119
dimensions of array 105-107
direct recursion 92-95
directory 165, 207
disjunction 62
diskette block numbers 204
diskette file 157, 170
display [17]
display buffer [17-19]
DISPOSE procedure 151
DIV operator 57
dividend 57-58, [61]
division (integer) 57
division (real) 57
division by 0 [61]
divisor 57-58, [61,66,71]
DLE character 179, 194, 215-216
DLE-blank code 214-216
DLE-blank code conversion 209, 215-216
DOTAT procedure [23]
DOTREL procedure [24]
DOWNTO 69-71
DRAWIMAGE procedure [26]
dynamic variable 17, 143-150

E

E notation 39, 193
Editor xv, 2, 183, 214, [ix]
element of array 104-106
ELSE 74
ELSEC compiler option [107]
empty set 117
END 68

end of file 168-169, 197-200
end of line 162, 179, 185-186, 197-200
end of text 179
ENDC compiler option [107]
EOF function 168-169, 180, 187-188, 197-200
EOLN function 180, 185-188, 197-200
error checking [98-100]
error message [92,154-155, 159,163]
ETX character 178-180
exception [58,59-62,77]
exception signal [58]
EXEC/ prefix [40]
Execute command [89]
execution error [96-97,154]
EXIT procedure 80, [41]
EXP function [7]
explicit set value 116
exponent 40, [57]
exponent field [82]
exponential function 50, [7]
expression 22, 52-55, 84, 87-89, [147]
extent of a procedure 97
EXTERNAL 100
external file 156, 158, 213-214
external function 100, [88]
external procedure 100, [88]

F

factor [148]
FALSE 45, 62, 63
field identifier 130
field list 130, 132, [141]
file 156
file block 202-204
file block numbers 204
file buffer variable 159-161, 183-185, 197-200, 203
file component type 158
file declaration 159
FILE OF CHAR 158, 183
file parameter 84-90
file record 158-159
FILE type 202
file type 17, 157-159, [142]
file variable 157, 243
Filer xv, [ix]
fill color [15,21]
FILLCHAR procedure 219, [124]
filling [15]
FILLPORT procedure [24]
fixed-point output 193
floating-point arithmetic [56-85]
floating-point number 13, 38-39, [57]
flow of control 67
font [33]
FOR statement 21, 69, [145]
formal parameters 88
formatted output 195
FORTRAN 4
FORWARD 96
forward definition 96
fraction field [58]
free memory 150-153
free union [120]
function 3, 25, 90-92
function call 25, 92, [149]
function complexity 101
function definition 91, [143]
function heading 236-238
function identifier 91
function size 101
function type 91

G

GET and PUT with text I/O 185, 197-200
GET procedure 169-171, 203, 215
GETCVAL procedure [41]
GLOAD procedure [34]
global 99
GOTO compiler option [100]
GOTO statement 22, 81, [100, 145]
GOTOXY procedure 224
gradual underflow [60]
GRAFIXMODE procedure [20]
GRAFIXON procedure [21]
graphics cursor [13,23]
graphics driver [12,25-26, 34-36]
graphics modes [13]
GSAVE procedure [34]

H

HALT procedure 80, [41]
host program 232-233, 250-251

I

I/O 159, 182-200, 202-203
I/O checking 172-174, [98, 156]
I/O devices 206-207, [165]
I/O error 162-164, 171-172, [155]
I/O facilities 161-162
I/O units 207, [165]
identifier 9, 28, 37, [151]
IDSEARCH 226-229
IEEE floating-point standard 23, 56, [2,56-85]
IF statement 19, 73-75, [146]
IFC compiler option [107]
IMPLEMENTATION 241, [136]
IN operator 117-118
incarnation 94
INCLUDE compiler option [102]
include file [103]
index of array 104-108
index type 104-108
index values 108
indirect recursion 95
inexact result [61]
infinities [58]
infinity [58,63,68]
infinity arithmetic [65-67]
INITGRAFIX procedure [23]
initial value 70
initialization 237, 243, 260
input 161
INPUT built-in file 184
input/output conversions [72-75]
INSERT procedure 126-127
INTEGER constant 41
INTEGER type 13, 40
INTERACTIVE type 158, 183, 216
INTERFACE 240-241, [136]
intersection operator 119-120
intrinsic unit 234-235, 238-239, 254-256, [88]
intrinsic unit heading [135]
invalid operations [61]
IOCHECK compiler option 172, [98,156]
IORESULT function 171-174, 210, [155]

J

Jensen and Wirth 2, [114]
joystick [49]

K

keyboard [48]
KEYBOARD built-in file 184, [158]
KEYPRESS function [48]

L

label 81
label declarations 81, [138]
length attribute (LONG INTEGER, STRING) 42, 124
LENGTH function 124
lexical level [96]
LIBMAP utility program 238-239, 256
Librarian xv, [ix]
library file 100, 232-251, 254-255, [89]
library unit 29, 232-251

LIBRARY utility program 233, 238
limit value 7Ø
line-oriented input 195
line-oriented output 195
linefeed character 179, 193-194, 2Ø9-21Ø
LINEREL [24]
LINETO [23]
linked list 146-147
Linker xv, 1ØØ, 233-234, 251, 254, [ix,88-89]
list 146
LIST compiler option [95]
listing [95, 159]
LN function [6]
loading of segments 258-26Ø
local 99
LOCK 165
LOG function [7]
logarithmic functions [7]
logical operators 23, 62
logical record 158
LONG INTEGER input 188-19Ø
LONG INTEGER output 192
LONG INTEGER type 42
LONGINTIO unit 43, 239

M

machine language 2
MARK procedure 95, 151-153
MAXINT 42
MEMAVAIL function 15Ø-151
member of set 115
memory allocation 139-14Ø, 142-143, 148, 151-152, [18]
minuend 59
mixed reading and writing 185, 197-2ØØ
MOD operator 58
mode parameter (device I/O) 2Ø8-21Ø
modes, arithmetic [4]
MOVELEFT procedure 221-222, [124]
MOVEREL procedure [24]
MOVERIGHT procedure 221-222, [124]
MOVETO procedure [23]
multidimensional array 1Ø6-1Ø7
multiplication 57

N

NaN [5,58,63,69-7Ø]
natural logarithm [6]
negation (arithmetic) 59
negation (boolean) 62
nested IF statements 75
nested WITH statements 137
nesting 97
nesting units 246-249
NEW procedure 143-144, 147-148
next record in file 16Ø-161, 169, 185
NEXTSEG compiler option 257, [1Ø1]
NIL 144
NOLOAD compiler option 26Ø-261, [1Ø1]
Non-standard devices [165]
NORMAL 165
normalized number [58,63]
normalizing mode [4,64,79]
NOT operator 62, [114]
NOTE procedure [52]
NUL character 214, 216
null statement 67
numeric constants 1Ø
numeric environment [79]
numeric functions 49
numeric-string input 188,19Ø
numeric-string output 191

O

object of a pointer 145-147
ODD function 5Ø, [116]
one-character string 123
one-dimensional array 1Ø6

Open Apple key 44
opening a file 159, 163-167
operand 52-64
operating system 159-16Ø
operator 52-64
Options command [12]
OR operator 62, [114]
ORD function 48, [116]
ordinality 48, [116]
OTHERWISE clause 76-79, [128, 147]
output 161
OUTPUT built-in file 184
overflow 43, 56, [59,6Ø]

P

P-code 2, [88]
P-machine 2, [88]
PACKED 113, 139
packed array 111-113
packed character array 113-114
packed record 139
packed variable 9Ø, 111-114, 218
PADDLE function [52]
PAGE compiler option [97]
PAGE procedure 197
parameter 19, 84-9Ø
parameter declaration 86, [143]
parameter list 68, 85, [143]
parentheses in expression 54
Pascal interpreter 2, 255, [88]
Pascal User Manual and Report 2
PASCALIO unit 176, 189, 239
passing arrays 1Ø9
pathname 157, 159, 163, 183, 213, 25Ø, [95,1Ø2]
pen color [15,21]
peripheral device [165]
PGRAF unit [12-38]
physical address 144, 151-153
physical diskette access 17Ø
plotting [14-16,23]
pointer 17, 143-147, 153
pointer assignment 147
pointer comparison 144
pointer reference 145, 147
pointer type 144-145, [14Ø]
pointer variable 144, 147, 152
POS function 124-125
power of ten 224
precedence of operators 53
precision of REALs [76]
PRED function 48
predecessor 48
predefined identifiers [158]
printer 156-157
private 241-242
procedure 3, 24, 84-87
procedure call statement 18, 68, 84-9Ø, [144]
procedure complexity 1Ø1
procedure definition 85, [142]
procedure heading 85, 236-238
procedure identifier 68
procedure size 1Ø1
procedure termination 94-95
Procedure too long error [166]
program [134]
program heading 5
program library 25Ø, 255, [2, 12,4Ø,46,1Ø2]
program listing [95]
program segment 254
program structure 5, 26, 84-1Ø1
projective mode [4,65-66]
pseudo-random number [46-48]
public 232, 24Ø
PURGE 165
PUT procedure 169-171, 2Ø3
PWROFTEN function 224

Q

QUIET compiler option [98]

R

railroad tracks xvii, [xi]
random access 175, 215
random numbers [46-48]
range checking 44, 46
RANGECHECK compiler option [99]
READ procedure 186-187, 197-200
READ with a char variable 187, 197-198
READ with a numeric variable 188, 198
READ with a string variable 188, 199
READLN procedure 190-191, 197-200, 214-215
real arithmetic [76]
real arithmetic environment [76]
REAL type 13, 34, 38-40
REALMODES unit [2,5,70,76-82]
record 130
record assignments 138
record comparisons 138-139
record field 130-131
record numbers 175-176
record parameter 84-90, 130-132
record type 16, 130-131, [141]
record variable 136
recursion 92-96
regular unit 233-234, 237, 254, [88,135]
relational operators 60, 63, [68]
RELEASE procedure 95, 151-153
REM function [5]
REMIN: [165]
REMOUT: [165]
REPEAT statement 20, 71, [145]
repetition statements 69
representation of arrays [119]
representation of REALs [76, 120]
representation of scalars [114]
reserved word 9, 226, [157]
RESET procedure 163-164
RESIDENT compiler option 261-263, [101]
result types 63
return character 169, 187, 193, 209, 214
REWRITE procedure 163
ROUND function 49, [59,78]
rounding [58]
rounding error [71]
rounding mode [71-72,78]
rows in array 106-107
Run command [89]
run-time error [97,154-156]
run-time error checking [97, 154-156]
run-time error message [97, 154-156]
run-time halt [97,156]
run-time segment table 255-256

S

scalar types 13, 34, 40-49
SCALB [62]
SCAN function 220-221, [124]
scope of built-in objects 99
scope of identifiers 28, 97
screen 224
screen control codes 180, 194
screen coordinates [13]
SEEK procedure 175-178
segment 254, [96]
segment dictionary 255
SEGMENT function 99, 239-240, 254, 258-259
segment number 238-239, 256-257, [96,154]
SEGMENT procedure 99-100, 239-240, 254, 258-259
segment table 255-256
semicolon 18, 67-68, 75
set 115
set comparison 120
set constructor 116-117, [149]
set difference 119
set equality 120
set inclusion 120
set inequality 120
set intersection 119
set member 104
set operations 23, 118
set restrictions 117
set types 16, 115, [140]
set union 118
set value 115
set variable 16, 115-117
SETC compiler option [107]
SETCHAIN procedure [40]
SETCTAB procedure [29]
SETCVAL procedure [41]
SETTIME procedure [50]
SETXCPN [59]
sign bit [69]
significand [57]
simple data types 34-50
simple expression [147]
simple type [139]
SIN function [6]
single quote --see "apostrophe"
size limitations 101, [166]
SIZEOF function 112, 218-219, [124]
SOS 156-160, 171-172
SOS character file 182
SOS device name 206-207, [165]
SOS-formatted diskettes 210
SOUND procedure [49]
source text 2, 8, 236, [88]
speaker [46]
special characters 187
special-purpose built-ins 218-229
SQR function 50
SQRT function [5]
square-root function 50, [5]
star --see "comments"
statements 18, 66, [144]
STR procedure 127
string 104
string built-ins 124
string comparison 122-123
string constant 11, 121
string element 123
string index 123
string input 188
string length 124
string output 191
STRING type 15, 121-122, [140]
string value 120
string variable 121-123
strong typing [114]
structured data types 104, 130-140
subexpression 54
subprogram 84
subrange types 14, 46, [140]
subroutine 84
subscript 104
subtraction 59
subtrahend 59
SUCC function 48
successor 48
SWAP compiler option [94]
swapping of code 260-261
symbol table [87]
symbols 9
syntax diagrams xvii-xix, [xi-xiii,135-155]
system font [17]
SYSTEM.COMPILER [88]
SYSTEM.EDITOR [89]
SYSTEM.LIBRARY 29, 43, 176, 194, 233, 250, 256, [2,12, 40,46,88]
SYSTEM.LINKER [89]
SYSTEM.PASCAL [89]
SYSTEM.STARTUP [43]
SYSTEM.SYNTAX [89,92,159]
SYSTEM.WRK.CODE [90]
SYSTEM.WRK.TEXT [88]
SYSTERM: [165]

T

tag field 133, 135
tag identifier 133

tag type 133, 135
term [148]
text I/O procedures 184-185
text in graphics mode [17,25]
text mode [21]
TEXT type 158, 183-185, 216
Textfile structure 214-215
Textfile type 158, 183, 213-214
TEXTON procedure [21]
time and date [50]

U

V

W

WORDSTREAM type 222-223
workfile [88]
WPROTECT 165
WRITE procedure 191-194, 214
write-protected file 165-168
WRITELN procedure 194-196, 214

X

XLOC procedure [25]
XYCOLOR procedure [25]

Y

YLOC procedure [25]

Z

zero value [75]
zero-length string 123-124, [41]

Symbols

* operator 57
/ operator 57
+ operator 58
- operator 59
> operator 60, 63
= operator 60, 63
< operator 60, 63
>= operator 60, 63
<= operator 60, 63
<> operator 60, 63
{$G+} compiler option 81, [100]
{$I-} compiler option 172, [98,156]
{$I+} compiler option 172, [98]
{$N+} compiler option 260, [101]
{$NS n} compiler option 257, [101]
{$R identifier} compiler option 261, [101]
{$S+} compiler option [94]
{$U filename} compiler option [102]
.ASCI suffix 213
.CODE suffix [90]
.LIB suffix 250
.GRAFIX [12]
.TEXT suffix 183, 213, [90]