

THINK Pascal™

The Fastest Way to Finished Software

**User
Manual** ◆
◆
◆

THINK PascalTM

The Fastest Way to Finished Software.

U S E R M A N U A L

Credits

User Manual	Philip Borenstein and Jeff Mattson
THINK Pascal Application	Rich Siegel, John McEnerney, and David Neal,
THINK Class Library	Don Podwall and Gregory H. Dow
Quality Assurance	David Allcott, Michael Rockhold, and Paul Vetri
Technical Support	Michael Carland, Mark Geschelin, Phil Shapiro
Marketing Manager	Susan Smith
Product Manager	Philip Borenstein

Copyright © 1988, 1990, 1991 Symantec Corporation. All Rights Reserved. Printed in U.S.A.

Symantec Corporation
10201 Torre Avenue
Cupertino, CA 95014
408/253-9600

THINK C and THINK Pascal are trademarks of Symantec Corporation. Other brands and their products are trademarks of their respective holders and should be noted as such.

ResEdit, SAREz, and SAdRez are copyrighted programs of Apple Computer, Inc. licensed to Symantec Corp. to distribute for use only in combination with THINK Pascal. Apple software shall not be copied onto another diskette (except for archive purposes) or into memory unless as part of execution of THINK Pascal. When THINK Pascal has completed execution, Apple Software shall not be used by any other program.

The *THINK Pascal User Manual* is copyrighted and all rights reserved. Information in this document is subject to change without notice and does not represent a commitment on the part of Symantec Corporation. The software described in this document is furnished under a license agreement. The document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Symantec Corporation. The *THINK Pascal User Manual* contains samples of names and addresses to illustrate features and capabilities of THINK Pascal. Any similarities to names and addresses of actual individuals is purely coincidental.

SYMANTEC CORPORATION MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, REGARDING THE ENCLOSED COMPUTER SOFTWARE PACKAGE, ITS MERCHANTABILITY, OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

Contents

ONE GETTING STARTED

1	Welcome.....	3
	Introduction.....	3
	What is THINK Pascal?.....	3
	What You Need.....	5
	What's in the Package.....	5
	What's in the Manuals.....	6
	What You Should Know.....	9
2	Installing THINK Pascal	13
	Introduction.....	13
	Summary.....	13
	Instructions.....	14
	What's in the Archives.....	16
	Disk Layout Diagram	18
	About the Self-Extracting Archives.....	18

TWO LEARNING THINK PASCAL

3	Tutorial: Bullseye.....	21
	Introduction.....	21
	Creating the Project.....	22
	Writing a Source File.....	26
	Running the Program.....	28
	Debugging the Program	29
	Stepping Through the Program.....	32
	Where to Go Next	33
4	Tutorial: ObjectDraw.....	35
	Introduction.....	35
	Creating the Project.....	36
	Adding the Libraries	37
	Adding the Source Files	40
	Setting the Run Options.....	44
	Setting the Compile Options	46
	Running the Project.....	47
	Building the Application.....	47
	Where to Go Next	49

5	Tutorial: Hex Dump DA	51
	Introduction	51
	Writing Desk Accessories	52
	Creating the Project	52
	Changing a Library	54
	Adding the Source Files	56
	Setting the Compile Options	59
	Setting the Run Options	62
	Segmenting the Project	63
	Running the Project	67
	Building the Desk Accessory	67
	Where to Go Next	72

THREE USING THINK PASCAL

6	Editing	75
	Introduction	75
	Creating and Opening Files	75
	Working with Windows	76
	Editing Files	78
	Searching and Replacing	83
	Working with Files	86
	Customizing Program Formatting	89
	Using MPW Projector	93
7	Working with Projects	95
	Introduction	95
	What Is a Project?	95
	Organizing Your Files	96
	The Project Window	96
	Creating a New Project	98
	Opening an Existing Project	100
	Closing Projects	100
	Adding Files to Projects	101
	Removing Files from Projects	102
	Arranging Files in the Project	102
	Segmenting a Project	103
	Setting the Compiler Options	108
	Getting Information on a Project's Code & Data Size	109
	Customizing the Project Window	110
	Recovering Corrupted Projects	111

8	Running Programs	113
	Introduction	113
	Running a Program	113
	When Something Goes Wrong.....	114
	Stopping Your Program	115
	Compiling, Building, and Linking Without Running	117
	Save Options	117
	Run Options	118
9	Debugging Programs	121
	Introduction	121
	Debugging in THINK Pascal	122
	Following the Finger.....	123
	Stepping Through Programs	124
	Stop Signs	124
	Restarting a Stopped Program	126
	The Execution Commands.....	127
	The Observe window	127
	The Instant window	129
	Examining Compiled Code	129
10	Units and Libraries	131
	Introduction	131
	Using Units	131
	Writing a Unit	132
	Using Libraries	136
	Writing Libraries.....	137
11	Using Predefined Routines	141
	Introduction	141
	Calling Standard Pascal Routines.....	141
	Calling Macintosh Toolbox Routines	142
12	Building Projects.....	147
	Introduction	147
	Setting the Project Type.....	148
	Using Resource Files.....	149
	Building Applications	150
	Building Desk Accessories and Device Drivers	151
	Building Code Resources.....	163
	Putting It Together	171

13	Assembly Language	173
	Introduction.....	173
	The Runtime Environment.....	174
	Pascal Data Types.....	176
	Pascal Calling Conventions.....	180
	Using Assembly Language	182
14	LightsBug	185
	Introduction.....	185
	Using LightsBug.....	186
	Examining Subroutines	188
	Examining Variables.....	191
	Examining Structured Variables	192
	Examining Many Variables.....	194
	Using Watchpoints	194
	Editing Variables	195
	Type Casting Variables.....	195
	Examining Registers	196
	Examining Heap Zones.....	197
	Displaying Memory	198
	Editing Memory	200
	Debugging Toolbox Routines	201
15	Compiler Directives.....	203
	Introduction.....	203
	What Are Compiler Directives?	204
	Using Compiler Directives	207
	Using Conditional Compilation	214
	Using the Compile Options... Command.....	215
FOUR	REFERENCE	
16	THINK Pascal Menus	223
	Introduction.....	223
	The Apple Menu.....	223
	The File Menu.....	224
	The Edit Menu	228
	The Search Menu.....	233
	The Project Menu	236
	The Run Menu	246
	The Debug Menu	250
	The Windows Menu	252

17	Language Reference.....	255
	Introduction	255
	1.0 Tokens and Constants.....	257
	2.0 Blocks, Scopes, and Activations	262
	3.0 Types	266
	4.0 Variables.....	283
	5.0 Expressions	287
	6.0 Statements	301
	7.0 Procedures and Functions	315
	8.0 Programs and Units	327
	9.0 Input/Output	331
	10.0 Standard Procedures and Functions	353
FIVE	UTILITIES	
18	Project Utilities.....	379
	Introduction	379
	Using Project Utilities	379
	Selecting Files.....	383
	Printing Files.....	384
	Backing Up Files	386
19	The Profiler.....	391
	Introduction	391
	Using the Profiler	391
	Reading the Report	392
	Summary	393
20	The Pascal Source Converter	395
	Introduction	395
	Scripts	395
	Input and Output Directories	396
	Specifying Directories and File Names.....	396
	Converting Files	397
	Pascal Source Converter Directives	400
	A Sample Script.....	404
21	Resource Description Files	405
	Introduction	405
	The Resource Compiler and Decompiler.....	406
	Structure of a Resource Description File	407
	Resource Description Statements	409
	Labels	425
	Preprocessor Directives	431
	Resource Description Syntax	434

22	Using SAREz.....	441
	Introduction	441
	What Is SAREz?.....	441
	Choosing Input Files	443
	Choosing an Output File	444
	Setting Options.....	446
	Saving and Restoring Options.....	450
	The Messages Window	451
23	Using SAdRez	453
	Introduction.....	453
	What Is SAdRez?.....	453
	Choosing Input Files	455
	Choosing Output Files.....	457
	Setting Options.....	458
	Saving and Restoring Options.....	460
	The Messages Window	461
24	Using SAPostRez	463
	Introduction.....	463
	What is SAPostRez?	463
	Using SAPostRez	463
SIX	APPENDICES	
A	What's New.....	467
	Introduction.....	467
	Compatibility with Earlier Releases	467
	System 7.0 Compatibility	467
	Working with Large Projects	468
	Working with Small Projects	468
	New and Improved Commands	470
	Enhanced THINK Class Library.....	473
	Other Changes.....	473
B	ANS Pascal Compatibility	475
	Introduction.....	475
	Exceptions to ANS Pascal Requirements	475
	Extensions to ANS Pascal	476
	Implementation-Dependent Features.....	478
	Treatment of Errors	478

C	Porting to THINK Pascal	481
	Introduction	481
	Identifier Length.....	482
	Reserved Words	482
	Comments and Directives.....	482
	The Uses Clause	483
	Types	484
	Data Representation.....	484
	Data Initialization	486
	Operators	486
	Integer Arithmetic	486
	Program Parameters	487
	Predefined Procedures and Functions	487
	Standard Units	488
	Input/Output.....	489
	Run Time Environment.....	490
	Extensions	490
	Using .o Files.....	492
D	Error Messages	493
	Index.....	563

THINK PascalTM

PART ONE

Getting Started

- 1 Welcome
- 2 Installing THINK Pascal

Welcome 1

Introduction

Welcome to THINK Pascal. This chapter tells you about THINK Pascal, what's in your THINK Pascal package, what equipment you need, and what you need to know to start writing Pascal programs on the Macintosh.

Before you begin

Be sure to fill out and return the registration card that came with your THINK Pascal package. Registering insures that you'll get the technical support you need and that you'll be notified of revisions and upgrades to THINK Pascal. Be sure to keep a copy of the serial number for your copy of THINK Pascal in this manual (or some other place where you can be sure you won't lose it). You'll find the serial number on your registration card.

If you don't read manuals

To get started quickly, read this chapter, the next chapter, and do one of the tutorials.

If you're an experienced THINK Pascal user

If you already use THINK Pascal, you'll be pleased with the new features in this release. Read Appendix A, "What's New," to learn about all the new features in THINK Pascal 4.0.

Topics covered in this chapter

- What is THINK Pascal
- What you need
- What's in the package
- What's in the manuals
- What you should know

What is THINK Pascal?

THINK Pascal is a unique development environment for the Macintosh. It features a very fast compiler, a faster linker, an integrated text editor designed specifically for Pascal syntax, an auto-make facility, advanced debugging tools, a class library, a class browser, and a project organizer that holds all the pieces together. Because the editor, the compiler, and the linker are all components of the same application, THINK Pascal knows when edited source files need to be re-compiled. If you edit the interface section of a unit, the auto-make facility recompiles only the source files that depend on it.

With THINK Pascal you can build Macintosh applications, desk accessories, device drivers, and any kind of code resource. The standard Pascal libraries include standard I/O functions like `writeln` so you can use Pascal programs that were written for other computers.

THINK Pascal lets you run your program as you work on it. Your program runs as if you had opened it from the Finder. You can use the debugging tools built into THINK Pascal to make sure your program runs correctly. The debugger lets you set breakpoints, step through your code, examine variables, and change their values—even structured variables like records and arrays—while your program is running.

THINK Pascal is fast. So fast that it will change the way you program. Not only are the compiler and the linker many times faster than other development systems, they are part of an integrated package that also includes an editor, debuggers, an automatic project manager that keeps track of your edits and only recompiles source files that have changed since the program was last built.

THINK Pascal is a remarkable product that delivers the efficiency and power you need. THINK Pascal will seem natural and obvious—the way things ought to be.

A development environment that works with you

In traditional development environments, the edit-compile-link-run cycle takes so long that it's impractical to make a small change just to see how it works. Because it takes so long to compile and link, programmers tend to make several changes at once to get more "bang for the buck." Unfortunately, when your program crashes, you have to figure out which one of the small changes actually caused the crash. The "bang" gets very expensive.

Because THINK Pascal is so fast, you can make a small change, compile and link your program, and run it to see the effect before a traditional development environment even starts linking. And because your program is running in the controlled THINK Pascal environment, you don't have to worry about most of the common crashes. THINK Pascal is right there to catch you so you can start fixing your code without restarting your Macintosh.

In traditional development environments, you have to keep track of various components such as source files, object files, a link-control file, an executable image, and administrative files. THINK Pascal takes care of all this bookkeeping for you. In THINK Pascal, you have only your source files, library files, and a **project document**. The project document serves as an on-line project administrator for your program development. THINK Pascal keeps track of whatever changes you make to your source files, automatically compiling and linking wherever necessary, so all you have to do is edit your program and run.

A rich set of debugging tools

Not only does THINK Pascal help you develop your program faster, it also helps you debug your program faster. THINK Pascal has a rich set of powerful debugging tools.

The integrated editor catches syntax errors and pretty-prints (formats) your program as you type it in. THINK Pascal lets you run your program a statement at a time or in slow motion. You can stop your program any time it's running, or you can set breakpoints anywhere in your program.

The Observe and LightsBug windows let you examine and change the values of your variables — even structured variables like records and arrays. The Instant window lets you try out small pieces of code to see how they'd work in your program.

Advanced features let you stop at Macintosh Toolbox routines, and you can even use a low level debugger like TMON or Macsbug.

A class library that implements an interface for you

The THINK Class Library is a collection of classes that implement the core of a standard Macintosh application. The THINK Class Library implements all the standard features of a Macintosh application and makes writing Macintosh applications easier.

What You Need

THINK Pascal works best when you have at least 1 megabyte (Mb) of RAM and a hard disk. With only 1Mb, you won't be able to take advantage of System 7.0, MultiFinder, the THINK Class Library, or MacApp.

Which Macintosh models?

You can run THINK Pascal on a Macintosh Classic, Plus, SE, Portable, the Macintosh SE series, or the Macintosh II series.

Which System and Finder?

Use the latest System and Finder provided by Apple. THINK Pascal requires at least System 6.0.5.

THINK Pascal is designed to work best under MultiFinder or System 7.0. If you're using a Macintosh with 1MB RAM, you're better off using the latest version of System 6.0 recommended for your machine with MultiFinder turned off.

How much RAM?

Under System 6, you can run THINK Pascal with 1MB of RAM. To use the THINK Class Library, you need 2MB. To use MacApp, you need 4MB.

Under System 7.0, you can run THINK Pascal with 2MB of RAM. To use the THINK Class Library or MacApp, you need 4MB.

How much disk space?

The basic THINK Pascal system takes up about 1200K on your disk, not including your own files. The size of the whole THINK Pascal system is 5.75 megabytes. The actual size of your system may be smaller, depending on the kinds of programs you work on.

What's in the Package

Your THINK Pascal package consists of four disks, this manual (*THINK Pascal User Manual*), and the *Object-Oriented Programming Manual*. In addition to THINK Pascal, the disks contain the

THINK Class Library, several programming examples, and some utilities, like the THINK Pascal profiler, ResEdit, SAREz, and SAdRez.

What's in the Manuals

The two manuals describe different parts of your THINK Pascal package. This manual, the *THINK Pascal User Manual*, tells you how to use THINK Pascal. The *Object-Oriented Programming Manual* describes Object Pascal in THINK Pascal and the THINK Class Library in detail.

Conventions in the manuals

Each chapter begins with an introduction that describes what's in the chapter, followed by a list of the major topics covered in the chapter. If you are interested in a specific topic, consult the index in the back of the manual.

The names of menus and commands are in **bold face**.

When a technical term or key word is introduced, it also appears in bold face.

Names of files, code fragments, resource names, function names, and variables appear in "typewriter face."

All numbers are decimal. Hexadecimal numbers are written in Pascal notation: \$3EFA.

In these manuals, the term **Toolbox routine** means any routine in ROM. The Macintosh ROM actually consists of two kinds of routines: Operating System routines and Toolbox routines. Operating System routines deal with low-level aspects of the machine like the file manager, the event posting mechanism, device management, etc. The Toolbox deals with high-level aspects like the drawing environment, the window mechanism, menus, dialogs, etc.

THINK Pascal User Manual

This manual is organized in six sections: Getting Started, Learning THINK Pascal, Using THINK Pascal, Reference, Utilities, and Appendices.

ONE Getting Started

This section contains this chapter and the installation instructions.

- 1 **Welcome** gives you an overview of THINK Pascal. This is the chapter you're reading
- 2 **Installing THINK Pascal** shows you how to install THINK Pascal. Even if you don't read manuals, be sure to read these instructions.

TWO Learning THINK Pascal

This section contains three tutorials.

- 3 **Tutorial:Bullseye** shows you how to write a program in THINK Pascal and how to use the basic debugging tools.

- 4 **Tutorial:ObjectDraw** shows you how to build a Macintosh application in THINK Pascal. It covers some of the more advanced topics like using resource files and libraries.
- 5 **Tutorial: Hex Dump DA** shows you how to build a desk accessory in THINK Pascal. It covers even more advanced topics like segmentation.

THREE Using THINK Pascal

This section contains ten chapters that describe the different components of THINK Pascal.

- 6 **Editing** describes the THINK Pascal editor which is specifically designed to help you write Pascal programs.
- 7 **Working With Projects** is about projects, the tool that keeps track of all your source files and object code. You'll learn how to work with source files, how to add them to your project, and how to segment your project.
- 8 **Running Programs** shows you how to run a program in THINK Pascal. You'll learn how to compile and link your program, and what to do when something goes wrong.
- 9 **Debugging Programs** introduces you to the basic set of debugging tools in THINK Pascal. These tools let you run your program line by line, set breakpoints anywhere in your program, and observe the values of your variables.
- 10 **Units and Libraries** teaches you how to create units and libraries. Units and libraries are collections of Pascal code that help you write modular programs.
- 11 **Using Predefined Routines** tells you how to use the procedures and functions built into THINK Pascal. It also tells you how to call the Macintosh Toolbox routines.
- 12 **Building Projects** shows you how to build the four kinds of programs you can write in THINK Pascal: applications, desk accessories, device drivers, and code resources.
- 13 **Assembly Language** gives you all the information you need to know to write assembly language routines that work with THINK Pascal.
- 14 **LightsBug** introduces you to LightsBug, THINK Pascal's powerful debugging tool. LightsBug lets you get a closer look at your program. You can examine and change the values of your variables (even arrays and records), look at any part of memory, and examine the heap.
- 15 **Compiler Directives** shows you how to use compiler directives — instructions to the compiler — to generate different code for different situations.

FOUR Reference

This section contains two reference chapters.

- 16 THINK Pascal Menus** describes the THINK Pascal menu commands.
- 17 Language Reference** describes the Pascal language implemented by THINK Pascal. This is a very long chapter, and you don't have to read all of it. The important sections are sections 9 and 10 which describe the built in input/output routines and the standard routines.

FIVE Utilities

This section describes how to use several utilities that come with THINK Pascal

- 18 Project Utilities** describes a utility that can back up or print the files in your project.
- 19 The Profiler** shows you how to use the THINK Pascal profiler to see how your program runs and how much time is spent in each routine
- 20 The Pascal Source Converter** helps translate programs written for Apple's MPW Pascal for use with THINK Pascal.
- 21 Resource Description Files** describes How to write text files that you compile with SAREz to produce resource files.
- 22 SAREz** creates resource files from resource description files.
- 23 SADeRez** creates resource description files from resource files
- 24 PostRez** converts a resource file for use with a MacApp program

SIX Appendices

This section contains four appendices.

- A What's New** describes the new features in THINK Pascal 4.0.
- B ANS Pascal Compatibility** describes THINK Pascal's compatibility with the ANS Pascal standard.
- C Porting to THINK Pascal** describes the limitations and non-standard features of THINK Pascal that you need to be aware of when you port programs from other Pascal compilers to THINK Pascal.
- D Error Messages** lists and describes all the error messages that THINK Pascal generates.

What You Should Know

This manual assumes you know how to use your Macintosh, and that you already know or are at least learning, how to program in Pascal. If you're just getting started in Pascal, THINK Pascal is a great platform.

If you're planning to write Macintosh applications, you should be familiar with the Macintosh Toolbox as described in *Inside Macintosh*. The Toolbox is the set of operating system and user interface routines that make a Macintosh a Macintosh. This manual won't show you how to write Macintosh applications or how to use the Macintosh Toolbox. There are several books that teach you how to build applications — see "Learning to write Macintosh programs" below. You can also look at the sample programs in your THINK Pascal package for examples of Macintosh applications.

One of the best ways to learn both Pascal and Macintosh programming at the same time is with Symantec's Just Enough Pascal, an on-line programming tutorial. Just Enough Pascal is a desk accessory that works with THINK Pascal to teach you Pascal, Macintosh programming, and THINK Pascal right at your Macintosh.

Learning Pascal

Pascal is the most popular language for learning how to program, so you'll find several books that teach programming in Pascal. Some of the Pascal books are written specifically for use with THINK Pascal or its cousin, Macintosh Pascal.

The standard reference for the Pascal programming language is the third edition of *Pascal User Manual and Report* (Springer-Verlag) by Kathleen Jensen and Niklaus Wirth, revised by Andrew Mickel and James Miner. The *User Manual and Report* is fairly technical, and it's designed for people who already know fundamental programming concepts.

Scott Kronick's *Macintosh Pascal Illustrated: The Fear and Loathing Guide* (Addison-Wesley) is an amusing and unorthodox introduction to programming the Macintosh in Pascal. His book covers most of the things you need to know to get started.

Oh! THINK's Lightspeed Pascal (W. W. Norton) by George Beekman and Michael Johnson is a companion to *Oh! Pascal* (W. W. Norton) by Doug Cooper and Michael Clancy. Together, these books are a good introduction to Pascal and THINK Pascal.

Pascal on the Macintosh: A Graphical Approach (Addison-Wesley) by David Niguidula and Andries Van Dam is for beginning Pascal programmers and uses THINK Pascal examples. Several universities use it as a first term programming text.

Another good book if you're learning Pascal is *Macintosh Pascal* (Houghton-Mifflin), by Robert Moll and Rachel Folsom. It is particularly helpful because of the many similarities between Macintosh Pascal and THINK Pascal.

Kurt Schmucker's *Object-Oriented Programming for the Macintosh* (Hayden) contains a great deal of information about object-oriented programming in general and some details about using Object Pascal.

Object-Oriented Programming Power for THINK Pascal Programmers (Microsoft Press) also teaches you how to use Object Pascal and is especially for THINK Pascal programmers. It also teaches you the fundamentals of application frameworks, like the THINK Class Library and MacApp.

Learning to write Macintosh programs

If you're new to programming the Macintosh, you might find yourself overwhelmed by the complexity of the Macintosh Toolbox and unfamiliar programming techniques. When the Macintosh was introduced in 1984, there was very little technical information available to casual programmers, and even commercial developers had a hard time figuring out how to get things to work correctly.

The Macintosh is even more complex today than it was in 1984, but now there are more places you can go for information. Several good books introduce programming the Macintosh and teach some of the finer points of using the Macintosh Toolbox. No matter which books you choose to get started, *Inside Macintosh* is indispensable.

Inside Macintosh Volumes I-VI (Addison-Wesley) is the official reference that describes the more than 1,000 Macintosh Toolbox routines. You might be able to get by without it for a while, but if you're planning to write serious applications, you just can't do without it. At six volumes, it represents a hefty investment. The first three volumes cover the fundamentals. Volumes IV and V cover the additions and changes made with the introduction of the Macintosh Plus, SE, and II. Volume VI covers the changes introduced with System 7.

In addition to *Inside Macintosh*, Apple also publishes these books through Addison-Wesley:

- *Human Interface Guidelines*
- *Technical Introduction to the Macintosh Family*
- *Programmers Introduction to the Macintosh Family*
- *Guide to the Macintosh Family Hardware, Second Edition*
- *Apple Numerics Manual, Second Edition*
- *LaserWriter Reference*
- *Inside AppleTalk*
- *Designing Cards and Drivers for the Macintosh Family, Second Edition*

You won't need all these books when you get started. Some of the books, like *Human Interface Guidelines*, are useful for all Macintosh programmers. Other books, like *Inside AppleTalk*, are meant for programmers working on specific kinds of applications. These books are available from APDA, technical bookstores and computer stores, and in some general bookstores.

THINK Reference, by Symantec, is a hypertext program that gives you instant access to the critical system information that you need to program the Macintosh. It describes nearly all of the Macintosh Toolbox routines from *Inside Macintosh I-V*, including information from the technical notes, code

examples, and Symantec programmer tips. The information is cross-referenced, so you can look up related Toolbox routines quickly. It includes customizable bookmarks and templates that let you copy Toolbox calls right into your own program.

The Macintosh Pascal Programming Primer: Inside the Toolbox Using THINK Pascal, Volume 1, by Dave Mark is a good introduction to Macintosh programming for those already familiar with Pascal. It explains how to use the Toolbox, handle resources, and write a Macintosh application. Also, the examples use some of the newer parts of the Macintosh system, such as the Notification Manager and HyperCard XCMDs and XFCNs.

Stephen Chernicoff's four volume set, *Macintosh Revealed* (Hayden Books), is another step-by-step introduction to Macintosh programming. Chernicoff shows you how to build a working application and points out the parts of *Inside Macintosh* you really need to know as opposed to the parts you just need to be aware of. The programs in the books are written in MPW Pascal, but they're not too difficult to translate to THINK Pascal or to THINK C.

Scott Knaster is the author of two books on Macintosh programming. The first, *How to Write Macintosh Software* (Hayden Books), teaches you what goes on inside the Toolbox. This book contains some valuable tips about debugging Macintosh programs. The second book, *Macintosh Programming Secrets* (Addison-Wesley), deals with some of the conventions and techniques that have become standard in Macintosh programs. It also contains information about the Macintosh II and the Macintosh SE. These books are more technical than *Macintosh Revealed* and are loaded with pictures, diagrams, and examples.

If you frequently use ResEdit to create and edit resource files, you may want the *ResEdit 2.1 Reference* (Addison-Wesley) by Apple Computer, Inc. It explains how to edit standard resource types and how to extend ResEdit by adding your own resource pickers and editors.

Finally, *MacTutor* is the leading technical journal for Macintosh programming. The articles range from tutorial examples to advanced techniques. *MacTutor* covers several languages, not just C and Pascal, and most of the examples are written in THINK C and THINK Pascal. (All of the programs described in the magazine are available on disk.)

Apple Computer, Inc.

Apple Computer is naturally one of the best places for information about Macintosh programming. Apple administers the Apple Partner and Apple Associate program for commercial and non-commercial software developers. For more information, contact Apple:

Apple Computer, Inc.
20525 Mariani Avenue, MS 75-2C
Cupertino, CA 95014
(408) 974-4897

Apple Programmer's and Developer's Association (APDA)

The Apple Programmer's and Developer's Association (APDA) is Apple's in-house membership organization that distributes technical information to programmers and developers. APDA is a great

source for Technical Notes, programming utilities, reference books, and information about announced (but unreleased) products. For information about membership and products, contact APDA directly:

Apple Programmer's and Developers Association (APDA)
Apple Computer, Inc.
20525 Mariani Avenue, MS 33G
Cupertino, CA95014-6299
(800) 282-2732 (USA)
(800) 637-0029 (Canada)
(408) 562-3910 (Other)
(408) 562-3971 (Fax)

CompuServe

Symantec has a forum on CompuServe specifically for its customers. Simply type GO SYMANTEC at any ! prompt. You'll find discussions here about programming in general and THINK C and THINK Pascal in particular. The data libraries contain utilities as well as sources for many programs.

CompuServe also has an Apple developers forum. Just type GO MACDEV at any ! prompt. This forum is a good place to get in touch with the Macintosh programming community.

Symantec Programming Languages Association (SPLash)

The Symantec Programming Languages Association (SPLash) is a user group for anyone who uses the Symantec programming languages THINK C and THINK Pascal. SPLash is an independent organization endorsed by, but not affiliated with, Symantec Corporation.

SPLash offers its members the following:

- *THINKin' CaP*, a 100-page quarterly that includes technical articles with source code, help for beginners, tips from Symantec Technical Support, insights on the THINK Class Library, and reviews of the latest tools and conferences.
- A quarterly disk containing all the source code from THINKin' CaP, programming utilities, and the latest patches for THINK C and THINK Pascal
- Meetings at major Macintosh conferences, including seminars on topics of interest and presentations from Symantec and SPLash members.
- A SPLash forum on America Online.

Yearly membership is \$30 for USA residents, \$40 for Canada and Mexico residents, and \$60 for residents of other countries. For more information, write to this address:

SPLash Resources
1678 Shattuck Ave., #302
Berkeley, CA94709

Installing THINK Pascal

2

Introduction

This chapter tells you how to install THINK Pascal on your Macintosh.

Before you start...

Before you install THINK Pascal 4.0, you'll want to take care of these steps:

- If there is a file named `READ ME` on the disk `THINK Pascal 1`, read it. It contains information that didn't make it into the THINK Pascal manuals. It's a text file that you can read with any word processor, including Teach Text.
- Make a copy of your THINK Pascal 4.0 disks. If something goes wrong during the installation, you'll be able to make another copy and continue.
- Fill out and send in your registration card. You'll find it in the Customer Service Plan envelope. If you want technical support, information about upgrades, or news about special promotions, you must become a registered user.
- Make sure you have at least 5.75 megabytes (5.75MB) of disk space free. If you don't have enough room, either clean up your hard disk, or read "What's in the Archives" later in this chapter and install only the parts of THINK Pascal 4.0 that you need.
- Some anti-viral software, like Symantec AntiVirus for the Macintosh (SAM), may warn you when the archive installs an application. If this happens, let the archive continue its operation. (In SAM, click Allow) You may want to turn off your anti-viral software before you install THINK Pascal 4.0.

Topics covered in this chapter

- Summary
- Before you start...
- Instructions
- What's in the archives
- Disk layout diagram
- About the self-extracting archives

Summary

This chapter tells you how to set up THINK Pascal on your hard disk. This setup ensures that THINK Pascal will know where to find all the files it needs to compile your programs. To learn more about why the files are organized this way, see Chapter 9, "Files & Folders."

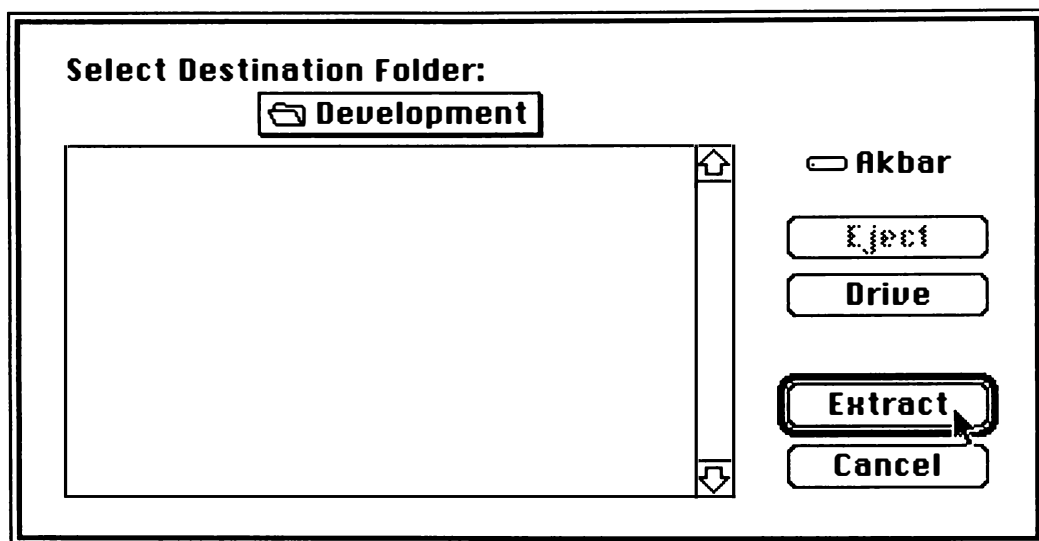
Your THINK Pascal 4.0 disks contain a total of seven self-extracting archives, applications that contain compressed files which they decompress and install on your hard disk. You'll create a folder on your hard drive and then run each self-extracting archive and let them install their files in

that folder. Then you'll move the THINK Pascal 4.0 application into the newly created THINK Pascal 4.0 Folder. If you want to install only some parts of THINK Pascal 4.0, read "What's in the Archives" later in this chapter to find out which archives you need to run.

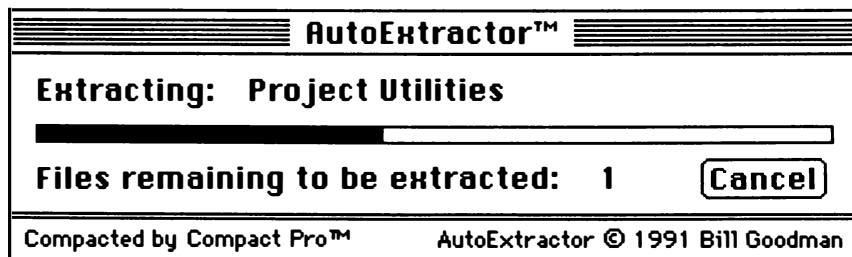
Instructions

Here's how to install THINK Pascal 4.0:

1. In the Finder, create a new folder and name it Development.
2. Insert the disk THINK Pascal 1, and double-click on THINK Pascal Utilities.sea.
3. A standard file dialog, like the one below, appears. Move to your Development folder and click Extract.



4. The archive decompresses its files and places them on your hard disk. It displays its progress in the dialog below. The archive quits when it's done.

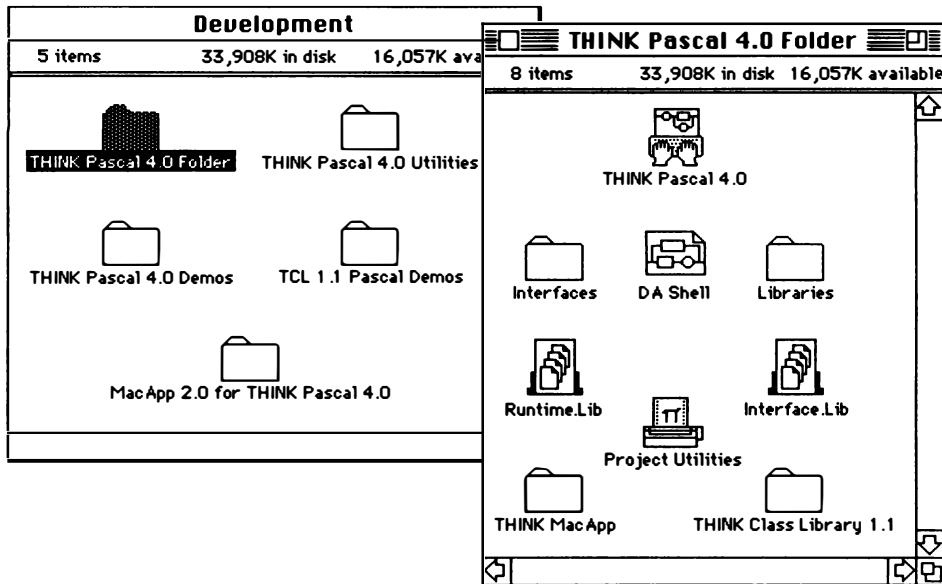


5. Repeat steps 2–4 for each of the other archives on THINK Pascal 2, THINK Pascal 3, and THINK Pascal 4.
6. Insert THINK Pascal 1, and move THINK Pascal 4.0 into the THINK Pascal 4.0 Folder in your Development folder.

When you're done, the Development folder contains these folders, as shown below:

THINK Pascal 4.0 Folder
THINK Pascal 4.0 Demos
MacApp 2.0 for THINK Pascal 4.0

THINK Pascal 4.0 Utilities
TCL 1.1 Demos



Note: The icons in your folders won't be in the same places as the icons in this illustration. You can arrange them anyway you want inside the folder.

What's in the Archives

You don't need to install everything included in your THINK Pascal 4.0 package. This table describes which archives you should run to get what you need.

If you want the...	Run these archives...
Basic THINK Pascal system	Interfaces & Libs.sea on THINK Pascal 2, and copy THINK Pascal 4.0 into your THINK Pascal 4.0 Folder from THINK Pascal 1.
THINK Class Library (Even if you've used the THINK Class Library before, you'll need the demos for examples of the new features.)	THINK Class Library 1.1.sea on THINK Pascal 3 and TCL 1.1 Pascal Demos.sea on THINK Pascal 2.
Tutorials and example programs	THINK Pascal 4.0 Demos.sea on THINK Pascal 2.
Resource utilities, like ResEdit and SAREz	Resource Utilities.sea on THINK Pascal 4.
Other utility programs, like Project Utilities.	THINK Pascal Utilities.sea on THINK Pascal 1.
MacApp 2.0 support	MacApp 2.0 for THINK Pascal.sea on THINK Pascal 3.

The rest of this section describes what's in each archive.

Interfaces & Libs.sea

This archive places these folders in the THINK Pascal 4.0 Folder:

- DA Shell
- Interface.Lib
- Runtime.Lib
- Interfaces folder
- Libraries folder, which includes SANELib.lib, SANELib881.lib, µRuntime.Lib, TCLRuntime.Lib, DRVRRuntime.Lib, and RSRCRuntime.Lib.

THINK Pascal 4.0 Demos.sea

This archive places these folders in the THINK Pascal 4.0 Demos folder:

- Hex Dump Folder
- Hypercard Demos folder
- LearnOOP folder
- ObjectDraw f folder

THINK Class Library 1.1.sea

This archive places the THINK Class Library in the THINK Class Library 1.1 folder.

TCL 1.1 Pascal Demos.sea

This archive places these folders in the TCL 1.1 Pascal Demos folder:

- New Class Demo folder
- Starter folder
- TinyEdit folder
- Art Class Folder

Resource Utilities.sea

This archive places these files in the THINK Pascal 4.0 Utilities folder:

- ResEdit 2.1 folder
- Rez Utilities folder, which includes SAREz, SAdRez, and SAPostRez.

THINK Pascal Utilities.sea

This archive places this application in the THINK Pascal 4.0 Folder:

- Project Utilities

It also places these files in the THINK Pascal 4.0 Utilities folder:

- AppleEdit 2.0, a text editor desk accessory
- Block Comment FKEY f folder

MacApp 2.0 for THINK Pascal.sea

This archive places these files in the MacApp 2.0 for THINK Pascal 4.0 folder:

- Pascal Source Converter
- Generic.Script
- MacApp.Script
- MacApp Seeds folder
- Samples from MacApp folder

It also places the THINK MacApp folder in the THINK Pascal 4.0 Folder, with these files:

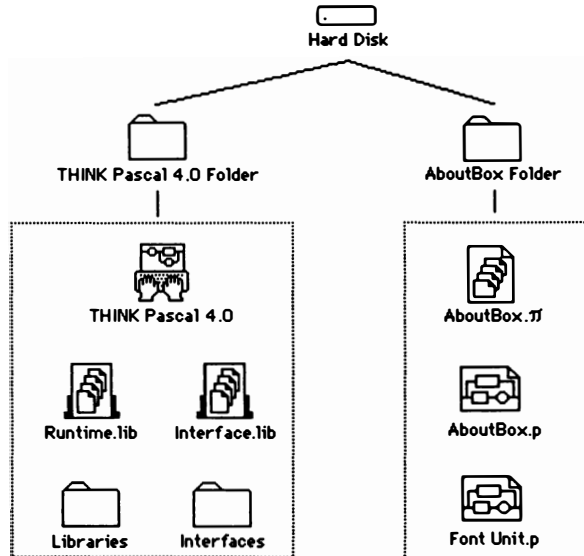
- Diffs folder
- MacApp Libraries folder
- Templates folder
- Settings.R
- Settings.Debug.R

Disk Layout Diagram

If you move or rename the folders containing your THINK Pascal files, THINK Pascal can still find your files. But you must follow these guidelines:

- Keep all your THINK Pascal libraries and interfaces in the same (or a nested) folder as the THINK Pascal application.
- Keep all your source files and other project-specific files in the same (or a nested) folder as your project document.

This diagram shows the recommended disk layout. (AboutBox is the name of a program you might be working on.)



About the Self-Extracting Archives

The self-extracting archives included with THINK Pascal 4.0 are created with the shareware program Compact Pro. You can open a self-extracting archive with Compact Pro to examine its contents or install files one by one. You can download Compact Pro from most on-line services, including CompuServe, or write the author for ordering information:

Bill Goodman
 109 Davis Avenue
 Brookline, MA 02146
 USA

THINK PascalTM

P A R T T W O

Learning THINK Pascal

- 3 Tutorial: Bullseye
- 4 Tutorial: ObjectDraw
- 5 Tutorial: Hex Dump DA

Tutorial: Bullseye 3

Introduction

This chapter shows you how to write a program with THINK Pascal. The program, called Bullseye, draws a series of concentric circles in the built-in Drawing window. This chapter shows you how to create a THINK Pascal project, how to write a Pascal source file, and how to use some of the more common debugging tools in THINK Pascal.

Note: The pictures for this tutorial were made under System 6. The tutorial works under System 7.0, but some dialog boxes will look slightly different from those in the pictures.

Before you begin

Be sure you followed the instructions in Chapter 2, "Installing THINK Pascal," to install THINK Pascal on your disk.

What you should know

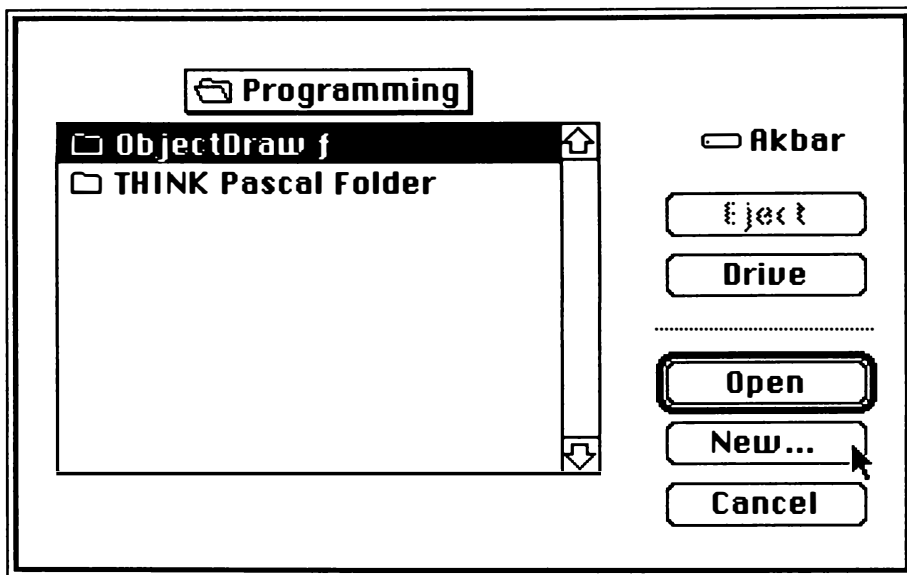
You should know how to use the standard file dialog boxes to move around to different folders. If you don't know how to do this, read the *Macintosh System Software User's Guide* that came with your Macintosh.

Topics covered in this chapter:

- Creating the project
- Creating a source file
- Adding the source file to the project
- Running the program
- Debugging the program
- Stepping through the program
- Where to go next

Creating the Project

To start creating the Bullseye project, double-click on the THINK Pascal icon. You'll see a dialog box that asks you to open a project:



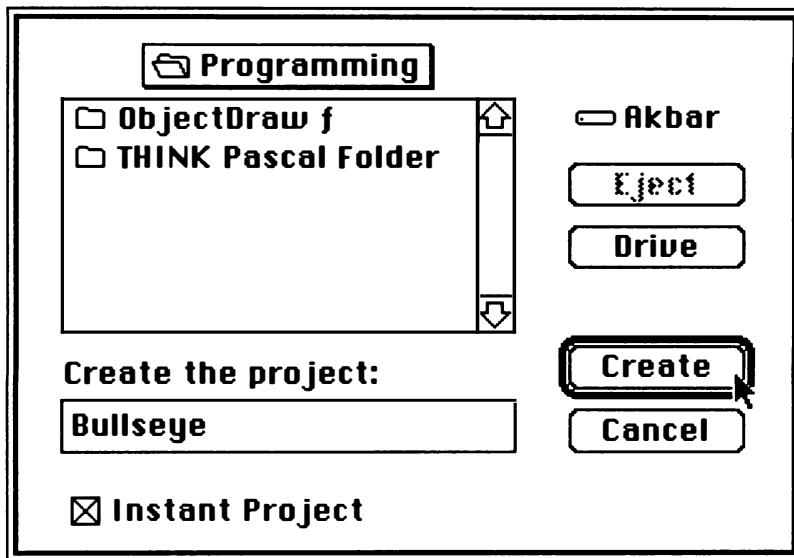
Since you're creating a new project, click on the New button.

You'll see another dialog box, one that lets you create projects. This is the same dialog you see when you select **New Project...** from within THINK Pascal.

Move outside the THINK Pascal Folder.

Note: It's very important to move outside the THINK Pascal Folder.

Check the Instant Project box The Instant Project option helps you create small programs by automating many steps. Name the project Bullseye, and click on the Create button.



THINK Pascal creates these things:

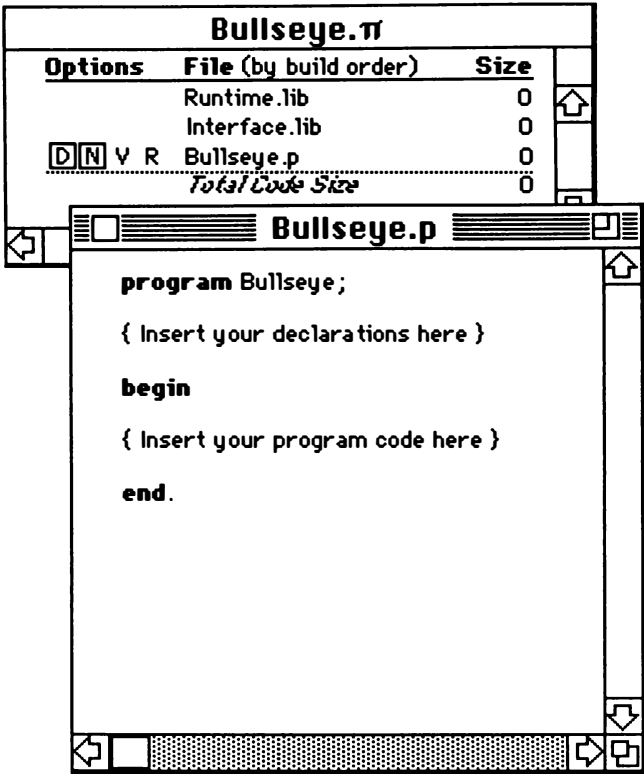
- A source file (Bullseye.p)
- A project file (Bullseye.π) that contains the source file
- A folder (Bullseye f) to hold the project file and the source file.

Note: If you don't use the Instant Project option, THINK Pascal creates only the project file. You need to create and add your own source file and create your own folder to hold them.

The Instant Project option always creates a source file that ends in .p, a project document that ends in .π, and a folder that ends in f. It uses what you entered in the **New Project...** dialog as the root for these names. If you entered something ending in .π, it strips that off before creating the names.

Note: You may want to follow this naming convention for your own folders and project documents. To type a *f*, press Option-f. To type a *π*, press Option-p.

THINK Pascal displays the project document and source file. If you want, you can resize them so they look like this:



The source file contains an outline for a program. Its comments show you where to insert your declarations and program code. Notice that the name of the program is the what you entered in the **New Project...** dialog.

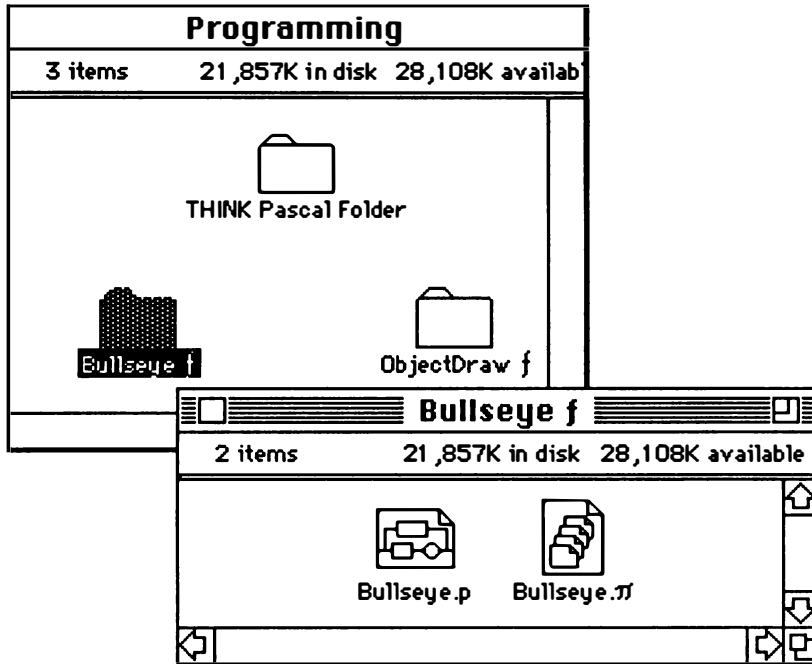
The project window contains three columns. The **Options** column lets you control some of the compiler options. You'll see how these work later in this chapter. The **File** column shows you the name of each file. The **Size** column shows you the size of the code of each file and library in bytes.

Note: The notation “by build order” means that the project window is showing you the files in the order they’ll be compiled. You’ll learn more about this in Chapter 5, “Tutorial: Hex Dump DA.”.

THINK Pascal automatically inserts two standard libraries into your project, in addition to your source file. `Runtime.lib` contains the code for all the standard Pascal routines (like `writeln`). `Interface.lib` contains the glue code for all the Macintosh Toolbox routines marked `[Not in ROM]`.

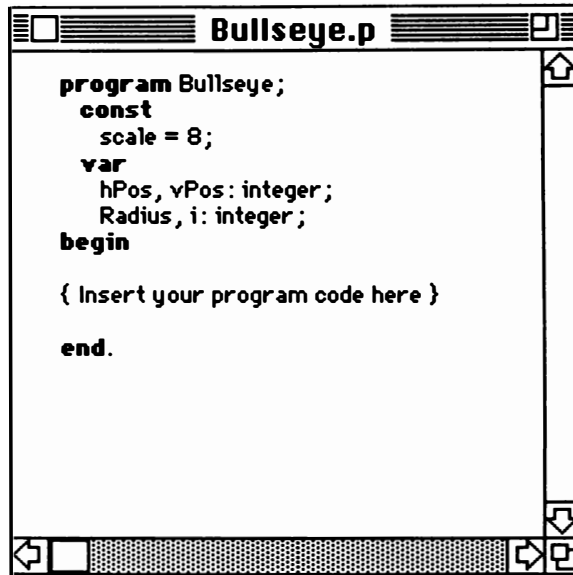
You'll use these two libraries in most of the programs you write. To learn more about libraries, read Chapter 10, "Units and Libraries," and Chapter 11, "Using Predefined Routines."

If you're running under MultiFinder, you can go to the Finder and look at Bullseye f, the folder THINK Pascal created. It holds both Bullseye.p and Bullseye.π.



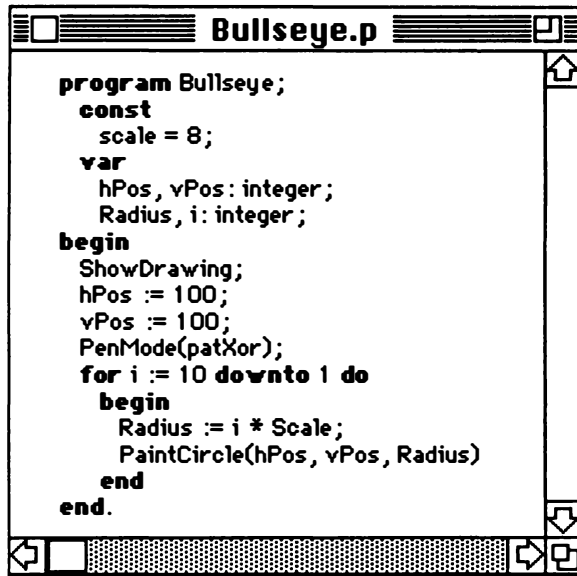
Writing a Source File

Now you're ready to create your source file. Triple-click on the line { Insert your declarations here } to select it. Type in the declarations shown below (between the lines **program** Bullseye and **begin**).



Note: Make sure you type in the program exactly as it appears above. Remember, in Pascal punctuation is very important.

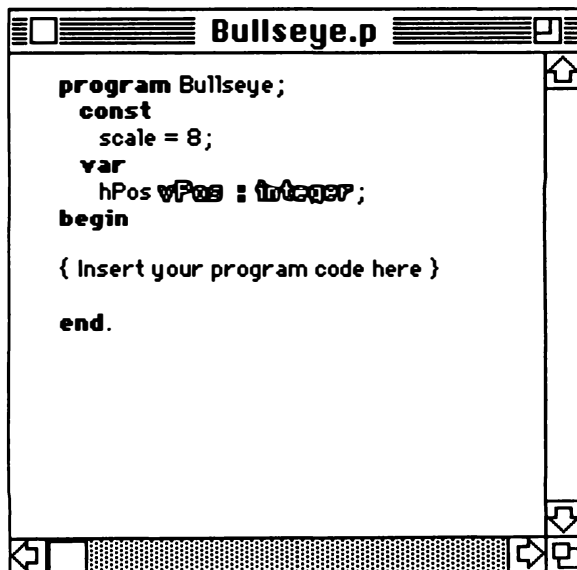
Now, triple-click on the line `{ Insert your program code here }` and type the program shown below (between the first **begin** statement and the last **end** statement.)



```

program Bullseye;
  const
    scale = 8;
  var
    hPos, vPos: integer;
    Radius, i: integer;
  begin
    ShowDrawing;
    hPos := 100;
    vPos := 100;
    PenMode(patXor);
    for i := 10 downto 1 do
      begin
        Radius := i * Scale;
        PaintCircle(hPos, vPos, Radius)
      end
    end.
  
```

As you type, you'll notice that the THINK Pascal editor formats (pretty-prints) your program for you. If you make a syntax error, the editor points it out by outlining the error text. In the example below, you forgot to type a comma between `hPos` and `vPos`:



```

program Bullseye;
  const
    scale = 8;
  var
    hPos vPos : integer;
  begin

    { Insert your program code here }

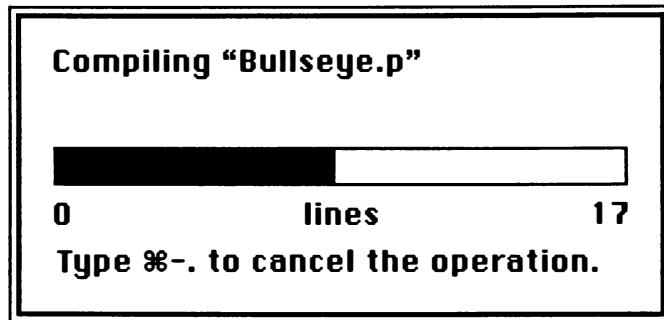
  end.
  
```

Aside from the special pretty-printing feature, the THINK Pascal editor works much like other text editors on the Macintosh. You can drag to select a range of text or double-click to select words. If you have a keyboard with arrow keys, you can use them to move around the file. To learn more about the THINK Pascal editor, see Chapter 6, "Editing."

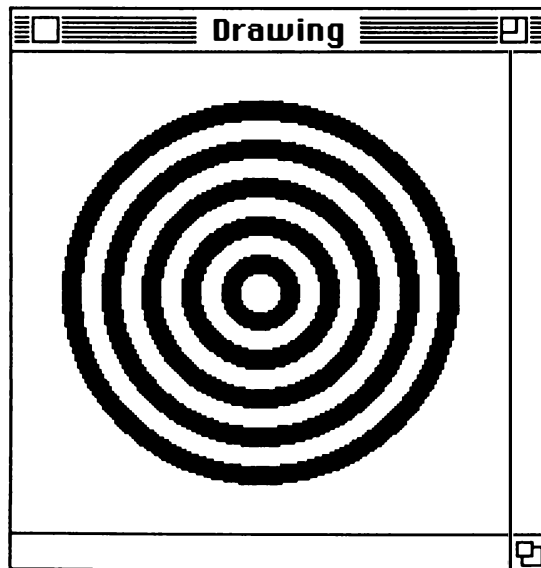
Running the Program

Now you're ready to run your program. Choose **Go** from the **Run** menu.

Because it's an integrated environment, THINK Pascal knows that it needs to load the libraries and that it needs to compile your source file. You'll see a dialog box that lets you know when THINK Pascal is loading or compiling.



When your program runs, it will open the drawing window and draw the bullseye design:



If you look at the project window now, you'll see that the Size column now contains the sizes of the libraries and of your compiled code.

Debugging the Program

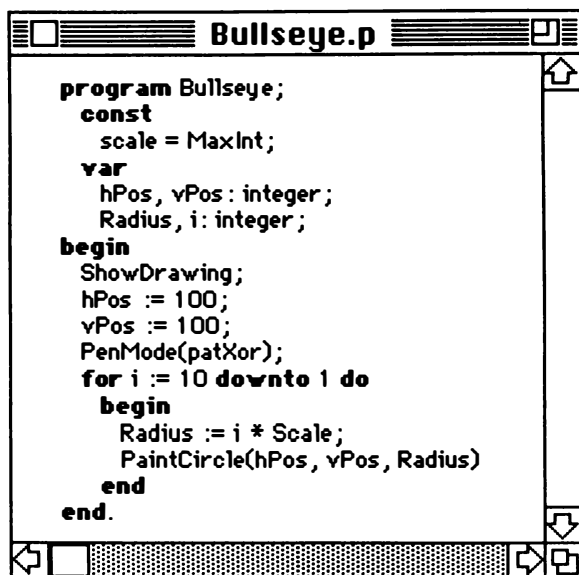
This program doesn't have any bugs in it, so to debug it, you'll have to introduce a bug. The bug will make THINK Pascal multiply two numbers so the result exceeds the legal range for integers.

First, click on the project window to make it the active window. Next, click on the V and the R next to the `bullseye.p` entry. Now both letters should be outlined with a box:

Options	File (by build order)	Size
	Runtime.lib	17884
	Interface.lib	10106
D N V R	Bullseye.p	160
<i>Total Code Size</i>		28150

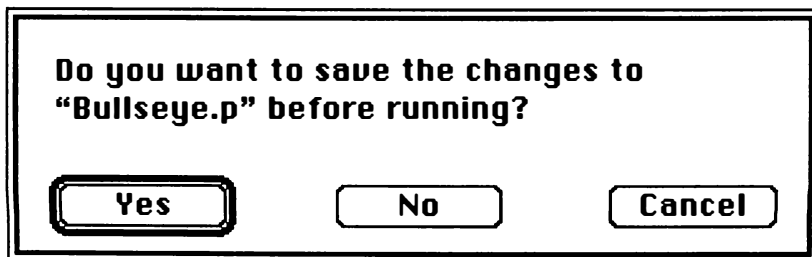
When you click on these two letters, you tell THINK Pascal to generate additional code to check for arithmetic overflow and for range checking. Range checking means that THINK Pascal checks to make sure that indices to arrays stay within the bounds of the array, and that values assigned to subrange types stay within subrange.

Now, change the program so the constant `scale` is equal to `MaxInt`. `MaxInt` is a built-in constant that's equal to the largest allowed integer (32767).



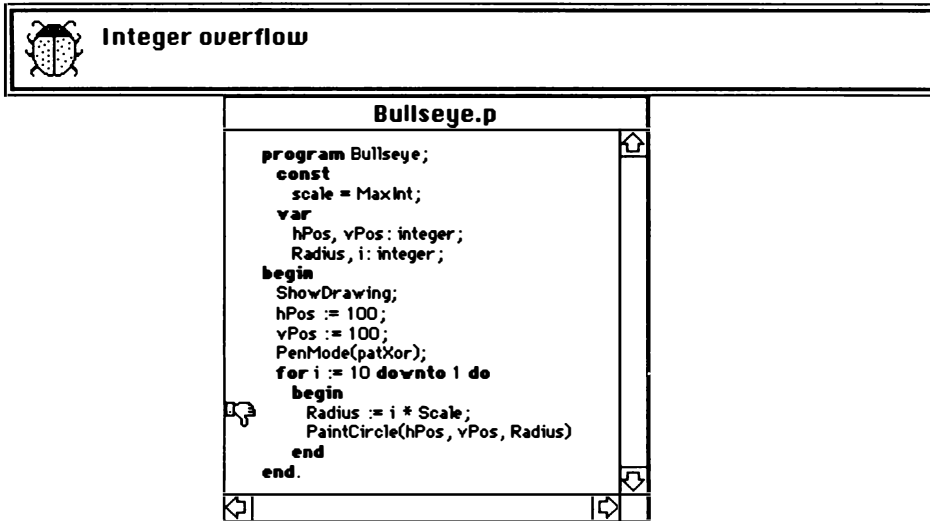
Make sure that the **Confirm Saves** option is on. This option checks whether you changed any of your source files before running your program. If it finds a file you changed, it asks you if you want to save them. Select the **Run** menu and see if there's a checkmark beside **Confirm Saves**. If there isn't, choose **Confirm Saves**.

Now choose **Go** from the **Run** menu. THINK Pascal recompiles the program, and then displays this dialog box:



THINK Pascal noticed that you had not saved the file. You can run the changed program without saving the file. In this case we know there's a bug, so we don't want to save the file. Click on the **No** button.

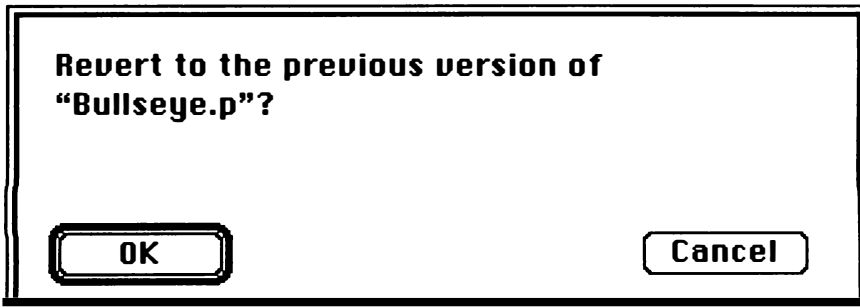
Immediately after the program begins, you'll see a bug dialog:



THINK Pascal puts a "thumbs down" next to the offending line. In this case, it's telling you that the multiplication `i * Scale` was out of the range for integers.

To dismiss the bug box, click anywhere or press the Return or Enter key.

The easiest way to undo the change you made to your file is to go back to the last saved version of the file. Choose the **R**evert command from the **F**ile menu. You'll see this dialog box:



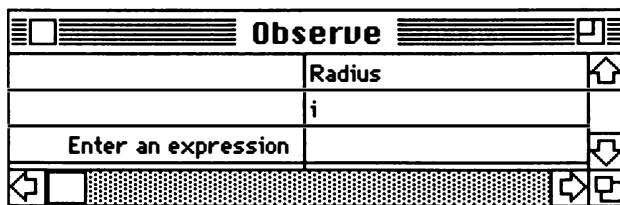
Since you do want to go back to the last saved version, click on the OK button. You'll be ready to learn how to use THINK Pascal's debugging tools to step through your program.

Stepping Through the Program

THINK Pascal lets you step through your program one line at a time so you can watch what it's doing. You can also watch the values that variables take on to make sure your program is behaving properly.

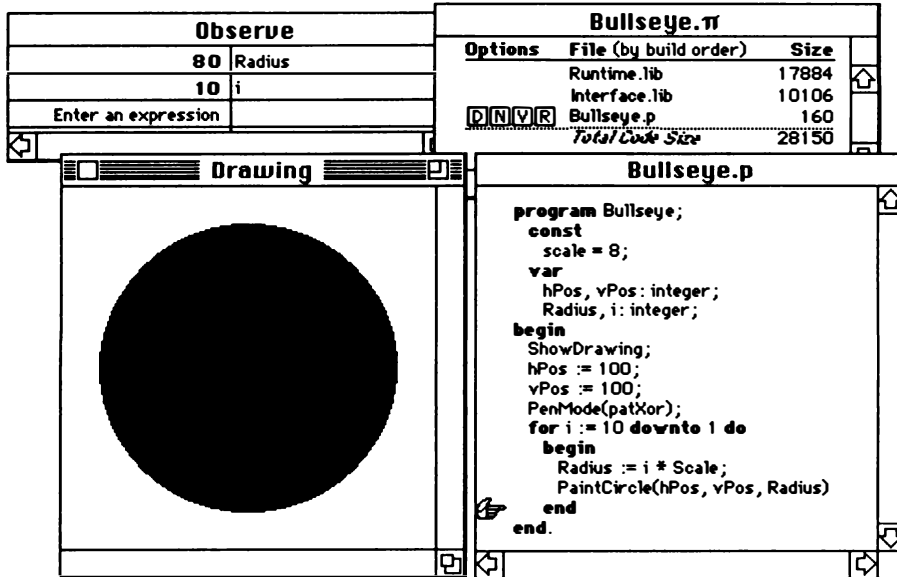
First, make the edit window about half as wide as it is now so it doesn't obscure the Drawing window. Next, choose **Observe** from the **Debug** menu. The Observe window appears.

Type in the word **Radius**, then press Return. Type in the letter **i**, then press Return. These are the names of the variables you'll be observing as you step through the program.



Drag the Observe window so it's above the Drawing window. It's OK if it overlaps the project window.

Now choose **Step Into** from the **Run** menu. THINK Pascal draws an execution finger next to the first line of your program. Keep choosing **Step Into** — better yet, use the Command-I equivalent — and watch what happens. The execution finger points to each line of the program. As you step, watch the values in the Observe window when the execution finger enters the loop.



Every time your program stops, THINK Pascal updates the values in the Observe window.

If you get tired of stepping, THINK Pascal can step automatically for you. Hold down the Option key and select the **Run** menu. The **Step Into** command is now the **Step-Step** command. Choose it. When you use this command, THINK Pascal runs your program in slow motion, as if you kept on choosing **Step Into** yourself. To pause your program, click on the Bug Spray icon in the upper-right hand corner. To start it up from where you paused, choose **Step-Step** again.

Where to Go Next

The tutorial in the next chapter is a more elaborate example. It shows you how to build a double-clickable Macintosh application, and some of the more advanced features of THINK Pascal.

If you would rather explore on your own, read the chapters of the "Using THINK Pascal" section that interest you. They're designed to be read in order, but you can skip around if you like.

Tutorial: ObjectDraw 4

Introduction

This tutorial shows you how to put together a Macintosh application in THINK Pascal. The application you'll build is called ObjectDraw. It's a simple drawing program that lets you draw several kinds of shapes in a window. ObjectDraw lets you print the window to any printer, and it lets you save your picture as a PICT format file that you can edit with drawing programs. It can also display any file saved in the PICT format, but it doesn't let you edit the picture.

ObjectDraw shows you how to use resource files with your THINK Pascal programs and how to add some of the special libraries to your project.

ObjectDraw uses Object Pascal, but this tutorial won't show you how to use Object Pascal. For more information, see the *Object-Oriented Programming Manual*, especially Chapter 4, "Object Pascal," and Chapter 5, "Tutorial: LearnOOP."

Note: The pictures for this tutorial were made under System 6. The tutorial works under System 7, but some dialog boxes will look slightly different from those in the pictures.

Before you begin

Make sure that you followed the instructions in Chapter 2, "Installing THINK Pascal," for installing THINK Pascal on your Macintosh. For this tutorial, you need to make sure these files and folders are installed:

- The Libraries and Interfaces folders in the THINK Pascal 4.0 Folder. They're from the Interfaces & Libs.sea archive
- The ObjectDraw f folder in the THINK Pascal 4.0 Demos folder. This folder contains all the source files and resource files that you'll use in this tutorial. It's from the THINK Pascal 4.0 Demos.sea archive.

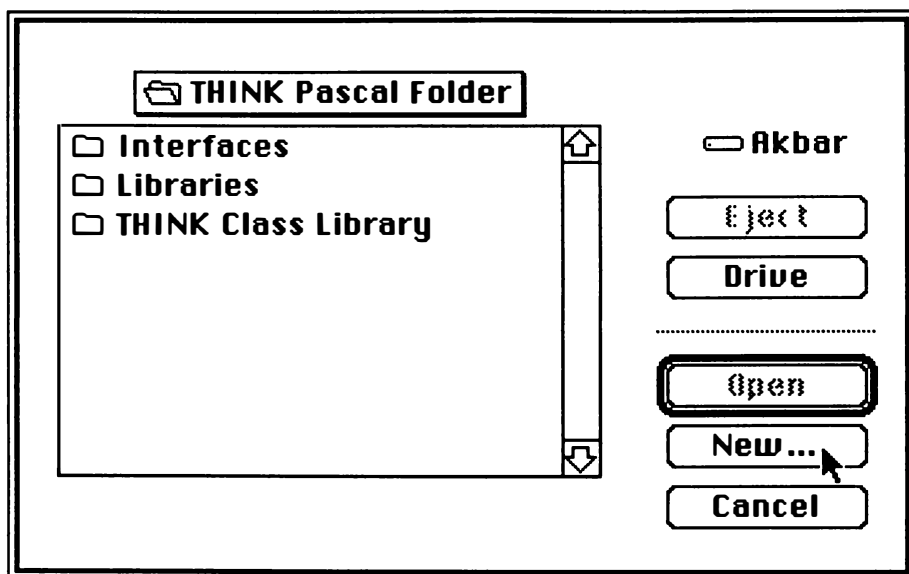
Topics covered in this chapter

- Creating the project
- Adding the libraries
- Adding the source files
- Setting the Run Options
- Setting the Compile Options
- Running the project
- Building the application
- Where to go next

Creating the Project

Since the ObjectDraw f folder is already on your disk, you don't need to create a new folder for the ObjectDraw project. But you do need to remove the project file, named ObjectDraw.π, that's in ObjectDraw f. That project lets you use ObjectDraw without completing this tutorial. Move it to another folder or back it up to another disk.

All you need to do now is double-click on the THINK Pascal icon. THINK Pascal displays this dialog when you launch it from the Finder.



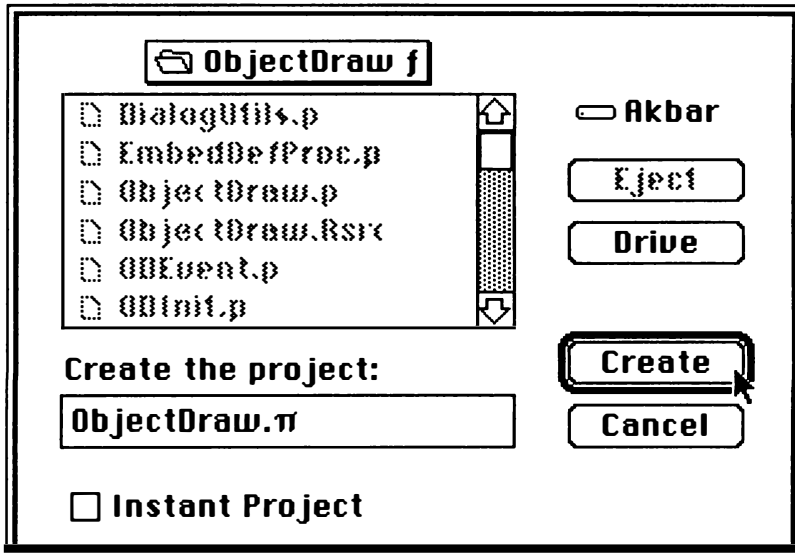
You're creating a new project, so click on the New button.

THINK Pascal displays a standard file dialog that lets you create projects.

Move to ObjectDraw f in the THINK Pascal 4.0 Demos folder..

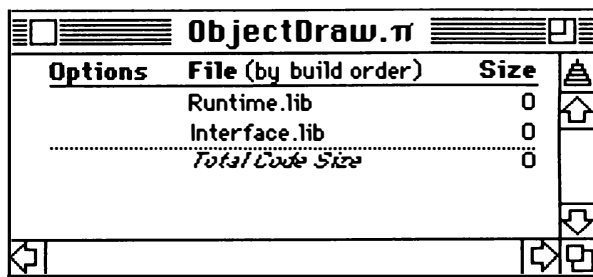
Note: It's very important to move to the ObjectDraw f folder.

Make sure the Instant Project option is off. Name the new project `ObjectDraw.π`, and click on the Create button. (To make a “π” hold down the Option key as you press the letter “p”. By convention, all THINK Pascal projects end in `.π`.)



Note: To learn more about the Instant Project option, see Chapter 3, “Tutorial: Bullseye.”

THINK Pascal creates a new project document on the disk and displays a project window. Your project window should look like this:



Next, you'll add the libraries to your project, and then you'll add the source files.

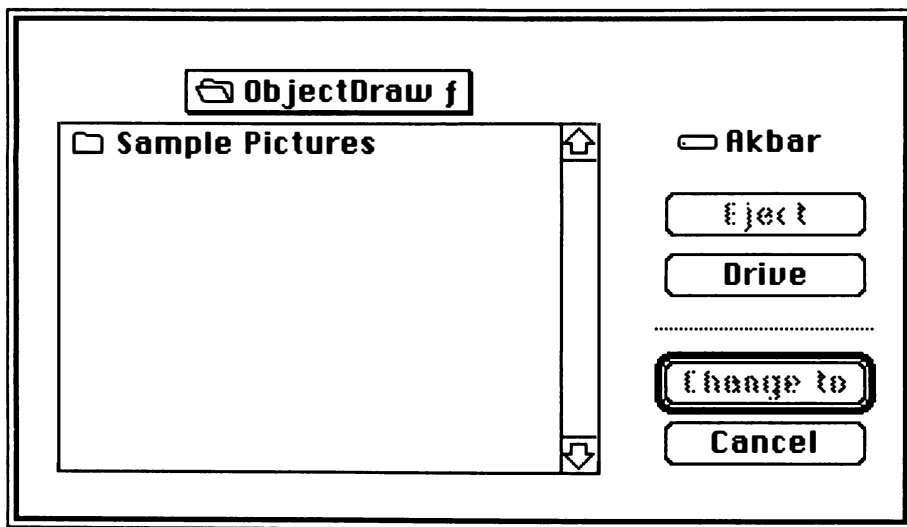
Adding the Libraries

THINK Pascal automatically inserts the names of two standard libraries into your project. `Runtime.lib` contains the code for all the standard Pascal routines (like `writeln`).

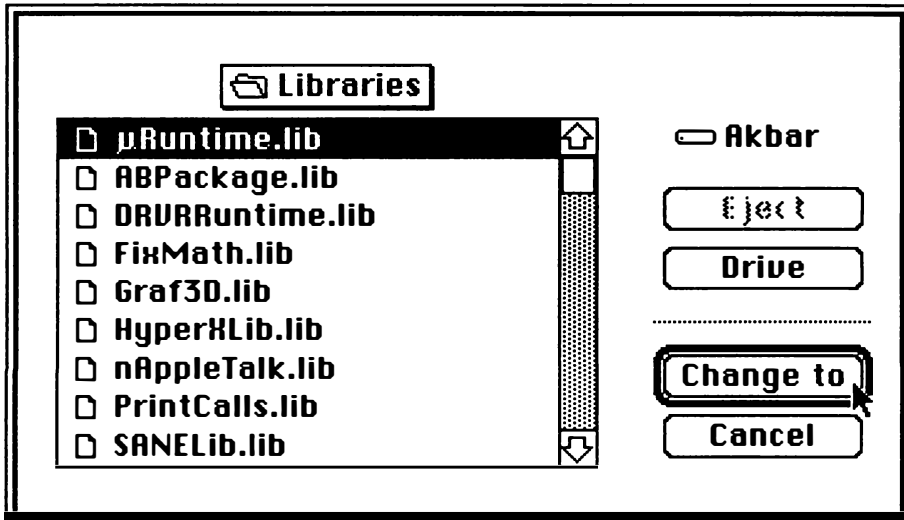
Interface.lib contains the glue code for all the Macintosh Toolbox routines marked [Not in ROM].

ObjectDraw is a true Macintosh application, so it doesn't use any of the Pascal input/output routines in Runtime.lib. If you leave the Runtime.lib library in your project, your program will run perfectly, but there will be code in your project that ObjectDraw never uses, and your project file will be bigger than it really needs to be. But ObjectDraw still needs some of the routines in Runtime.lib. Even if a program doesn't explicitly call any of the routines in it, it contains code to handle 32-bit multiplication, set operations, and other things that your program may need.

THINK Pascal has a smaller version of Runtime.lib called μ Runtime.lib which doesn't include the Pascal standard input/output code. To replace Runtime.lib with μ Runtime.lib, hold down the Option key as you double-click on Runtime.lib in the project window. THINK Pascal displays this dialog:



`μRuntime.lib` is in the Libraries folder, so move to that folder, select `μRuntime.lib`, and click on the Change To button.

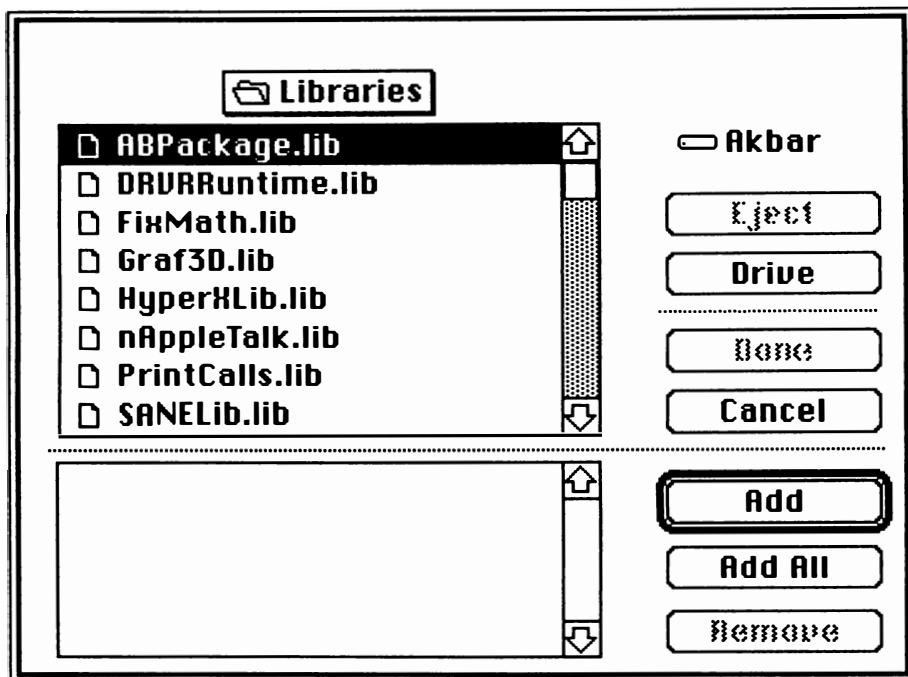


When you use `μRuntime.lib`, you can't use the THINK Pascal input/output routines like `reset`, `rewrite`, `writeln`, `readln`, etc. If you plan to write Macintosh applications exclusively, it's a good idea to use `μRuntime.lib` instead of `Runtime.lib` to make your projects smaller.

Note: Whether you use `Runtime.lib` or `μRuntime.lib`, your finished application will be the same size because THINK Pascal uses smart linking to remove references to unused code in your final application. The difference is only in the project size.

Adding the Source Files

Now you'll add the source files to your project. Hold the Option key and select the **Add Files...** command from the **Projects** menu. To see the command, hold down the Option key as you select the **Project** menu. You see this dialog:

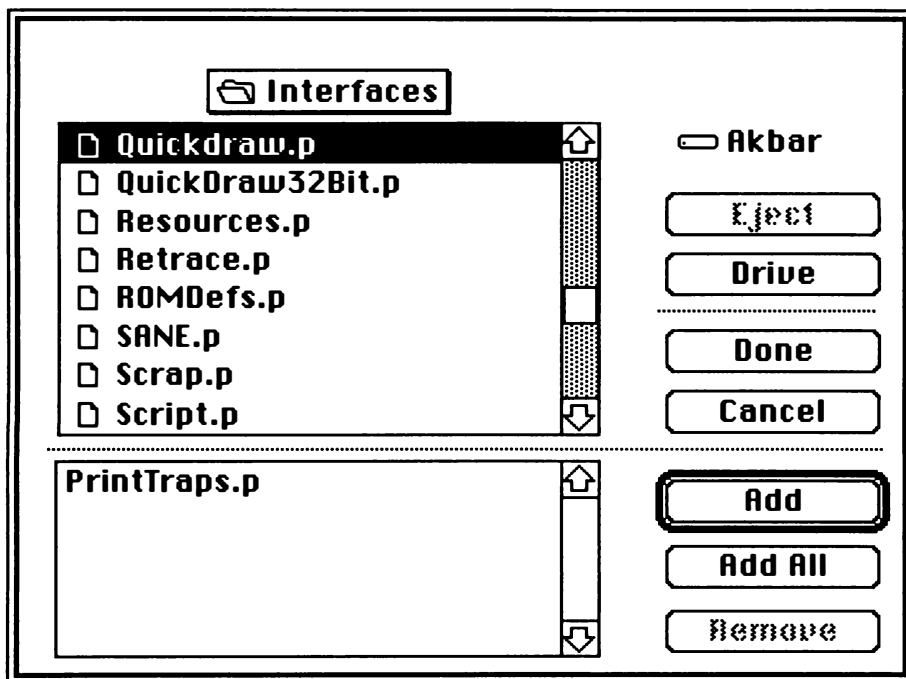


The top list displays the contents of the folder you're currently in. The bottom list shows the files THINK Pascal will add to your project.

The first source file you'll add is `PrintTraps.p`, the interface for the Print Manager described in *Inside Macintosh IV*. Move to the Interfaces folder. To add `PrintTraps.p`, either select it and click on the Add button, or double-click on it. Notice that `PrintTraps.p` disappears from the top list and appears in the bottom list.

Note: If you accidentally add the wrong file to the bottom list, select it and click on Remove. Then add the correct file.

Your dialog should look like this:



The next two files you need to add are in here, too. Add the file `Script.p` by selecting it and clicking on **Add** or by double-clicking on it. It's the interface for the Script Manager, in *Inside Macintosh V*. ObjectDraw doesn't use the Script Manager, but the interface file defines the function `GetMBarHeight` which ObjectDraw uses to figure out where to position its windows.

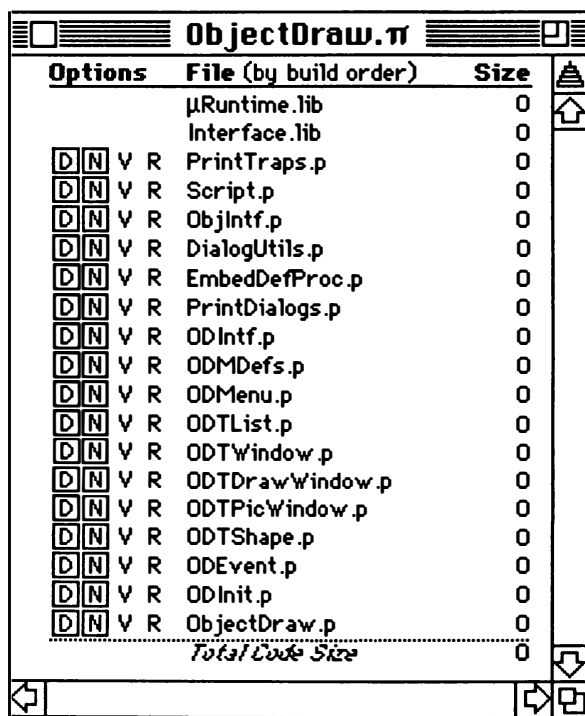
Finally, add `ObjIntf.p`. It contains the most primitive object class, `TObject`.

Note: You don't have to add `ObjIntf.p` to your project to use Object Pascal. You do need it if you define your objects in terms of `TObject` like ObjectDraw does.

Now move back to ObjectDraw.f. The rest of the source files are in this folder. Add the source files in this order:

- DialogUtils.p
- EmbedDefProc.p
- PrintDialogs.p
- ODIntf.p
- ODMDefs.p
- ODMenu.p
- ODList.p
- ODTWindow.p
- ODTDrawWindow.p
- ODTPicWindow.p
- ODTShape.p
- ODEvent.p
- ODInit.p
- ObjectDraw.p

You finished adding all the files you need to your project, so click Done. Your project window should now look like this:



If you discover that you've made a mistake in the order that you add the files, don't worry. After you've added all the files to the project, just click on a file name and drag it to the proper position. To learn more about arranging files in projects, see Chapter 7, "Working with Projects."

Note: Files have to appear in the project in a particular order because Pascal requires that you define something before you use it.

This is what's in each of the files. You might want to print them out to study them, but for this tutorial that's not necessary.

File	Description
DialogUtils.p	Useful utilities for dialog boxes
EmbedDefProc.p	A routine to let a program use definition routines (like an MDEF) that are in a program's source, and not in a separate resource.
PrintDialogs.p	Routines to handle the customized print dialogs.
ODIntf.p	The basic types, constants, and variables that ObjectDraw uses.
ODMDefs.p	The MDEF resources for the special pallet-like Tool , Color , and Pattern menus.
ODMenu.p	Routines to handle menu and Command-key commands.
ODTList.p	The class TList, a linear list of objects, used for the list of shapes in an ObjectDraw window.
ODTWindow.p	The class TWindow. ObjectDraw never creates an instance of this class, but both TDrawWindow and TPicWindow inherit it. It defines behavior common to both classes of windows.
ODTDrawWindow.p	The class TDrawWindow, an ObjectDraw window you draw in.
ODTPicWindow.p	The class TPicWindow, an ObjectDraw window that displays a picture.
ODTShape.p	Classes for all the kinds of shapes you can draw.
ODEvent.p	The event loop for ObjectDraw and routines to handle most Macintosh events, like mouse, window, and key events.
ODInit.p	The initialization routines for ObjectDraw.
ObjectDraw.p	The main program for ObjectDraw.

Setting the Run Options

Before you run the project, you must let the project know where its resources are. The resources contain the contents of the menus, text used in the **About...** box, and more. Choose the **Run Options...** command in the **Run** menu. You'll see this dialog box. Click on the Use Resource File: check box.

Run-time Environment Settings

Resources

☒ Use resource file:

for resources used by the project.

Text Window

Text Window saves characters

☐ Echo to the printer

☐ Echo to the file:

Hello world. x = 811.79.

Monaco ▼
9 ▼

Memory

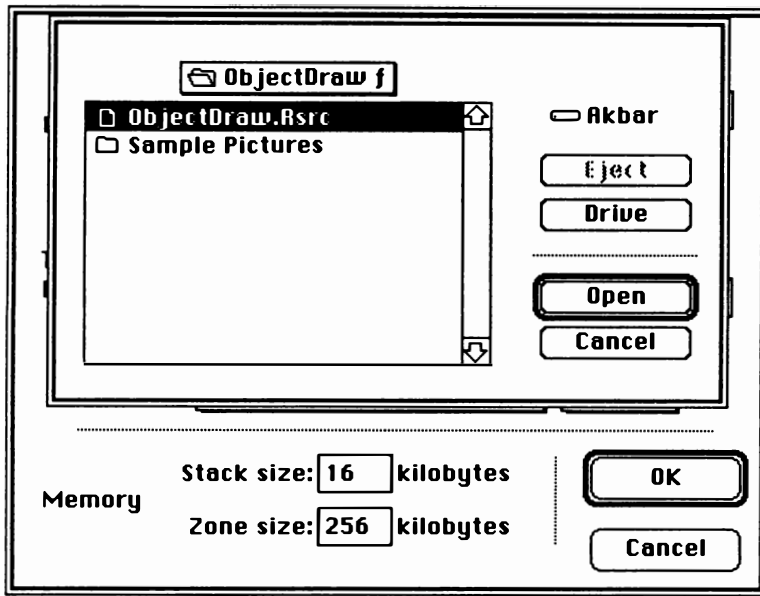
Stack size: kilobytes

Zone size: kilobytes

OK

Cancel

You'll see a standard file dialog. Select `ObjectDraw.Rsrc` and click on the Use button. Then, click on the OK button in the **Run Options...** dialog.



When you run ObjectDraw in the THINK Pascal environment, THINK Pascal will open this file to find ObjectDraw's resources. And when you build the ObjectDraw application, THINK Pascal will place the resources in this file into the application.

The resource file `ObjectDraw.Rsrc` was created with ResEdit. You can use ResEdit to look at or change some of the resources.

NOTE: A RESOURCE FILE USED BY A PROJECT MUST BE IN THE SAME FOLDER AS THE PROJECT.

Setting the Compile Options

This project takes advantage of a THINK Pascal extension to the Pascal language, called the 'USES' Extensions. This option lets you shorten the **uses** clause in your files.

To take advantage of this option, you have to turn it on. Choose **Compile Options...** from the **Project** menu. Click on the 'USES' Extensions box. The dialog should look like this:

Compiler Variables:

```

THINK_Pascal = TRUE;
THINK_Pascal_Version_4 = TRUE;
Elem881 = TRUE;
    
```

☒ **'USES' Extensions**

Code Generation:

☐ **68020/68030** ☐ **Large Sets**

☐ **68881/68882** ☐ **Long Names**

☐ **Profile**

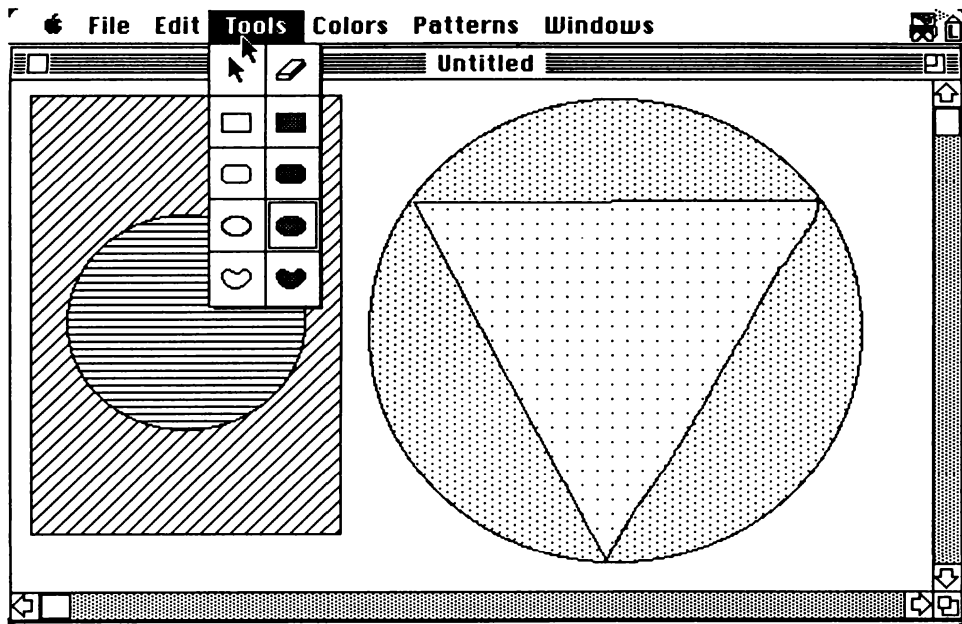
Now, click on OK.

Note: You don't have to understand the 'USES' Extensions option to finish this tutorial. The following explanation gives you an idea of what the option does. If you want more information, see Chapter 10, "Units and Libraries."

The 'USES' Extensions option lets you use **propogated uses**. If your unit uses other units, any unit that uses your unit also uses those units automatically. Say, you have three units A, B, and C. Unit A includes unit B in its **uses** clause, and unit C includes unit B in its **uses** clause, but unit C doesn't include unit A. If you use the 'USES' Extensions, THINK Pascal assumes that unit C uses unit A, even though unit A doesn't appear in unit C's **uses** clause. If you don't use the 'USES Extensions, THINK Pascal assumes unit C doesn't use unit A.

Running the Project

Now you're ready to run the project. Choose **Go** from the **Run** menu. THINK Pascal loads all the libraries and compiles all the source files. When it finishes compiling, THINK Pascal launches the ObjectDraw application.



If you like, you can click on the bug spray can in the far right of the menu bar to halt the program so you can use THINK Pascal's debugging tools. The debugging tools are described in Chapter 9, "Debugging Programs, and Chapter 14, "LightsBug."

Building the Application

Once you're sure that the ObjectDraw application runs correctly, you can build it as a double-clickable application. First, you need to give the application a signature, so the Finder knows how to display its icon.

Choose **Set Project Type...** in the **Project** menu. Set the creator to **RS\$\$** and make sure that the **Bundle Bit** check box is checked. Click on the OK button to continue.

File Information

Type: **APPL** Creator: **RS\$\$** ☒ **Bundle Bit** ☐ **Far Code**

Resource Information

Name:

Type: ID: Attributes: ☐

☐ **Multi-Segment** ☐ **Custom Header**

Segment Type:

Driver Information

Flags: ☐ Delay:

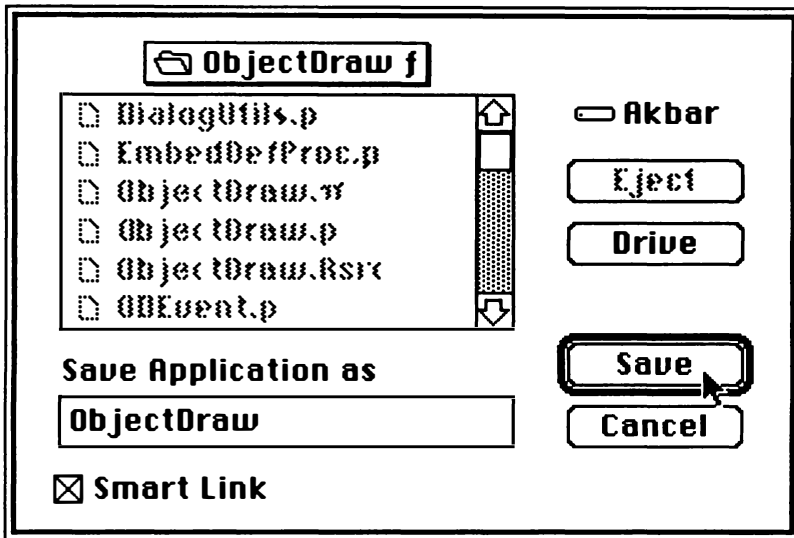
Mask: ☐

OK **Cancel**

Note: If you don't set the creator to **RS\$\$**, the Finder won't display ObjectDraw's icon. **RS\$\$** are the initials of the developer who wrote ObjectDraw (Rich Siegel). The **\$\$** is there because he's Rich.

The Finder uses the signature, the **BNDL**, **FREF**, **ICN#**, and **RS\$\$** resources in ObjectDraw's resource file to give an application an icon. *Inside Macintosh III*, Chapter 1, "The Finder Interface", and *Inside Macintosh VI*, Chapter 9, "The Finder Interface," describes the mechanism in detail.

You're finally ready to build the application. Choose **Build Application...** from the **Project** menu. You'll see a dialog box asking you to name the application. Name the application **ObjectDraw**.



When the Smart Link check box is checked, THINK Pascal strips out any code the application doesn't use so that the final application is as small as possible.

After THINK Pascal finishes building your application, quit to the Finder, and look in the ObjectDraw folder. There you'll see your new application's icon.



Double-click on it to make sure it works. Congratulations! You've built a real Macintosh application.

Where to Go Next

The next chapter shows you how to build a desk accessory. If you want to keep playing with ObjectDraw, read Chapter 9, "Debugging Programs," and Chapter 14, "LightsBug," to learn how to use THINK Pascal's debugging tools. If you want to start creating your own programs, skip to the "Using THINK Pascal" section. And if you want to learn more about Object Pascal, which is used throughout ObjectDraw, see the *Object-Oriented Programming Manual*, Chapter 4, "Object Pascal," and Chapter 5, "Tutorial: LearnOOP."

Tutorial: Hex Dump DA

5

Introduction

This tutorial shows you how to test and build a desk accessory in THINK Pascal. The desk accessory is Hex Dump, a utility that displays the contents of a file in hexadecimal and ASCII. Some of the instructions in this tutorial are the same as for building an application, so you'll be familiar with the process. This tutorial takes you step by step, so if you didn't build ObjectDraw, you'll still be able to create Hex Dump. You might want to take a glance over at the previous chapter before you begin, though.

Building a desk accessory is a little more difficult than building an application, and this tutorial will introduce you to some new concepts and commands fairly quickly. You may want to skim the tutorial first before you start just to make sure you're ready.

Before you begin

Make sure that you followed the instructions in Chapter 2, "Installing THINK Pascal," for installing THINK Pascal on your Macintosh. For this tutorial, you need to make sure these files and folders are installed:

- The Libraries folder. It's from the Interfaces & Libs.sea archive
- The Interfaces folder. It's from the Interfaces & Libs.sea archive
- The DA Shell file in the THINK Pascal 4.0 Folder. It's from the Interfaces & Libs.sea archive.
- The Hex Dump Folder in the THINK Pascal 4.0 Demos folder. This folder contains all the source files and resource files that you'll use in this tutorial. It's from the THINK Pascal 4.0 Demos.sea archive.

Note: The pictures for this tutorial were made under System 6. The tutorial works under System 7.0, but some dialog boxes will look slightly different from those in the pictures.

Topics covered in this chapter

- Writing desk accessories
- Creating the project
- Changing a library
- Adding the source files
- Setting the Compile Options
- Setting the Run Options
- Segmenting the project
- Building the desk accessory
- Where to go next

Writing Desk Accessories

Desk accessories and applications are structurally quite different. A desk accessory is a small program that runs as the “guest” of an application. (If you’re using MultiFinder, the DA Handler is the “host” program.) While applications get their events from the Toolbox’s Event Manager, desk accessories get their events directly from the system. Desk accessories need to be able to respond to certain conditions — the host application quitting, for instance — that don’t apply to applications.

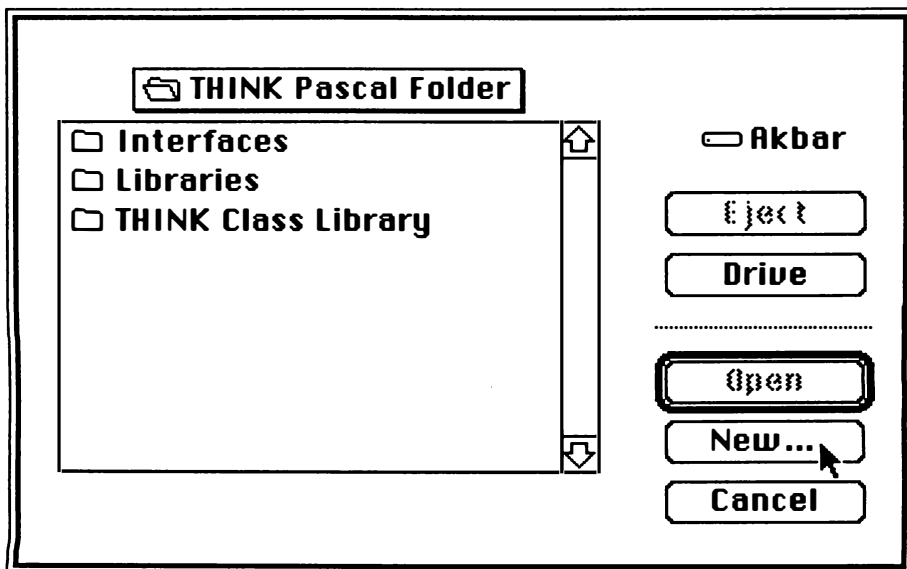
To debug a desk accessory, you have to fool THINK Pascal into thinking that the desk accessory is actually an application. The DA Shell, included in your THINK Pascal package, is designed to provide an environment that looks like an application to THINK Pascal and that looks like a host application to your desk accessory.

In this tutorial, you’ll learn how to use the DA Shell to debug your desk accessories. The main point to remember is that you test your desk accessory in one environment and you build it in another. It may sound hard at first, but it’s really not. All it takes is a couple of extra steps.

Creating the Project

Since you’ve already copied the Hex Dump Folder to your disk, you don’t need to create a new folder for the Hex Dump project. But you do need to remove the project file, named Hex Dump.π, that’s in the Hex Dump Folder. That project lets you use Hex Dump without completing this tutorial. Move it to another folder or back it up to another disk.

Double-click on the THINK Pascal icon to launch THINK Pascal. THINK Pascal displays this dialog when you launch it from the Finder.



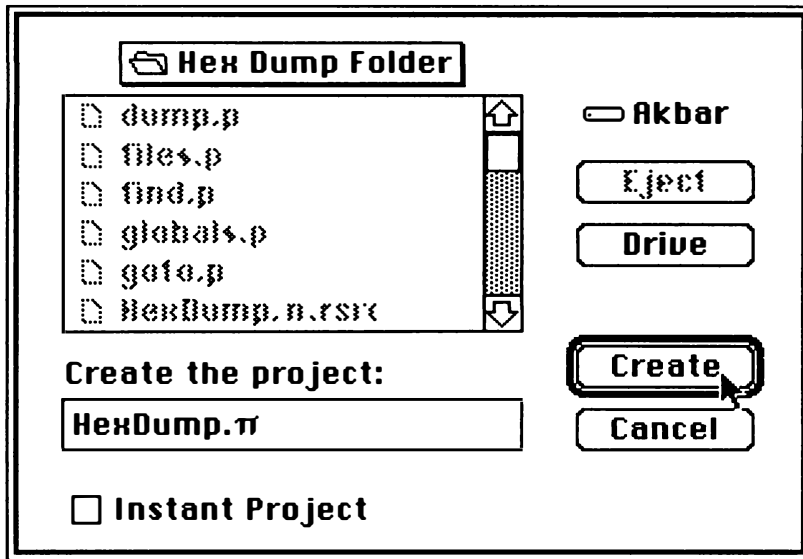
You're creating a new project, so click on the New... button.

THINK Pascal displays a standard file dialog that lets you create projects.

Move to the Hex Dump Folder that you copied from your THINK Pascal disk.

Note: It's very important to move to the Hex Dump Folder.

Name the new project HexDump.π. Make sure the Instant Project option is off, and click on the Create button. (To make a "π" hold down the Option key as you press the letter "p". By convention, all THINK Pascal projects end in .π.)



Note: To learn more about the Instant Project option, see Chapter 3, "Tutorial: Bullseye."

THINK Pascal creates a new project document on the disk and displays a project window. Your project window should look like this:



Next, you'll change a library in your project, and then you'll add the source files.

Changing a Library

THINK Pascal automatically inserts the names of two standard libraries into your project. `Runtime.lib` contains the code for all the standard Pascal routines (like `writeln`). `Interface.lib` contains the glue code for all the Macintosh Toolbox routines that *Inside Macintosh* marks [Not in ROM].

If you went through the tutorial in the last chapter, you remember that it's better to use the `μRuntime.lib` library when you're writing real Macintosh applications — that is, applications that don't use any Pascal standard input/output routines. Well, the fact of the matter is that you can't use the Pascal input/output routines in desk accessories anyway.

When you're writing desk accessories, you use another library called `DRVRRuntime.lib`. This library is designed specifically for programs that are not applications: desk accessories, device drivers, and code resources. The difference between `μRuntime.lib` and `DRVRRuntime.lib` is that `DRVRRuntime.lib` uses register A4 instead of register A5 to access global variables.

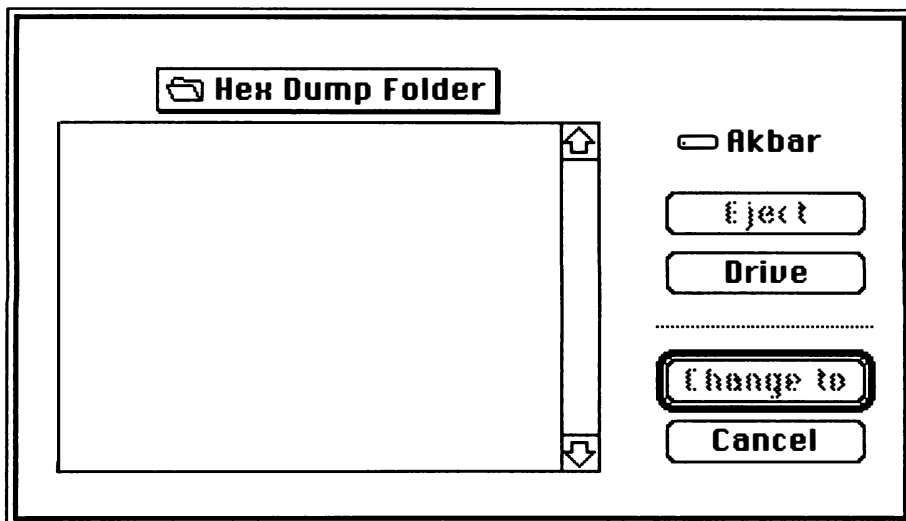
Note: You don't have to know what this means to write desk accessories, but if you're interested, look at Chapter 12, "Building Projects."

When you're testing your desk accessory with the DA Shell (which you're about to do), you use `μRuntime.lib`. After you've tested your desk accessory, and it's time to build it, you use `DRVRRuntime.lib`. This sounds more difficult than it is. Here's a chart that makes things clearer:

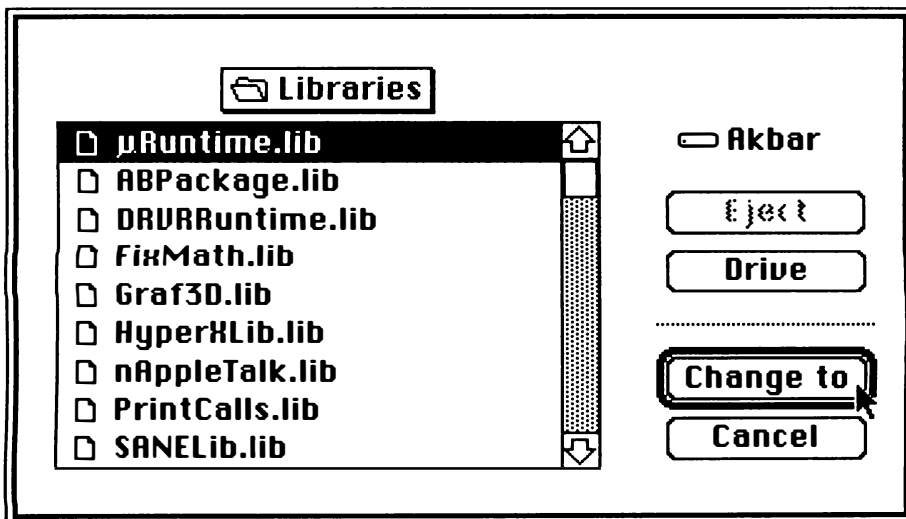
When you're...	You use...
Building an application that uses standard Pascal input/output	<code>Runtime.lib</code> (the default)
Building a Macintosh application that doesn't use Pascal input/output	<code>μRuntime.lib</code>
Testing a desk accessory with the DA Shell	<code>μRuntime.lib</code>
Building a desk accessory	<code>DRVRRuntime.lib</code>
Building a code resource	<code>RSRCRuntim.lib</code>

If you're still not clear about all this library stuff, don't worry. This tutorial takes you step by step.

To replace `Runtime.lib` with `μRuntime.lib`, hold down the Option key as you double-click on `Runtime.lib` in the project window. THINK Pascal displays this dialog:

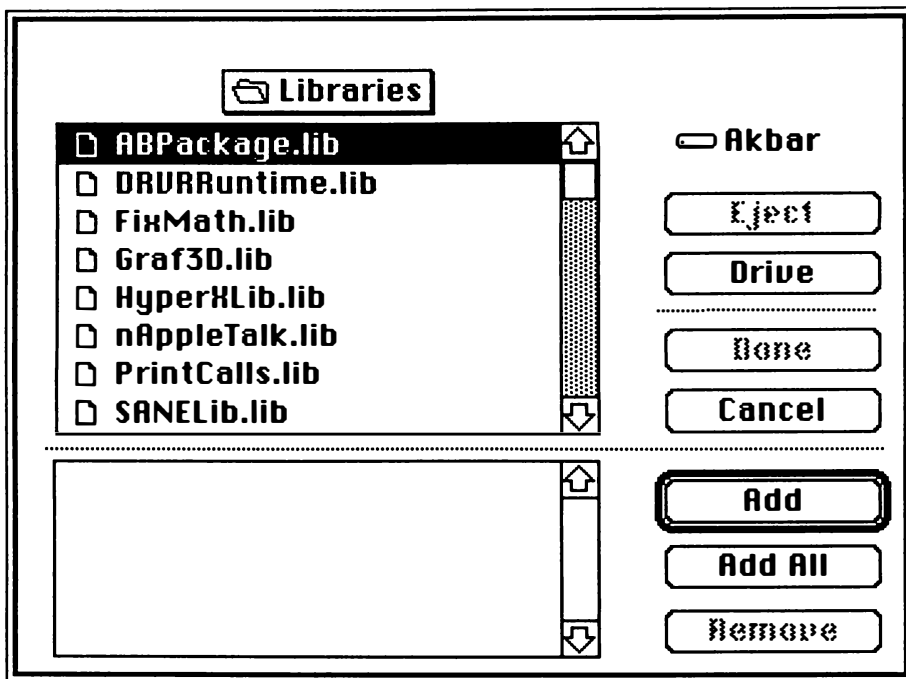


`μRuntime.lib` is in the Libraries folder, so move to that folder, select `μRuntime.lib`, and click on the Change To button.



Adding the Source Files

Now you'll add the source files to your project. Hold down the Option key and select the **Add Files...** command from the **Projects** menu. To see the command, hold down the Option key as you select the **Project** menu. You see this dialog:

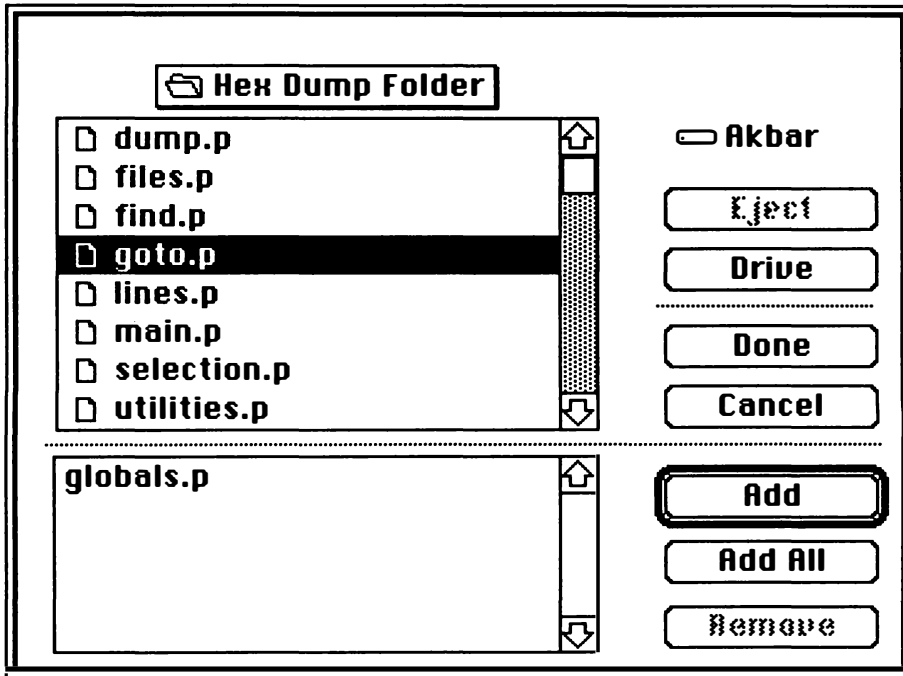


The top list displays the contents of the folder you're currently in. The bottom list shows the files THINK Pascal will add to your project.

The first source file you'll add is `globals.p`. Go to the `Hex Dump Folder`. To add `globals.p`, either select it and click on the **Add** button, or double-click on it. Notice that `globals.p` disappears from the top list and appears in the bottom list.

Note: If you accidentally add the wrong file to the bottom list, select it and click on **Remove**. Then add the correct file.

Your dialog should look like this:



Now add the rest of the source files in this order:

- utilities.p
- lines.p
- windows.p
- files.p
- selection.p
- goto.p
- find.p
- dump.p
- main.p

Since you'll be testing the desk accessory before you actually build it, you need to add the DA Shell to your project. This source file is in the same folder as THINK Pascal. If you followed the installation instructions, that folder is THINK Pascal 4.0 Folder. Move to that folder and add the file DA Shell.

You finished adding all the files you need to your project, so click Done. Your project window should now look like this:

HexDump.π			
Options	File (by build order)	Size	
	μRuntime.lib	0	
	Interface.lib	0	
<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> V R	globals.p	0	
<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> V R	utilities.p	0	
<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> V R	lines.p	0	
<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> V R	windows.p	0	
<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> V R	files.p	0	
<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> V R	selection.p	0	
<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> V R	goto.p	0	
<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> V R	find.p	0	
<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> V R	dump.p	0	
<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> V R	main.p	0	
<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> V R	DA Shell	0	
<i>Total Code Size</i>		0	

If you discover that you've made a mistake in the order that you add the files, don't worry. After you've added all the files to the project, just click on a file name and drag it to the proper position. To learn more about arranging files in projects, see Chapter 7, "Working with Projects."

Note: Files have to appear in the project in a particular order because Pascal requires that you define a procedure or function before you can use it.

Here is a description of what each of the files does. You might want to print them out to study them, but for this tutorial that's not necessary.

File	Description
globals.p	All the global variables for the project.
utilities.p	Some utility routines used in the project.
lines.p	Fills a string with a line of the Hex Dump display.
windows.p	Handles Hex Dump's window.
files.p	Opens, closes, and reads the displayed file.
selection.p	Highlights selected data from the displayed file (used by goto.p and find.p).
goto.p	Lets the user go to a position in the file, and displays the data at that position.
find.p	Lets the user enter an ASCII or hexadecimal string, and finds it.
dump.p	Dumps the hexadecimal and ASCII representation of the displayed file to a text file.
main.p	Defines a function main that THINK Pascal uses as the entry point of the desk accessory. See Chapter 12, "Building Projects," for details.

Setting the Compile Options

THINK Pascal lets you use a desk accessory two ways: you can run it under the DA Shell to test it or you can build a "suitcase" file when it's finished. However, when you build the suitcase file, you need to include some code that you don't need when you run under the DASHell. Hex Dump uses conditional compilation so you can use the same source files in either case. You'll define a compiler variable called DASHell to let THINK Pascal know which code to compile.

For example, this procedure will set up and restore the A4 register only when you're building a suitcase file:

```
procedure ScrollProc(theControl: ControlHandle; theCode: integer);
...
begin
{$IFC NOT DASHell}
    SetUpA4;
{$ENDC}
    if theCode = scrollCode then
...
{$IFC NOT DASHell}
    RestoreA4;
{$ENDC}
end;
```

Note: For more information on compiler variables, see "Using Conditional Compilation" in Chapter 15. For more details on writing desk accessories in THINK Pascal, see "Building Desk Accessories and Device Drivers" in Chapter 12.

THINK Pascal lets you define compiler variables with the **Compile Options...** command. Choose **Compile Options...** from the **Project** menu. You'll see this dialog:

Compiler Variables:

```

THINK_Pascal = TRUE;
THINK_Pascal_Version_4 = TRUE;
Elms881 = TRUE;
    
```

☐ **'USES' Extensions**

Code Generation:

<input type="checkbox"/> 68020/68030	<input type="checkbox"/> Large Sets
<input type="checkbox"/> 68881/68882	<input type="checkbox"/> Long Names
<input type="checkbox"/> Profile	

OK Cancel

THINK Pascal defines three symbols for you. `THINK_PASCAL` is always defined as true. You can use this symbol if you need to know whether your program is being compiled in THINK Pascal. `THINK_Pascal_Version_4` is always true, too. Use this symbol if you need to know whether your program is being compiled by the latest version of THINK Pascal. The symbol `Elms881` is used to generate code for the MC68881 math coprocessor. See Chapter 15, "Compile Directives," to learn more.

To run HexDump under the DAShell, define the compiler variable `DAShell` to be `true`. Type `DAShell=TRUE;` into the dialog and click on the OK button:

Compiler Variables:

```
THINK_Pascal = TRUE;  
THINK_Pascal_Version_4 = TRUE;  
Elms881 = TRUE;  
DAShell = TRUE;
```

☐ 'USES' Extensions

Code Generation:

<input type="checkbox"/> 68020/68030	<input type="checkbox"/> Large Sets
<input type="checkbox"/> 68881/68882	<input type="checkbox"/> Long Names
<input type="checkbox"/> Profile	

Later on, when you build the desk accessory, you'll set `DAShell` to `FALSE`.

Note: There's nothing magic about this compiler variable. You don't have to write your desk accessories this way if you don't want to.

Setting the Run Options

Now you have to make sure that the project knows where its resources are. Choose the **Run Options...** command in the **Run** menu. You'll see this dialog box. Click on the Use resource file: check box.

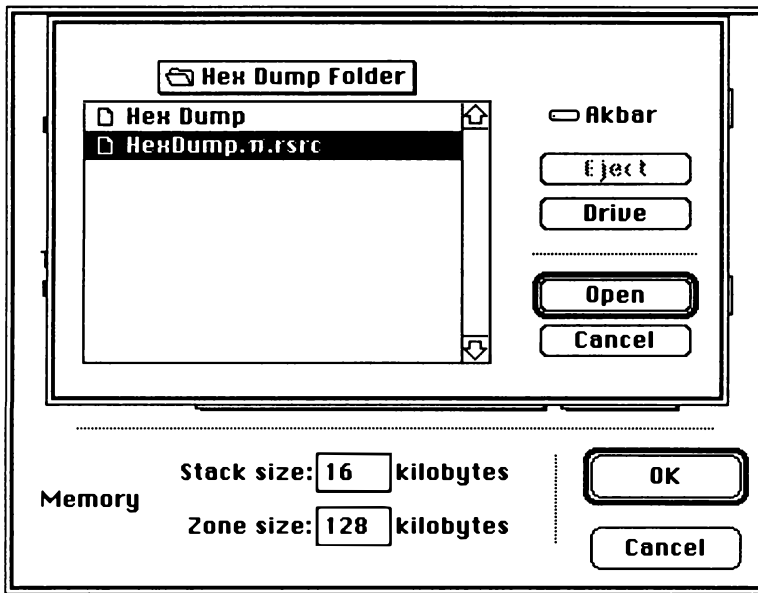
Run-time Environment Settings

Resources ☒ Use resource file:
for resources used by the project.

Text Window Text Window saves characters
☐ Echo to the printer
☐ Echo to the file:

Memory Stack size: kilobytes
Zone size: kilobytes

You'll see a standard file dialog. Select `HexDump.π.rsrc` and click on the Use button. Then, click on the OK button in the **Run Options...** dialog.



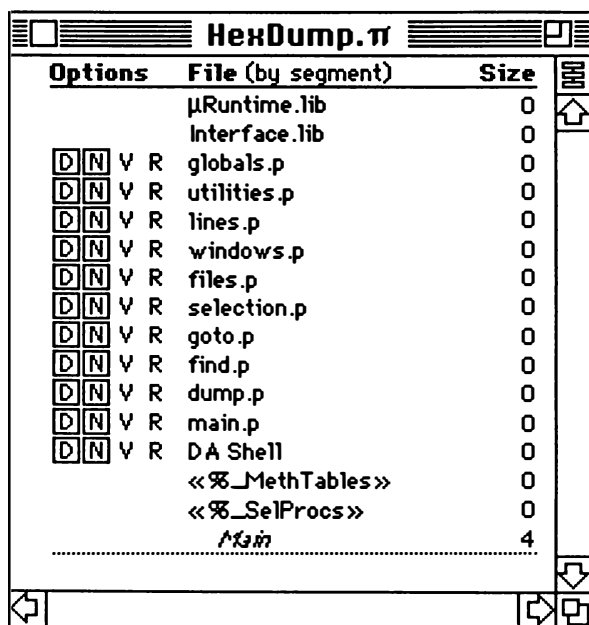
The resource file `HexDump.π.rsrc` was created with ResEdit. You can use ResEdit to look at or change some of the resources.

NOTE: A RESOURCE FILE USED BY A PROJECT MUST BE IN THE SAME FOLDER AS THE PROJECT.

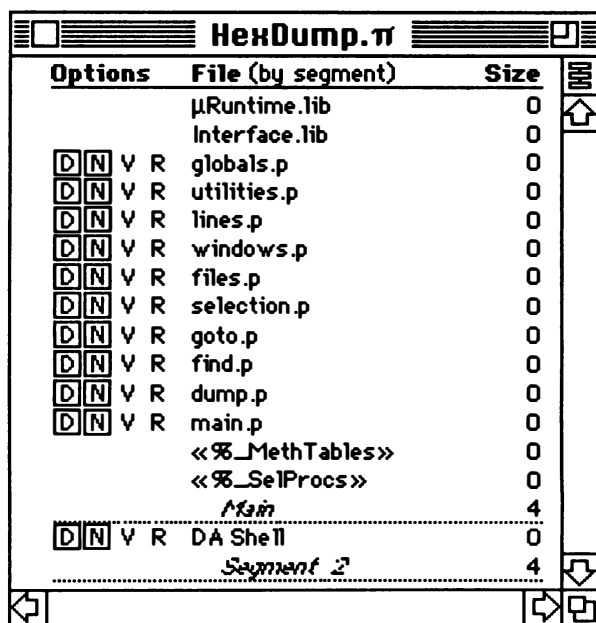
Segmenting the Project

To make sure that Hex Dump will work well when memory is low, you'll split it into two segments, moving some infrequently used code to a new segment. Macintosh programs are made up of one or more segments that contain the machine language code of your program. The Macintosh operating system takes care of loading these segments when you need them. If you don't use any code in a segment while using a desk accessory or application, that segment will not be loaded. Although it's a good idea to split your program into a few small segments, the only time you must segment your program is when it is over 32K. When THINK Pascal discovers that a segment is larger than 32K, you must break up your program into more segments.

Click on the "Towers of Hanoi" icon (🏰) just above the up-arrow on the right of your project window. The project window changes, and it looks like this:



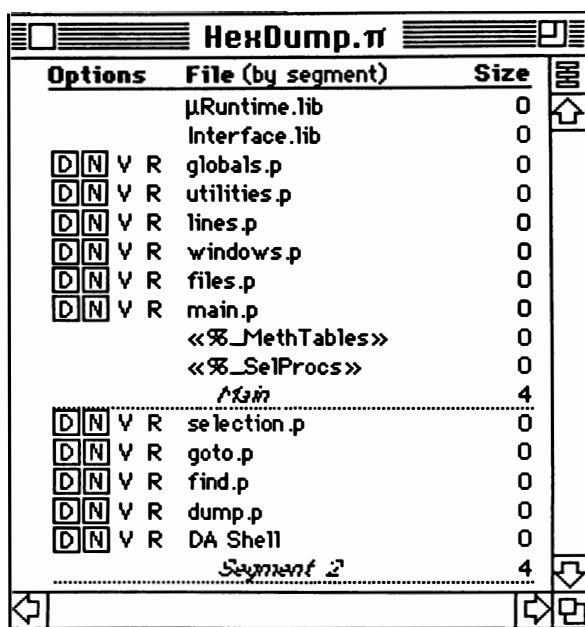
Click on the last file in the project, DA Shell, and drag it below the line labeled Main. Your project window should now look like this:



Now your program has two segments. Move these files into the new segment with DA Shell:

- selection.p
- goto.p
- find.p
- dump.p

Your project window should look like this:



Click in the icon in the upper right corner of the window again to go back to the original view of your project.

The normal view you're used to by now is called the Build Order view because it lists the files in the order that THINK Pascal compiles them. The other view is called the Segment view because it shows how your program is broken up into different segments.

Note: To learn more about segmentation, see Chapter 7, "Working with Projects."

Running the Project

Now you're ready to run the project. Choose **Go** from the **Run** menu. THINK Pascal loads all the libraries and compiles all the source files. When it finishes compiling, THINK Pascal launches the DA Shell. To use your desk accessory, choose the **Sample DA** command from the **Apple** menu. The DA Shell starts your program and feeds it events so it looks like it's a desk accessory.

Hex Dump will ask you to choose a file. After you choose one, you'll see a Hex Dump window, like this one:

main.p										Data fork
EOF = 12968 (\$32A8)										
000000:	7820	096D	6169	6E2E	4865	7844	756D	7044	{..main.HexDumpD	
000010:	412E	707D	0D7B	2020	2020	7D0D	7820	0948	A.p).{...}.{..H	
000020:	6578	4475	6D70	2069	7320	6120	4441	2074	exDump.is.a.DA.t	
000030:	6861	7420	6469	7370	6C61	7973	2074	6865	hat.displays.the	
000040:	2063	6F6E	7465	6E74	7320	6F66	2061	2066	.contents.of.a.f	
000050:	696C	6520	696E	2068	6578	2061	6E64	2041	ile.in.hex.and.A	
000060:	5343	4949	2C09	097D	0D7B	2009	696E	7465	SCII,...){..inte	
000070:	6E64	6564	2061	7320	616E	2065	7861	6D70	nded.as.an.examp	
000080:	6C65	206F	6620	686F	7720	746F	2077	7269	le.of.how.to.wri	
000090:	7465	2061	2064	6573	6820	6163	6365	7373	te.a.desk.access	
0000A0:	6F72	792E	2020	416D	6F6E	6720	6F74	6865	ory...Among.othe	
0000B0:	7209	7D0D	7820	0974	6869	6E67	732C	2069	r.){..things,i	
0000C0:	7420	7368	6F77	7320	7468	6520	7374	7275	t.shows.the.stru	
0000D0:	6374	7572	6520	6F66	2061	2044	412C	2068	cture.of.a.DA,h	
0000E0:	6F77	2061	2044	4120	6765	7473	2061	6E64	ow.a.DA.gets.and	
0000F0:	2072	6573	706F	6E64	7320	746F	097D	0D7B	.responds.to.){	
000100:	2009	6576	656E	7473	2C20	616E	6420	686F	.events,and ho	
000110:	7720	6120	4441	206D	616E	6167	6573	2074	w.a.DA.manages.t	
000120:	6865	206D	656E	7520	6261	722E	0909	0909	he.menu.bar.....	
000130:	0909	0909	097D	0D7B	2009	7D0D	7820	094A){..}{..J	
000140:	6566	6620	4D61	7474	736F	6E20	616E	6420	eff.Mattson.and.	
000150:	5374	6576	6520	5374	6569	6E2C	2053	796D	Steve.Stein,.Sym	

If you like, you can click on the bug spray can in the far right of the menu bar to halt the program so you can use THINK Pascal's debugging tools.

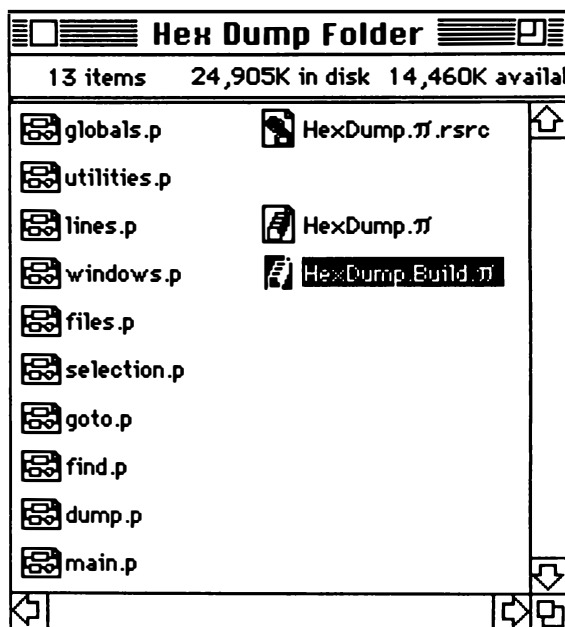
To go back to THINK Pascal, choose **Quit** from the DA Shell's **File** menu.

Building the Desk Accessory

Once you're sure that the Hex Dump runs correctly, build it as a desk accessory that you can add to your system with the Font/DA Mover. Start by quitting THINK Pascal and going into the Finder. Duplicate HexDump.π and name the copy HexDump.Build.π. You'll modify this copy of the project so you can easily switch from testing to building the desk accessory.

Note: When you build a desk accessory, device driver, or code resource, you should keep two separate project documents: one for debugging and one for building. To change from debugging to building (or vice versa), you only need to open a new project document instead of changing several options. The projects can share the same files.

Your Hex Dump Folder should look something like this:



Double-click on HexDump.Build.π to start THINK Pascal again.

Choose the **Set Project Type...** command in the **Project** menu and click on the Desk Accessory icon.

In the Resource Information Name box, type in Hex Dump. This is the name that will appear in the Apple menu when you install your desk accessory with the Font/DA Mover. Check the Multi-Segment check box in the Resource Information section.

Your dialog box should look like this:

The dialog box is titled "Resource Editor" and contains three main sections: File Information, Resource Information, and Driver Information. On the left side, there are four icons representing different resource types: Application (hand holding a diamond), Desk Accessory (suitcase), Driver (printer), and Code Resource (monitor). The "Desk Accessory" icon is selected and highlighted with a thick border.

File Information

Type: Creator: ☐ Bundle Bit ☐ Far Code

Resource Information

Name:

Type: ID: Attributes: ☒ 00

☒ Multi-Segment ☐ Custom Header

Segment Type:

Driver Information

Flags: ☒ 0400 Delay:

Mask: ☒ 016A

OK Cancel

The Type and Creator values let the Finder know that the final file is a desk accessory, so it'll be displayed with the suitcase icon. The resource type DRVr lets the Macintosh know that it's a desk accessory.

When you've set all the fields, click on the OK button.

Remember that compiler variable, `DAShell`, that controls how your source files are compiled? You need to change it when you compile your program as a stand-alone desk accessory. Choose the **Compile Options...** command in the **Project** menu, and change this compiler variable:

Compiler Variables:

```

THINK_Pascal = TRUE;
THINK_Pascal_Version_4 = TRUE;
Elms881 = TRUE;
DAShell = FALSE;
    
```

☐ **'USES' Extensions**

Code Generation:

☐ **68020/68030**
☐ **Large Sets**
☐ **68881/68882**
☐ **Long Names**
☐ **Profile**

Now replace `μRuntime.lib` with `DRVRRuntime.lib`. Hold down the Option key as you double-click on `μRuntime.lib` in your project window. When THINK Pascal displays the standard file dialog, move to the **Libraries** folder, and double-click on `DRVRRuntime.lib`.

Next remove the DA Shell source file from your project. Select the DA Shell file in the project window and choose **Remove** from the **Project** menu. This command lets you take source files and libraries out of your project.

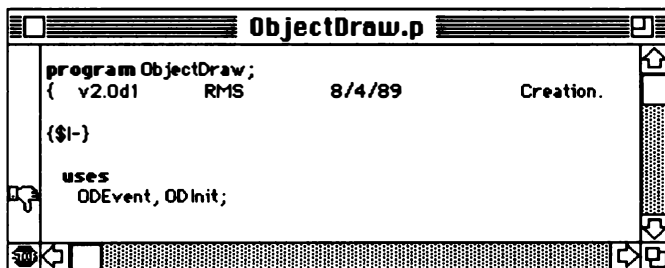
Turn off all the compile options. The options are the D's, N's, V's, and R's to the left of the file names. Click on each option that's in a box or hold down the Option key as you click on a box to turn off that option for all the files in the project.

For example, if a file that uses the unit `ODInit` is in the wrong order, THINK Pascal displays this error message:



"ODInit" isn't in the current project, hasn't been successfully compiled, or is in the wrong build order.

and points out the error:



Segmenting a Project

Most Macintosh applications are made up of several **segments**. Segments are collections of code that the Macintosh segment loader moves in and out of memory.

Note: Segments correspond loosely to overlays in other development environments.

The Macintosh limits segments to 32K, so if you're writing a large program, you will have to segment your code. To learn more on how the Macintosh uses segments, read *Inside Macintosh II*, Chapter 2, "The Segment Loader."

Note: When you're running a program under the THINK Pascal environment, each segment can contain up to 64K. This lets you use segments that are over 32K before smart linking. But when you build your program, each segment must be under 32K after smart linking.

Making segments in the project window

The easiest way to segment programs is to use the segment view of the project window. To see the segment view, click on the "Towers of Hanoi" icon (▲) in the upper-right-hand corner of the project window. This is the segment view of the Hex Dump project:

Options	File (by segment)	Size
	DRVRRuntime.lib	2948
	Interface.lib	10098
D N V R	globals.p	0
D N V R	utilities.p	772
D N V R	lines.p	882
D N V R	windows.p	2748
D N V R	files.p	596
D N V R	main.p	1868
	«%_MethTables»	0
	«%_SelProcs»	0
	<i>Main</i>	19916
D N V R	selection.p	562
D N V R	goto.p	570
D N V R	find.p	1876
D N V R	dump.p	600
	<i>Segment 2</i>	3612

Dotted lines separate the segments, and the name and size of the segment appear after the last entry for the segment. Each line in a segment's listing is called a **file entry**. It represents the code that comes from a file or library.

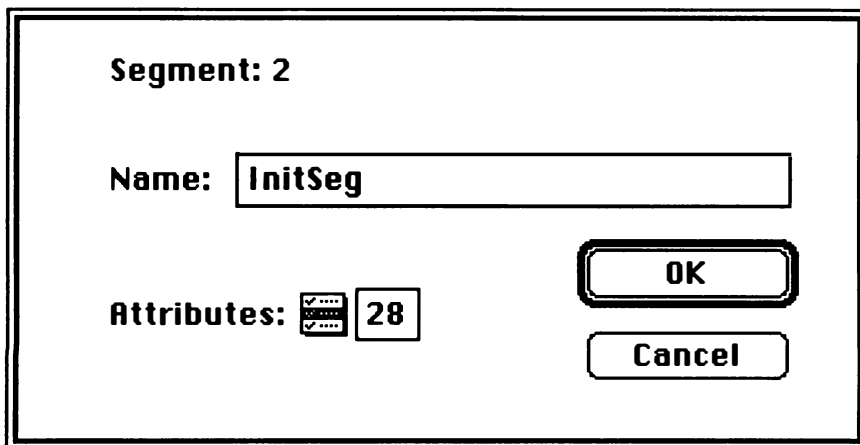
Note: For now, don't worry about the entries «%_MethTables» and «%_SelProcs». They're described later in the section "Object Pascal and segmentation."

By default, THINK Pascal creates a segment called *Main* and places all the code from all the files there. To create a new segment, drag a file entry into the space just below the last file in the window. To change the position of a segment, click on its name and drag it to a new position. To add a file's code to an existing segment, just drag the file entry into the segment. Any segments that become empty are automatically deleted.

The build-order view and the segment view are completely independent. If you change the build order, the segmentation does not change. If you change the segmentation, the build order stays the same. In the segment view you're not moving files. You're moving file entries.


Naming segments

You can edit the name and attributes of a segment. Double-click on a segment's summary line to open a dialog box like this:



Segment: 2

Name:

Attributes: 

OK

Cancel

To change the attributes, either click on the pop-up menu icon to select them or enter a new number into the box. The possible attributes are Purgeable, Locked, Preload, and Protected. By default, Purgeable and Protected are selected. You may want to change the attributes if, for example, you don't want a segment moved out of memory or you want it loaded into memory as soon as the program starts.

Note: Don't use `$_SelfProcs` or `$_MethTables` as segment names. They are reserved by THINK Pascal.

Making segments with the segmentation directive

Sometimes, segmenting your project file by file isn't powerful enough. For example, you may have a large file that doesn't fit into one 32K segment.

In this case, you can create segments with the segmentation directive, `{ $S name }`. This directive puts all the procedures and functions that follow it (up to the next segmentation directive or the end of the file) into the segment *name*. For more information on this directive, see Chapter 15, "Compiler Directives."

Here's an example of the segmentation directive:

```
program myProgram;
    ... Utility Routines ...
    {$S InitSeg}
    ... Initialization Routines ...
    {$S TermSeg}
    ... Termination Routines ...
    {$S Main}
    ... Main Routines ...
end.
```

In this example, THINK Pascal creates three segments: InitSeg, TermSeg, and Main. It places the code as follows:

In this segment...	THINK Pascal places these routines...
Main	The utility and main routines. (THINK Pascal places all code that is not affected by a segmentation directive here. It also places code that follows a {\$S Main} or a {\$S} directive here.)
InitSeg	The initialization routines.
TermSeg	The termination routines.

Choosing how to segment

You can segment your project one of three ways:

- **File by file.** All the routines in a file are in the file's segment. If you want to put a routine in a different segment, put it into a different file. If you want a segment to contain code from several files, move those files into that segment. This is the traditional way to segment applications in THINK Pascal, and it is still the best.
- **Routine by routine.** You place every routine into a segment with Segmentation directives. THINK Pascal places the source files to the Main segment, but no code is in them. Use this method to port MPW Pascal programs to THINK Pascal quickly.
- **A little of both.** Most routines are in their file's segment, but some units are so large you must use the segmentation directive. Use this method when you have exceptionally large units.

Note: Compiling a file that contributes to more than one segment can take a lot of time and memory. THINK Pascal needs enough memory to load in each segment that the file contributes to.

More about the segment view

This is the segment view of the project window for the previous example:

Options	File (by segment)	Size
	Runtime.lib	14928
	Interface.lib	10098
<input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> V R	myProgram.p	242
	«%_MethTables»	2
	«%_SelProcs»	0
	----- <i>Main</i>	25274
	«InitSeg»	70
	----- <i>InitSeg</i>	74
	«TermSeg»	70
	----- <i>TermSeg</i>	74

In addition to file entries, these segments also contain **directive entries**. Directive entries are enclosed in angle quotes («») and represent code from segments created with the segmentation directive. Here, the directive entries are «InitSeg» and «TermSeg». The **file entries** represent code that wasn't placed in a segment with a segmentation directive or code that was explicitly placed in the file's segment with { \$\$ } or { \$\$ main }. Here, the file entries are myProgram.p, Runtime.lib, and Interface.lib.

Note: «%_MethTables» and «%_SelProcs» are directive entries that THINK Pascal creates. They are described below.

You can treat directive entries just like file entries. For example, you can drag the entry «TermSeg» into the segment InitSeg. The project window would look like this:

Options	File (by segment)	Size
	Runtime.lib	14928
	Interface.lib	10098
<input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> V R	myProgram.p	242
	«%_MethTables»	2
	«%_SelProcs»	0
	----- <i>Main</i>	25274
	«InitSeg»	70
	«TermSeg»	70
	----- <i>InitSeg</i>	144

Now, all the code that follows either a `{ $$ InitSeg }` directive or a `{ $$ TermSeg }` directive goes into the `InitSeg` segment.

You can remove any directive entry with a size of 0 (zero). Simply select it, and select **Remove** from the **Project** menu.

Note: You can't remove a file entry with a size of 0. For example, interface files have a size of 0 but are still essential to your project.

Object Pascal and segmentation

THINK Pascal adds two entries to every project: `«%_MethTables»` and `«%_SelProcs»`. These contain the code that Object Pascal uses to dispatch messages. By default they're both placed in the segment `Main`, but you can drag them anywhere. If you use Object Pascal heavily in your programs, you may want to move `«%_MethTables»` into a segment by itself. If you don't use Object Pascal at all, you can remove `«%_SelProcs»`.

Note: If you move `«%_MethTables»` out of `Main`, set the attributes of its new segment to be `Preload` and `Locked`.

If the "Far Code" option is on, the entry `«%_MethTables»` can be up to 64K., instead of just 32K. However, because of this extended limit, you must put `«%_MethTables»` in its own segment whenever the "Far Code" option is on. For more information on the "Far Code" option, see "Building applications with large jump tables" in Chapter 12, "Building Projects."

Setting the Compiler Options

The Options column in the project window displays the state of the debugging options for each file in the project. Enabling or disabling these options affects the code the compiler generates. Chapter 15 goes into detail about the compiler options. This section gives you a brief overview.

Option	Meaning
D	Debug. Generates additional information between each Pascal statement to support stepping, stopping, and observing.
N	Names. Inserts the name of all routines into the code. This is useful for debugging with LightsBug and Macsbug.
V	Overflow Checking. Generates code that checks for integer arithmetic overflows.
R	Range Checking. Generates code that does range checking for array indexing, assignments, and parameter passing. It also generates code that checks for <code>nil</code> pointer dereferencing.

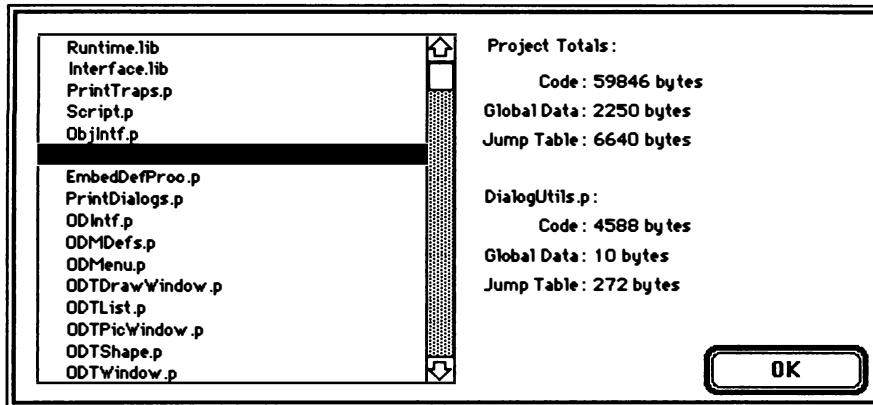
A box around the letter means that the option is enabled. By default, the D and N options are on. To turn the options on and off, click on the option letters.

Note: It's a good idea to turn the V and the R options on while you're debugging your program.

If you hold down the Option or Command key when changing an option, the new state of that option applies to all the files in the project.

Getting Information on a Project's Code & Data Size

The **Get Info...** command in the **Project** menu shows how much code and data each file in your project produces. Choosing the command brings up this dialog:



The list on the left contains all the files in your project. To examine a file, click on it. To examine several files in a range, hold down the Shift key and select the range. To examine several scattered files, hold down the Command key and click on each file.

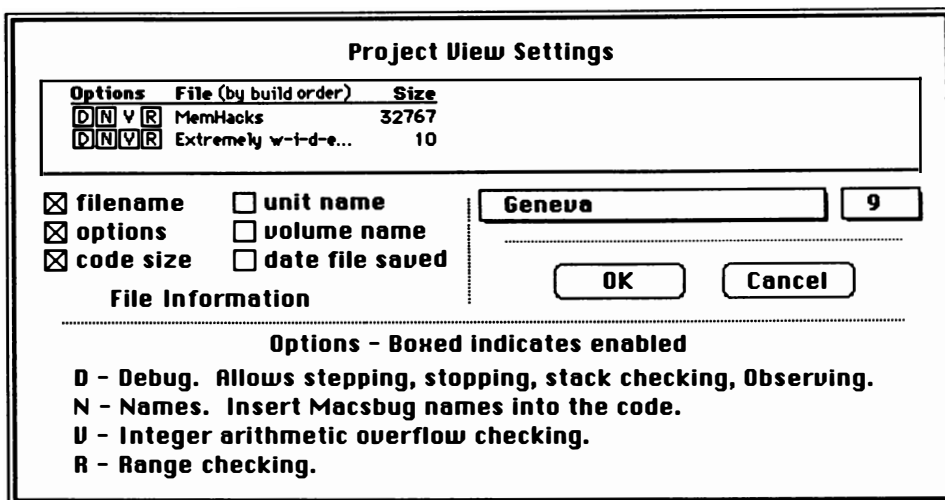
The right side displays the amount of code and data. On top, the "Project Totals" displays total amount of code and data in your project. Underneath is the amount of code and data that the selected file (or files) contributes to the total.

This table explains what the dialog displays:

Title	Meaning
Code	The amount of object code. This is same number in the Size column of the project window.
Global Data	The amount of global data.
Jump Table	The size of the jump table, which contains the addresses of functions and procedures in your application.

Customizing the Project Window

The **View Options...** command in the **Project** menu lets you customize what appears in the project window. When you choose this command, you see this dialog box:



The dialog box is titled "Project View Settings". It contains a table at the top showing file information, followed by several checkboxes for file information, a section for file information, and a section for options.

Options	File (by build order)	Size
<input checked="" type="checkbox"/> D <input checked="" type="checkbox"/> N <input checked="" type="checkbox"/> V <input checked="" type="checkbox"/> R	MemHacks	32767
<input checked="" type="checkbox"/> D <input checked="" type="checkbox"/> N <input checked="" type="checkbox"/> V <input checked="" type="checkbox"/> R	Extremely w-t-d-e...	10

Below the table are four checkboxes:

- ☒ filename
- ☒ options
- ☒ code size
- ☐ unit name
- ☐ volume name
- ☐ date file saved

Below these checkboxes is a section titled "File Information" with a dotted line separator. To the right of this section are two text boxes: "Geneva" and "9". Below these are "OK" and "Cancel" buttons.

Below the "File Information" section is a section titled "Options - Boxed indicates enabled" with a dotted line separator. It contains the following text:

- D** - Debug. Allows stepping, stopping, stack checking, Observing.
- N** - Names. Insert Macsbug names into the code.
- V** - Integer arithmetic overflow checking.
- R** - Range checking.

The two pop-up menus let you choose the font THINK Pascal uses to display the file names in the project window. The check boxes let you choose the information that appears for each file. The box at the top of the dialog box shows you what the project window would look like with the current settings.

Option

filename

options

code size

unit name

volume name

date file saved

Meaning

Displays the name of the file.

Displays the compiler options.

Displays the size of the compiled code.

Displays the name of the unit. The unit name appears only after the file has been compiled.

Displays the volume or folder the file is in.

Displays the last date and time you saved the file within the THINK Pascal environment.

The order that you click the check boxes determines the order of the display. In this example, the check boxes checked were filename, code size, date file saved, and options:

Project View Settings

File (by build order)	Size	Date Saved	Options
MemHacks	32767	12/27/85 10:43	<input checked="" type="checkbox"/> D <input checked="" type="checkbox"/> N <input checked="" type="checkbox"/> V <input checked="" type="checkbox"/> R
Extremely w-i-d-e...	10	01/02/86 23:07	<input checked="" type="checkbox"/> D <input checked="" type="checkbox"/> N <input checked="" type="checkbox"/> V <input checked="" type="checkbox"/> R

☒ filename ☐ unit name

☒ options ☐ volume name

☒ code size ☒ date file saved

Geneva

9

OK

Cancel

File Information

Options - Boxed indicates enabled

D - Debug. Allows stepping, stopping, stack checking, Observing.

N - Names. Insert Macsbug names into the code.

U - Integer arithmetic overflow checking.

R - Range checking.

Recovering Corrupted Projects

When you're debugging a program, you may run across errors that you can't explain. Maybe you see an "Illegal Instruction" error you can't trace. Or maybe THINK Pascal can't open your project.

A common cause of these problems is a corrupted project. When you run a faulty program under THINK Pascal, it can accidentally write to its project file and corrupt it. Writing to an uninitialized pointer or a pointer with the wrong value is frequently the cause. If you have a bug you can't explain or have a project that THINK Pascal refuses to open, try these solutions.

- **Recompile your project.** If you can't compile or run your project, its object code may be corrupted. Choose **Remove Objects** from the **Project** menu and recompile your project.
- **Rebuild your project.** If you can't open your project, try opening another one. If it does open, rebuild your project from scratch, creating a new project file and adding the source files.
- **Make a backup of your project.** If a program corrupts its project frequently, try making a backup of the project. Build the project from scratch and compile it with the **Build** command. When your project is corrupted, make a copy of the backup and use it. THINK Pascal automatically recompiles the files you changed since you made the backup.

Running Programs

8

Introduction

This chapter shows you how to run a program in THINK Pascal. It tells you how THINK Pascal decides which files need to be recompiled, and what happens if there's an error in your program.

What you should know

To run a program in THINK Pascal, you need a project document. If you don't know about projects, read the preceding chapter. If you want to experiment as you read this chapter, use one of the projects included in your THINK Pascal package or one that you created from the tutorials.

Topics covered in this chapter

- Running a program
- When something goes wrong
- Stopping your program
- Compiling, building, and linking without running
- Save options
- Run options

Running a Program

To run a program, just choose **Go** from the **Run** menu. You'll use the other execution commands — **Step Into**, **Step Over**, **Go-Go**, and **Step-Step** — when you debug your program. In the next chapter, you'll learn how to use these commands.

When you choose one of the commands in the **Run** menu, THINK Pascal examines your project to see which files need to be recompiled and relinked. If you've edited any source files or if you've added new source files to the project, THINK Pascal marks them for compilation. If you change the interface part of a unit, THINK Pascal marks all the files that use that unit. After marking the files, THINK Pascal compiles them by build order.

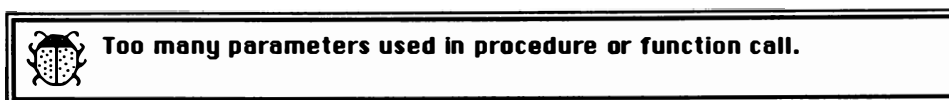
Next, THINK Pascal links your program. It resolves references between files, and makes sure that names are not missing or defined more than once.

If there are any files that have changed, THINK Pascal asks you if you'd like to save them before running the program. You can use the save options discussed below to tell THINK Pascal to always save your files before running or to never save your files before running.

When the program starts running, your program's menu bar replaces the THINK Pascal menu bar, and a bug spray can appears on the right side of the menu bar. You'll use the bug spray can to stop your program when you're debugging it.

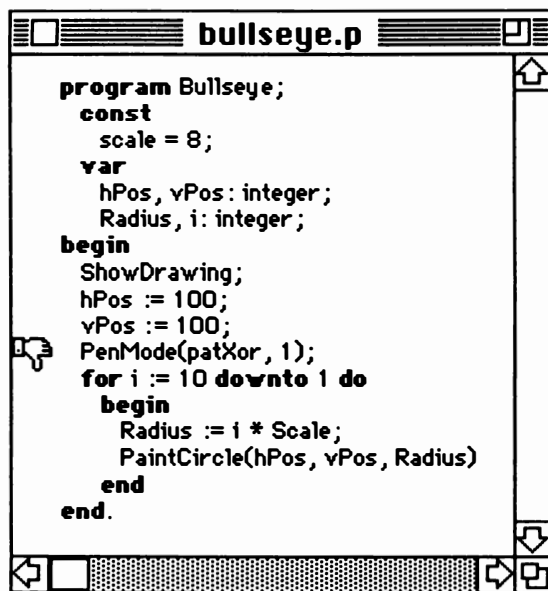
When Something Goes Wrong

If something goes wrong while THINK Pascal is compiling or linking your program, or if there is an error in your code, you'll see a bug box telling you what went wrong. Here's an example of a bug box:



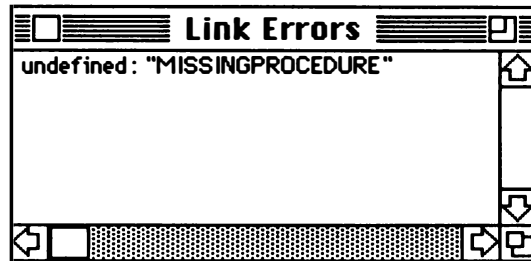
To get rid of the bug box, click the mouse or press the Enter or Return key. Of course, you still need to fix the bug!

If your program has an error in it, THINK Pascal points out the error with a "thumbs down" next to the offending line in the file's editing window.



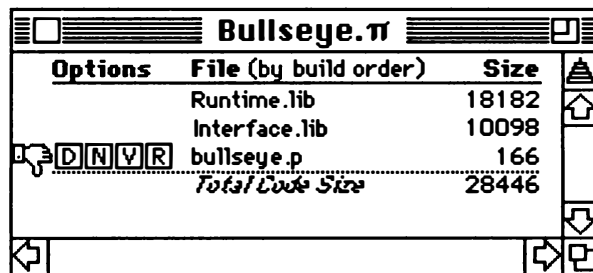
If the error happens while your program is running, and the offending file's editing window is open, THINK Pascal shows you the error the same way.

If THINK Pascal finds a problem while it's linking all the files in your project, it displays the message "Link Failed" and it shows you a Link Errors window. The messages in the Link Errors window tell you the names of undefined routines or the names of routines you've defined more than once.



If you want to know in which files THINK Pascal found the errors, use the **Check Link** command in the **Run** menu. When you use this command, THINK Pascal reports the file names as well as the link errors.

If the error happens while your program's running and the file's edit window is not open, the thumbs down will appear in the project window, next to the name of the appropriate file.



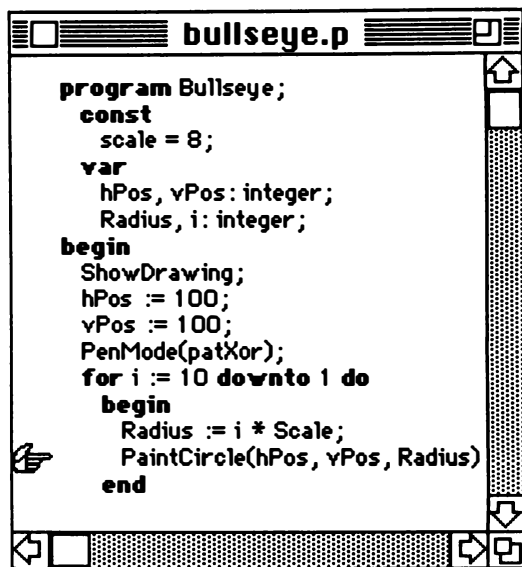
Appendix D contains a list of error messages you might see when your program is compiling, linking, or running.

Stopping Your Program

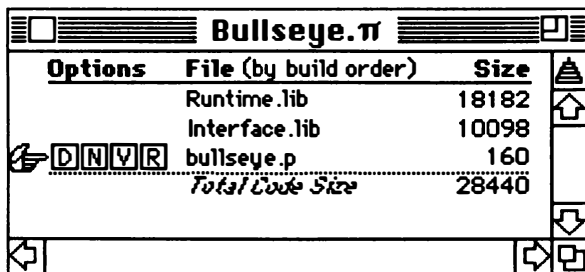
To stop your program while it's running, click on the bug spray can in the far right of the menu bar. Your program's menu bar disappears, and THINK Pascal's menu bar and windows come to the front so you can look at what your code is doing. You can also type Command-Shift-Period to stop your program.

Note: You can stop your program only when it's running code that was compiled with the Debug option on. If your program is running code that was compiled with the Debug option off, it will stop at the first place that was compiled with the option on.

If the file your program stops in has an open edit window, THINK Pascal points to the statement you're stopped at.



If the file your program stopped in does not have an open edit window, THINK Pascal points to the file in the project window.



To continue execution, choose any of the execution commands — **Go**, **Step Into**, **Step Over**, **Go-Go**, or **Step-Step** — from the **Run** menu. To restart the program from the beginning, choose **Reset** from the **Run** menu before choosing **Go**.

Note: THINK Pascal automatically resets your program when you edit a source file or modify your project. For more information, see “Restarting a Stopped Program” in Chapter 9.

Compiling, Building, and Linking Without Running

You can compile your program without running it. You might want to make sure that a file compiles properly, or that you called all the procedures with the right number of arguments, or that you didn't forget to define a procedure or function.

To check the syntax of the file in the active edit window, use the **Check Syntax** command. This is a quick way to make sure your Pascal syntax is correct. To recompile the file, choose **Compile**. To see the **Compile** command, hold down the Shift key as you select the **Run** menu.

To compile all the marked files, use the **Build** command. This command looks through your project to see which files have changed and recompiles them. If THINK Pascal finds any errors during compilation, you'll get a bug box and a thumbs down.

To make sure there aren't any undefined symbols or any symbols defined more than once, use the **Check Link** command. This command does a **Build** and then links all the files in your project. If there are any link errors in your project, the **Check Link** command tells you which files contain multiply defined or undefined routines.

Save Options

Before THINK Pascal runs your program, it usually asks you if you want to save files that you've edited. It's a good idea to save changes before you run. If you don't save your changes, and your program crashes, you'll lose any unsaved edits.

The Save Options section of the **Run** menu lets you specify whether THINK Pascal saves the changes to your edited files before running.

Option	Meaning
Auto-Save	Saves any changes automatically before running your program.
Confirm Saves	THINK Pascal asks you if you want to save changes before running your program.
Don't Save	Don't save any changes before running the program. You'll have to use the Save command in the File menu to save your changes.

The Text and Drawing Windows

The Text and Drawing Windows let you program even if you don't completely understand the Macintosh Toolbox. This list describes what you can do with these windows:

- You write to and read from the Text Window with the standard Pascal I/O routines, described in "9.0 Input/Output" in Chapter 17, "Language Reference."
- You draw to the Drawing Window with the Macintosh Toolbox graphics routines.

- You can manipulate the windows from your program with the routines described in “10.6 THINK Pascal Window Manipulation Procedures” in Chapter 17, “Language Reference.”
- You can print the windows with the **Print...** command.
- You can save the contents of the windows with the **Save As...** command. THINK Pascal saves the contents of the Text Window as a Teach Text file and the contents of the Drawing Window as an ObjectDraw file. (ObjectDraw is the drawing program you create in Chapter 4, “Tutorial: ObjectDraw.”)

Run Options

THINK Pascal lets you set up certain run-time environment settings. Choose **Run Options...** from the **Run** menu. You'll see this dialog box:

Run-time Environment Settings

Resources

Text Window

Memory

☐ **Use resource file:**
 for resources used by the project.

Text Window saves 5000 **characters**

☐ **Echo to the printer**

☐ **Echo to the file:**

Hello world. x = 811.79.

Monaco ▼
9 ▼

Stack size: 16 **kilobytes**

Zone size: 128 **kilobytes**

OK

Cancel

Resources

The Resources section of this dialog lets you specify the resource file your program uses. When you click on the check box, you'll see a standard file dialog box. Choose a resource file, and its name appears in the box.

Note: The resource file must be in the same folder as the project.

When you run your program in the THINK Pascal environment, THINK Pascal opens your resource file so you can access your resources. When you build your final application, THINK Pascal merges the resources from your resource file into the final file.

To learn how THINK Pascal builds your project see Chapter 12, "Building Projects."

Text Window

The Text Window section lets you specify how much text the window saves and the font to display in the text window. It also lets you redirect the output to the printer or to a file.

To send all the output from the text window to a printer, check the Echo to printer check box. To send all the output from the text window to a file, click on the Echo to file check box, and name the file.

Note: You can also print the Text Window with the **Print...** command and save its contents in a file with the **Save As...** command.

Memory

The Memory section lets you specify how much memory your program gets while it's running in the THINK Pascal environment. THINK Pascal uses the stack for local variables and parameters. In most cases, 16K is plenty. But if a routine in your program uses a lot of local variables, or if your program does a lot of deep recursion, you might want to increase this value.

The Zone size value tells THINK Pascal how much memory to set aside for your program's heap. The heap contains your program's code as well as any dynamically allocated data structures. Depending on the size of your program and on how much memory you allocate dynamically, you might want to make this value larger or smaller.

If you're running with MultiFinder, you may want to increase the size of the THINK Pascal partition when you increase the zone and stack sizes.

Note: When you build your application, THINK Pascal uses the stack size value you specify. The zone size value applies only while you're running in the THINK Pascal environment.

For more information on stacks and zones, see Chapter 13, "Assembly Language," *Inside Macintosh II*, Chapter 1, "The Memory Manager," and *Inside Macintosh VI*, Chapter 28, "The Memory Manager."

Debugging Programs

9

Introduction

It's almost unheard of to write a program that runs perfectly the first time. THINK Pascal has several tools that help you track down bugs that creep into your program. This chapter shows you how to use them.

The THINK Pascal editor catches syntax errors as you type in your program. And the project manager catches compiler and link errors such as undeclared variables and multiply defined symbols. These kinds of errors happen while you're still writing your program.

There are other errors that turn up while your program is running. You might discover that your program tries to divide by zero, or that it's trying to store a value into the seventh element of a six element array. THINK Pascal helps you catch **run time errors** by displaying a thumbs down icon next to the line that contains the error.

The most insidious errors are **errors of intention**. Your program runs, but it doesn't do the right thing. These errors occur when you forget to account for a special case or when the code you write doesn't do what you think it does.

No compiler or development system can catch these errors of intention for you, but THINK Pascal gives you a set of tools that makes finding errors easier and faster. THINK Pascal lets you watch your program as it runs. You can step through each line of your program, stop at any point, and examine and modify any variable. If you like, you can make temporary fixes to your program without recompiling, so you can see if your fixes really work.

Note: THINK Pascal also provides a more advanced debugging tool — LightsBug. See Chapter 14 after you read this chapter to learn more about LightsBug.

Topics covered in this chapter

- Debugging in THINK Pascal
- Following the finger
- Stepping through your program
- Stop Signs
- Restarting a stopped program
- The execution commands
- The Observe window
- The Instant window
- Examining compiled code

Debugging in THINK Pascal

The basic trick of debugging is to find out what your program is *really* doing instead of what you thought it was supposed to do. The first thing to do is to run your program in a controlled way, so you know exactly what happens when your program is running. THINK Pascal lets you control program execution several ways.

- You can use the **Step Into** or **Step Over** command to step through your program one statement at a time. The **execution finger** appears next to the line about to be executed. Your program stops after each statement.
- You can use the **Step-Step** command to let THINK Pascal step through each statement automatically. To see this command, hold down the Option key and choose the **Run** menu. This command steps through your program a statement at a time, pausing briefly between statements to update the LightsBug and Observe windows.
- You can insert one or more Stop Signs anywhere in your code. If you use the **Go** or **Step-Step** commands to run your program, it will stop whenever it encounters a Stop Sign. With the **Go-Go** command, the program only pauses at a Stop Sign to update the LightsBug and Observe windows.
- You can click on the bug spray can to stop the program at any point.
- You can type Command-Shift-Period to stop your program.

Note: To use the THINK Pascal debugging tools, make sure that you've compiled your files with the Debug option on.

Sometimes, just slowing program execution is enough. By watching the output of the program at the same time as you watch the source lines being executed, you can see what's going on. Often, though, you need to know the values of variables and expressions, not just which statement is being executed. For this kind of debugging, you can use the Instant and Observe windows.

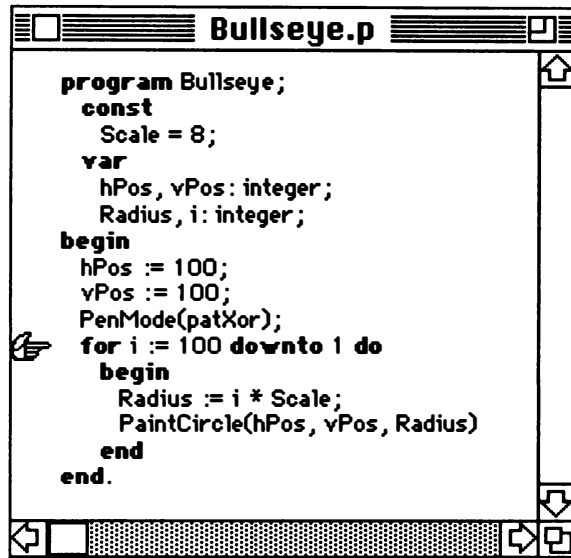
The Observe window lets you type in (or copy from your source file) any number of valid Pascal expressions (including variable names) whose value you want to know about. THINK Pascal updates the values in the Observe window when your program is stopped or whenever your program pauses between statements.

If your program is already stopped, you can find out the current value of a variable or expression by typing it into the Observe window and pressing the Enter key.

The Instant window takes things one step further. You can actually change the value of a variable or expression, or insert additional statements into your program, and have them executed on the spot — in the context of the program at the point at which it stopped.

Following the Finger

Whenever your program is stopped or paused, the **execution finger** points to the statement about to be executed.



If the file you're stopped in doesn't have an open edit window, the execution finger points to the file in the project window.

Note: To see the execution finger, the file must be compiled with the Debug option on. If you compile a file with the Debug option off, you'll never see the execution finger in it, even if the file has an edit window.

When the finger moves from one window to another, the new window becomes the active window. If you want to keep another window frontmost as you step through your program, you can turn off the **Auto-Show Finger** option in the **Debug** menu. For instance, if you want to keep the Observe window from being covered up as the edit windows become active, you would turn **Auto-Show Finger** off. The downside of turning this option off is that if an edit window is obscured, you won't be able to see the execution finger.

Note: It's a good idea to keep **Auto-Show Finger** off when you're debugging how your application handles update events.

When your program is stopped, you can scroll through the editing window or switch to other windows to look at other parts of your program. The **Show Finger** command in the **Debug** menu makes the window with the execution finger the active window and scrolls until the finger is visible.

Stepping Through Programs

To run a program one statement at a time, choose either **Step Into** or **Step Over** from the **Run** menu. THINK Pascal executes the first statement in the program, updates the Observe and LightsBug windows, and then stops. Choosing either command again continues with the next statement. The **Step Into** command steps into procedure and function calls, moving the execution finger into the procedure or function called. The **Step Over** command steps over function and procedure calls, keeping the execution finger within the current block.

To exit the procedure or function you're in, choose the **Step Out** command. THINK Pascal continues executing until it returns from the current routine, and then stops. You'll find this command especially useful if you accidentally step into a routine with the **Step Into** command.

Stepping through a program can take a long time, so you'll normally use the step commands when you want to take a very close look at a part of your program. The **Step-Step** command is the automatic version of **Step Into**. It executes your program faster than **Step Into**, but slowly enough so you can watch the program's execution. THINK Pascal updates the Observe and LightsBug windows between statements. To see this command, hold down the Option key and choose the **Run** menu.

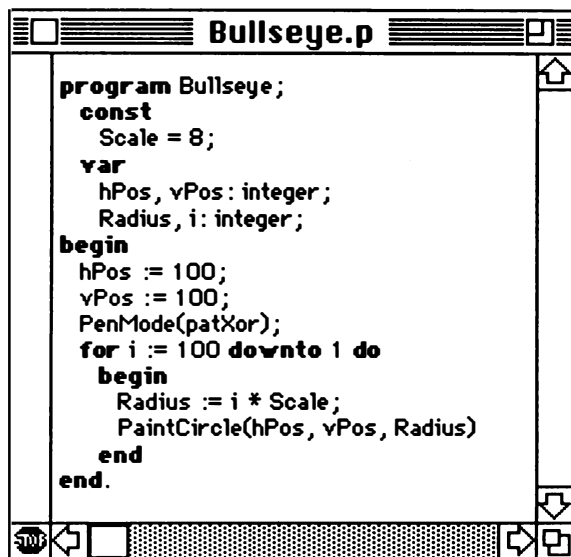
Stop Signs

Sometimes even stepping automatically is too slow. You may want to run your program at full speed up to a specific point, and then stop it. Then you can either start stepping or tracing, possibly using the Observe window.

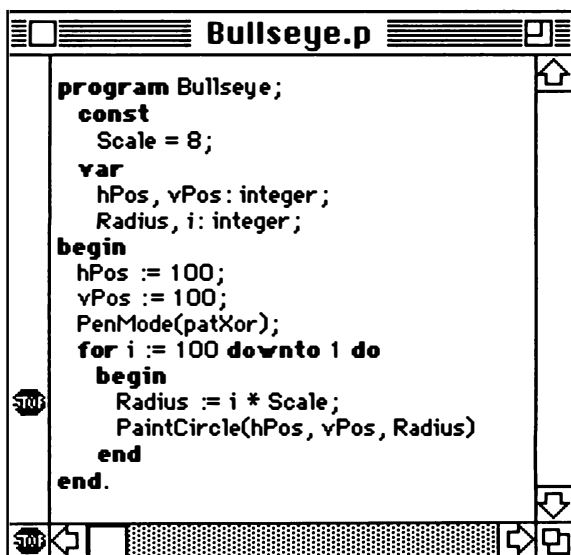
Stop Signs let you specify the statements where you want to interrupt execution of your Pascal program. If you run your program with **Go** or **Step-Step**, the program runs until it reaches the Stop Sign, and then it stops. The statement at the Stop Sign is *not* executed.

The **Go-Go** command is the automatic version of **Go**. It pauses momentarily at each Stop Sign just long enough to update the values in the Observe and LightsBug windows. To see this command, hold down the Option key and choose the **Run** menu. If there are no Stop Signs in the editing window, **Go-Go** is equivalent to **Go**.

To put Stop Signs in your programs, choose the **Stops In** option from the **Debug** menu. A Stop Sign appears in the lower left-hand corner of the active editing window, and a vertical bar appears along the left side.



When you move the cursor into the bar on the left side of the edit window, the cursor turns into a Stop Sign. Move the Stop Sign until it is directly to the left of the statement you want execution to stop before, and click. A Stop Sign remains in the stop bar like this:



You can put in as many Stop Signs as you want, but only on lines that contain Pascal statements. You can't put a Stop Sign before a comment, a declaration, a label, or an empty statement.

To use Stop Signs in a file, the project must contain debug information for the file. If you try to put a Stop Sign in a file that has the Debug option turned off, THINK Pascal displays a dialog asking if you want to turn the option on. If you click Yes, THINK Pascal turns the option on and inserts the Stop Sign. If you click No, THINK Pascal leaves the option off and doesn't insert the Stop Sign.

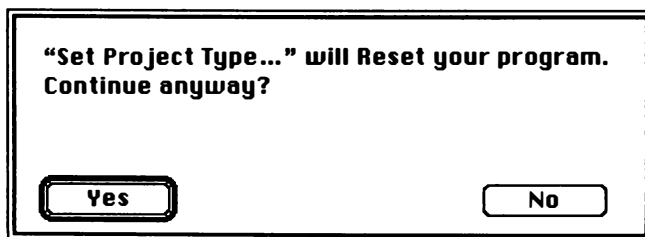
If you turn the **Stops In** option off, THINK Pascal hides and ignores all your Stops Signs. If you turn the option on again, the Stop Signs reappear, and THINK Pascal will stop at them when you run your program. If you save a file with the Entire Document option, any Stop Signs placed in the file will be saved as part of the file.

To remove a Stop Sign, click on it. To remove all the Stop Signs from the active window, choose **Pull Stops** from the **Debug** menu. To remove all the Stop Signs from all the open windows, choose **Pull All Stops**. To see **Pull All Stops**, hold down the Option key and select the **Debug** menu.

Restarting a Stopped Program

When you stop a program with the bug spray can or with a Stop Sign, you don't have to start it all over from the beginning. If you choose one of the execution commands — **Go**, **Step Into**, **Step Over**, **Step Out**, **Go-Go**, or **Step-Step** — from the **Run** menu, the program continues with the next statement.

THINK Pascal will reset your program to start again from the beginning when you edit a source file, choose a command that can modify your project (such as **Add File**, **Set Project Type**, or **Compile Options**), move a file in your project window, or choose the **Reset** command from the **Run** menu. If you try to change a source file or project, THINK Pascal warns you with a dialog like this:



If you answer Yes, THINK Pascal resets your program and proceeds with the command or editing. If you answer No, THINK Pascal cancels the command or editing. If you don't want to see these warnings, select the **Quietly Auto-Reset** option from the **Debug** menu.

Note: If you reset your program by choosing **Reset** or moving a file in your project window, THINK Pascal does not warn you with a dialog.

The Execution Commands

THINK Pascal gives you several execution commands, each of which runs your program a different way. This chart summarizes what the different execution commands in the **Run** menu do:

Command	Description
Go	Runs your program, stopping at Stop Signs (or break points)
Step Over	Executes the next statement, without stepping into procedures and functions
Step Into	Executes the next statement, stepping into procedures and functions.
Step Out	Steps out of the current procedure or function and stops

The Option key changes a few of the execution commands into automatic versions. **Go** becomes **Go-Go**, and **Step Into** becomes **Step-Step**. The automatic versions pause when their counterparts would stop. Choosing them is like choosing **Go** or **Step Into** over and over again. To see the automatic commands, hold down the Option key as you choose the **Run** menu.

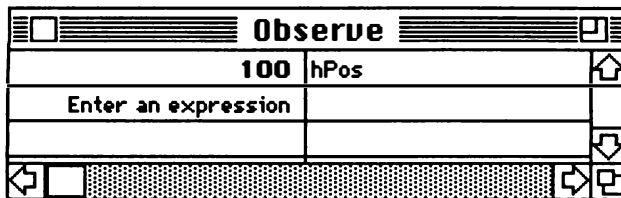
Command	Description
Go-Go	Runs your program, pausing at Stop Signs
Step-Step	Runs your program, pausing after each statement and stopping at Stop Signs.

The Observe window

The Observe window lets you track the value of a variable or expression while your program is running. This window takes the place of the old debugging technique of inserting `writeln` statements into your program to print out the values of variables or expressions.

Note: In fact, only values that can be printed with `writeln` can be observed.

To type an expression in the Observe window, choose **Observe** from the **Debug** menu.



You can type in any valid Pascal expressions in the cells to the right of the vertical bar in the middle of the window. You can copy an expression from your program, from the Instant window, or from another Observe window cell and paste it into the Observe window. The expressions will be evaluated and the results appear in the left cells whenever you press the Enter key or when your program stops or pauses.

The Observe window uses the point at which your program stopped for the context of its expressions. Once the program finishes, the Observe window displays the message "No context."

It is relatively harmless to type invalid expressions or expressions with run-time errors in the Observe window. An abbreviated error message appears in the left hand cell.

Some handy tips for using the Observe window:

- To observe floating point numbers with precision, use the `stringof` function described in Section 10.9.4 of Chapter 17.
- You can use Observe to convert hexadecimal numbers to decimal. Type '\$' followed by the hex number. The equivalent decimal value appears in the left cell..
- You can convert decimal numbers to hexadecimal by casting them to pointers. For example `Pointer(-1)` yields `FFFFFFFF`.
- You can use the Observe window to examine variables and expressions if execution is stopped because of a run-time error. This powerful debugging feature can help you figure out the cause of the run-time error.

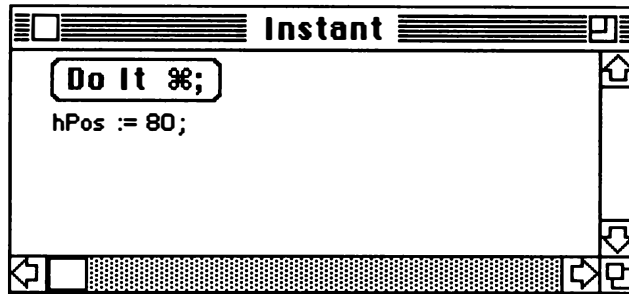
This Observe window illustrates some of the tips:

Observe	
3.14159265358979	stringof(pi : 1 : 14)
3.141593	pi
-1330	\$FACE
FFFFFFFF	Pointer(-1)

THINK Pascal lets you save the contents of the Observe window. When the window is active, choose the **Save As...** command from the **File** menu. To open a saved Observe window, use the **Open...** command in the **File** menu. The saved window replaces the contents of the current window.

The Instant window

Any time your program is stopped, you can use the Instant window to execute any THINK Pascal statements. Choose **Instant** from the **Debug** menu if the Instant window is not visible. Type in and edit any Pascal statements in the window just as you would in an edit window. Then click the **Do It** button to execute the statements.



Note: If you use Command-; (Command-Semicolon), you can run the code in the Instant window without making it the active window.

Almost any Pascal statement is valid in the Instant window if it is legal at the point in the program at which execution is stopped. The statements in the Instant window are executed as if they were inserted into your program at the point where execution paused. You can even use the Instant window after a run-time error occurs. You might be able to fix the cause of the error and continue executing your program.

Note: You cannot use these statements in the Instant window: `goto`, `exit`, or interactive I/O like `readln`.

The Instant window lets you try out new code. Type the new code in the Instant window and execute it. If it doesn't work, try something else. You can do this until your code works. Then copy the new code from the Instant window and paste it into the program.

Note: If you call a procedure containing Stop Signs from the Instant window, the Stop Signs are ignored.

THINK Pascal lets you save the contents of the Instant window. When the window is active, choose the **Save As...** command from the **File** menu. To open a saved Instant window, use the **Open...** command in the **File** menu. The saved window replaces the contents of the current window.

Examining Compiled Code

A special mode of the execution commands (**Go**, **Step Into**, **Step Over**, **Step Out**, **Go-Go**, and **Step-Step**) lets you use a low level debugger like TMON or Macsbug to examine compiled code. To examine the compiled code for a particular statement in your program, hold down the Shift key and choose an execution command from the **Run** menu. Instead of executing the next statement,

THINK Pascal will drop into the low level debugger just before an RTS instruction that leads to your code.

To examine your compiled code in TMON, choose an execution command and open an assembly window anchored to the PC. In Macsbug, type T, then IL PC to display the code. If you compiled your program with the Names option on, you'll see the name of the subroutine in the low level debugger. When you're through examining the code, just choose the Exit command in TMON or the G command in Macsbug.

Note: If you don't have a low-level debugger, choosing an execution command with the Shift key down is the same as choosing the command without the Shift key down.

Units and Libraries

10

Introduction

This chapter goes into more detail about the components of a Pascal program. You'll learn how to create and use units and libraries.

A **unit** is a file that is part of a larger program. Units usually handle a certain kind of action or deal with a particular data structure. A unit can be generic, but more often it is tailored for its specific role in your program. Other programming languages use the terms module or package instead of unit.

A **library** is a collection of precompiled functions and procedures. Like units, libraries usually deal with a kind of action or handle some data structure, but libraries are more generic. You can use libraries in any program.

Topic covered in this chapter

- Using units
- Writing units
- Using libraries
- Writing libraries

Using Units

A THINK Pascal program usually consists of a main program and several units. A unit is a collection of constants, types, variables, and procedures and functions that you use in your program to handle a particular set of actions or data structures. For instance, in a drawing editor, a unit might handle drawing shapes. In fact, this is what the ODTShape unit of the ObjectDraw example in your THINK Pascal package does.

To use a unit in your program or in another unit, you put the unit name in a **uses** clause in the program or unit. For example, if your program were to use a unit called *Conversions*, this is what the **uses** clause in your program would look like:

```
program Gazelle;

uses
    Conversions;
    ... types and vars ...

begin
    ... the program ...
end.
```

Units let you break up your program into small, manageable pieces. Usually, you'll make changes only to one unit a time, and then test the changes. Because THINK Pascal recompiles and relinks only the units that have changed, you'll be able to work faster if you use units.

When you view the project window by build order, all units must be listed in the order you use them. In other words, files that are used must precede the files that use them, with the main program always last. See Chapter 7, "Working with Projects," to learn how to change the build order.

Writing a Unit

A unit has two parts: an **interface** and an **implementation**. The interface section declares all the constants, types, variables, procedures, and functions that other files can see. The implementation section contains the code for the visible procedures and functions. The implementation section can also define its own constants, types, variables, procedures, and functions, but these will be private to the unit.

A unit begins with the keyword **unit** followed by a unit name. The name of the unit is not necessarily the same as the file name, but it's less confusing if you use the same name for them. Just remember that when you use a unit, you give the unit name in the **uses** clause, not the file name.

Note: You can use the **View Options...** dialog to have THINK Pascal display the name of the unit as well as the file name in the project window. The unit name shows only after the unit has been compiled. See Chapter 7, "Working with Projects," for more information.

The interface section follows the unit name. The interface section contains the public constants, types, variables, procedures, and functions that your unit defines. For procedures and functions, you provide only the headers (along with the parameter lists).

The implementation defines all the private constants, types, variables, procedures, and functions. The implementation section also defines the bodies of the procedures and functions you declared in the interface section. The procedure and function headings in the implementation section can

contain parameter lists only if they match the parameter lists in the interface section. Here's an example of a unit.

```

unit Calculations;           { Sample Unit - doesn't do much }

interface
  uses
    SANE;                     { The RoundPre type is defined in SANE }
  var
    DisplayWidth : integer; { This variable is exported }

{ Public procedures and functions. }
{ The bodies are in the implementation section }

  procedure Initialize (Precision: RoundPre);
  function Square (aNumber: longint) : longint;

implementation
  const
    ConstantPrivateToCalculations = 42;
  var
    RoundPrecision: RoundPre;

  procedure Initialize (Precision: RoundPre);
  begin
    RoundPrecision := Precision;
    SetRound (RoundPrecision);
  end;

  function Square (aNumber: longint) : longint;
  begin
    Square := aNumber * aNumber;
  end;

  procedure PrivateToCalculations(x: integer);
  begin
  end;
end.

```

The program that uses this unit would be in another file and would look something like this:

```

program Gazelle;

    uses
        SANE, Calculations;      { The program uses SANE because }
                                   { the TowardZero constant is      }
                                   { defined there, and because       }
                                   { Calculations uses it in its      }
                                   { interface part                   }

    var
        aNum : longint

    ... { more declarations }

begin
    Initialize(TowardZero);      { Public proc in Calculations unit }
    aNum := Square(1961);
end.

```

In this example, the main program would not be able to call the procedure `PrivateToCalculations` or use the constant `ConstantPrivateToCalculations` because both are defined only in the implementation section of the unit and not in the interface.

The uses clause

The **uses** clause specifies the dependencies between the files that make up a program. If your unit uses something that's defined in another unit, that other unit needs to appear in your unit's **uses** clause. Everything in the used unit's interface clause appears as though it's in the unit that contains the **uses** clause. THINK Pascal uses this clause to determine which files are affected by an edit. Whenever you change the interface section of a unit, THINK Pascal recompiles all the files that use it.

The "USES Extensions" option lets you choose between two ways that THINK Pascal can interpret your **uses** clauses. To see the option, choose **Compiler Options...** in the **Project** menu.

If you turn on the "USES Extensions" option, THINK Pascal lets you use these features:

- **Propagated uses.** If your unit uses other units, any unit that uses your unit also uses those units automatically.
- **Implementation uses.** You can put a uses clause in a unit's implementation section.

Propagated **uses** lets you shorten your **uses** clauses. If your unit uses a lot of other units, any unit that uses your unit automatically uses those other units. THINK Pascal propagates the other units from your unit. For example, say you write this:

```
unit MyShapesUnit;
interface
  uses
    Circles, Squares, Triangles, Rectangles;
  . . .
```

This unit uses MyShapeUnit and all the units MyShapeUnit uses (Circles, Squares, Triangles, Rectangles):

```
unit MyPictureUnit;
interface
  uses
    MyShapeUnit;
  var
    aSquare: RedSquare;           { Defined in Squares unit }
    aTriangle: LoveTriangle;      { Defined in Triangles unit }
    aCircle: TrafficCircle;       { Defined in Circles unit }
  . . .
```

Even though MyPictureUnit doesn't contain Circles, Squares, and Triangles in a **uses** clause, it can use what they declare. THINK Pascal propagates those units into MyPictureUnit from MyShapeUnit.

Implementation **uses** lets you put your **uses** clause close to the code that needs it. (Your code needs a **uses** clause if it refers to something that's defined in a unit in the **uses** clause.) If a unit's implementation section needs a unit but the interface section doesn't need it, you can put that unit in a **uses** clause in the implementation section. If the interface section needs the unit, it must be in a **uses** clause in the interface section. For example, here is the rest of MyPictureUnit:

```
unit MyPictureUnit;
interface
  uses
    MyShapeUnit;
  var
    aSquare: RedSquare;           { Defined in Squares unit }
    aTriangle: LoveTriangle;      { Defined in Triangles unit }
    aCircle: TrafficCircle;       { Defined in Circles unit }
  procedure DrawPicture;

implementation
  uses
    MyPrivateColorsUnit;
  procedure DrawPicture;
  var
    aColor: Mauve;
```

```
begin
    . . .
end;

end.
```

Since only the implementation section needs `MyPrivateColors` unit, you can put it in a implementation **uses** clause.

THINK Pascal does not propagate implementation **uses** clauses. It propagates only interface **uses** clauses. This fact helps you create private units that you can use but that other people can't. For example, say someone writes the unit `AGalleryUnit` that uses `MyPictureUnit`. That person can use what's defined in `MyPictureUnit`, but not what's defined in `MyPrivateColorsUnit`.

If you turn off the "USES Extensions" option, you can't use propagated uses or implementation uses. For example, this is how you would need to rewrite `MyPictureUnit`:

```
unit MyPictureUnitII;
interface
uses
    Circles, Squares, Triangles, MyShapeUnit, MyPrivateColorsUnit;
var
    aSquare: RedSquare;           { Defined in Squares unit }
    aTriangle: LoveTriangle;      { Defined in Triangles unit }
    aCircle: TrafficCircle;       { Defined in Circles unit }
    . . .
```

Since THINK Pascal does not propagate the units used in `MyShapesUnit`, you need to explicitly include `Circles`, `Squares`, and `Triangles` in the **uses** clause. Notice that you don't need to include the `Rectangles` unit since you don't use it. Since you can't use implementation **uses**, you must put `MyPrivateColorUnit` in an interface **uses** clause. Anyone who uses `MyPicutureUnitII` will be able to use whatever's in that private unit.

Using Libraries

A library is like a unit in that it contains procedures and functions to perform an action or to manipulate a particular data structure. But unlike units, a library is made up of compiled code and can consist of more than one Pascal file.

To use a library, you simply add it to your project with the **Add File...** command in the **Project** menu. Most of the time, a library has an associated **interface** file. The interface file is a unit that contains the type declarations and calling sequences for the procedures and functions defined in the library. To add the interface file, use the **Add File...** command.

Your THINK Pascal package comes with several libraries. The next chapter tells you what's in each library.

Writing Libraries

Writing a library is like writing a unit. In fact, a library is made up of one or more units. The difference between adding a library into your project and adding the units that comprise the library into your project is that a library consists of compiled code *whereas* you would have to compile each unit. In most cases, you won't even have the source code for libraries.

To create a library, make a new project and add all the units that comprise it. Then use the **Remove** command in the **Project** menu to remove `Interface.lib` and `Runtime.lib`. If you leave these two libraries in your library, any project that uses them and your library will have multiply defined symbols. Then use the **Build Library...** command in the **Project** menu to create your library. A standard dialog box will appear asking you to name the library. By convention, libraries end in `.lib`. THINK Pascal builds the library in a format that is compatible with MPW `.o` files.

Note: Even though they use the same format, compilers that create `.o`-compatible files can't use THINK Pascal libraries, and THINK Pascal can't use most of their `.o` files. For more information, see "Using `.o` Files," Appendix C.

Your library must conform to these two constraints:

- It contains at least one file .
- It contains only units, no main program.

Writing the Interface file

Because the library consists of compiled code, there's no way for your program to know how to call the procedures and functions in it. You need to create the **interface file** to let the program that uses your library know what the routines are called and how to call them. The interface file also defines the public types, variables, and classes you use in the library.

For instance, suppose you created a library, `Names.lib`, that contains types, variables, procedures, and functions that deal with names in a list. Your interface file might be called `NamesIntf.p` and look like this:

```
unit NamesIntf;
interface
  type
    { This is a type your program can use. The library may }
    { define private types that your program can't "see" }
    Str32 = string[32];
    NameRec = record
      FirstName: Str32;
      LastName: Str32;
    end;
  var
    {$J+}
    ThisName: NameRec;
    NextName: NameRec;
    {$J-}

    function AddName(first: Str32, last: Str32): integer;
    function FindName(ix: integer; var theName: NameRec): Boolean;
    procedure DelName(ix: integer);

implementation
  { These declarations are optional because THINK Pascal }
  { defaults them to external. }
  function AddName(first: Str32, last: Str32): integer;
  external;

  function FindName(ix: integer; var theName: NameRec): Boolean;
  external;

  procedure DelName(ix: integer);
  external;
end.
```

To use the library, you would add `NamesIntf.p` and `Names.lib` to your project. The unit that calls procedures and functions from `Names.lib` would have this clause in it:

```
uses NamesIntf;
```

The interface file looks just like a unit. The interface section shows what procedures and functions are public and how they should be called. The implementation section is a little bit different. Instead of defining the bodies of the procedures and functions, it contains the routine headers and then the directive `external`.

Note: If there is no definition of a procedure or function declared in the implementation part, THINK Pascal assumes that the procedure or function is external.

It's a good idea to declare these routines external explicitly to make your intentions clear to others who use your libraries.

To declare public variables in an interface file, you must use the External Variable directive { \$J± }, which turns on and off the External Variable option. This option tells THINK Pascal not to allocate space for the following variables. { \$J+ } turns on the option on, and { \$J- } turns it off. If you don't use this directive in your interface file, THINK Pascal declares space for your variables twice: once in the library and again in the interface file.

Note: For more information on using compiler directives and the External Variable directive, see Chapter 15, "Compiler Directives."

To declare public classes in an interface file, you must also use the External Variable directive { \$J± }. When THINK Pascal comes across a class declaration, it generates code to resolve calls to its methods. The External Variable directive tells THINK Pascal not to generate that code. If you don't use this directive in your interface file, THINK Pascal generates that code twice: once in the library and again in the interface file. For example, this class declaration will not generate that code:

```
type
{$PUSH}
{$J+}
  MyWindow = object (Window)
    windowData: Handle;
    subWindow: ToolWindow;

    procedure Hit (where: Point);
    override;
    procedure Draw;
    override
  end;
{$POP}
```

Note: At the end of the variable or type declaration section, you must turn off the External Variable option (with either { \$J- } or { \$POP }).

The external directive

The external directive tells the compiler that the body of the procedure or function is not there, but that it should try to find it when the linker tries to link the program.

Usually, you use the external directive in the implementation section of interface files for libraries or to access procedures and functions that were originally written in assembly language. But you can use it to describe the calling sequence for a procedure or function that isn't in a used unit.

Note: To learn more about using assembly language routines in THINK Pascal, see Chapter 13, "Assembly Language."

For example, if you added the library `Names.lib` from the example above into your project, and you didn't want to add the interface file, you could just write:

```
function AddName(first : Str32, last : Str32) : integer;  
external;
```

in the file that uses `AddName` to define the calling sequence for the `AddName` function.

Generally speaking, you should limit use of the `external` directive to an interface file.

Using Predefined Routines

11

Introduction

THINK Pascal predefines all of the standard Pascal procedures and functions as well as the most common Macintosh Toolbox routines. Your THINK Pascal package also includes several libraries for those routines that aren't built into the compiler. This chapter shows you what you need to do to access the standard Pascal procedures like `write`, `writeln`, `abs`, etc. and how to access the Macintosh Toolbox routines described in *Inside Macintosh*.

Topics covered in this chapter

- Calling standard Pascal routines
- Calling Macintosh Toolbox routines

Calling Standard Pascal Routines

THINK Pascal provides a large number of predefined procedures and functions. When you create a new project, two libraries are automatically added to it. The library `Runtime.lib` contains all of the standard Pascal routines like `writeln`, `readln`, and `abs`, as well as routines that the compiler uses for sets, 32-bit multiplications, and so on. `Runtime.lib` also contains some procedures and functions for compatibility with Symantec's (formerly Apple's) Macintosh Pascal. For a complete description of the THINK Pascal predefines, see Sections 9 and 10 of the Chapter 17, "Language Reference."

Depending on the type of project you're building you may use different versions of `Runtime.lib`. The default `Runtime.lib` that's included in every THINK Pascal project you create is rather large because it's supporting all of the Pascal input/output and the code that manages the Text window.

If you're writing a Macintosh application, and you're not using any of the Pascal input/output routines, you can use a smaller version of `Runtime.lib` called `μRuntime.lib`. This version of the library does not contain any Pascal input/output routines, so you can't use the Text or Drawing windows.

If you're writing a desk accessory, device driver, or code resource, you cannot use `Runtime.lib` or `μRuntime.lib` because both libraries reference their globals through register A5. For code resources, use `RSRCRuntime.lib`. For desk accessories and drivers, use `DRVRRuntime.lib`. These libraries are similar to `μRuntime.lib` and do not contain any Pascal input/output routines, so you can't use the Text or Drawing windows. Unlike `μRuntime.lib`, they use register A4 to access their globals. The next chapter shows you how to build these kinds of projects.

Calling Macintosh Toolbox Routines

THINK Pascal lets you use all the Macintosh Toolbox routines described in *Inside Macintosh I-VI* including those marked [Not in ROM]. To use the Toolbox routines, call them exactly as they appear in *Inside Macintosh*.

Most of the Toolbox routines in *Inside Macintosh I-VI* are predefined in THINK Pascal, so you don't have to do anything special to use them. The constants, types, and routines are built into THINK Pascal. The library `Interface.lib` contains the "glue" code for Toolbox routines that aren't stack-based.

THINK Pascal generates in-line calls for all of the stack-based traps. For the register-based traps, routines marked [Not in ROM], and for routines that work off dispatched traps, THINK Pascal uses the "glue" code in `Interface.lib` so you can call them as Pascal procedures and functions.

For the less common routines, you need to add specific libraries and interface files to your project.

Note: This section assumes you already know what interface files and libraries are. If you need to learn about them, see Chapter 10, "Units and Libraries."

Toolbox Interfaces

The Toolbox interfaces are organized along the lines of the *Inside Macintosh* chapters. Each file contains the routines described in a different chapter of *Inside Macintosh*. For example:

This file...	Contains the routines described in this chapter...
<code>Events.p</code>	<i>Inside Macintosh I</i> , Chapter 8, "The Toolbox Event Manager"
<code>Windows.p</code>	<i>Inside Macintosh I</i> , Chapter 9, "The Window Manager"
<code>Menus.p</code>	<i>Inside Macintosh I</i> , Chapter 11, "The Menu Manager"

The following interfaces are built into THINK Pascal. You don't need to include them in your project or your unit's `uses` clause.

<code>Controls.p</code>	<code>Desk.p</code>	<code>Devices.p</code>
<code>Dialogs.p</code>	<code>DiskInit.p</code>	<code>Errors.p</code>
<code>Events.p</code>	<code>Files.p</code>	<code>Fonts.p</code>
<code>GestaltEqu.p</code>	<code>Lists.p</code>	<code>MacPrint.p</code>
<code>Memory.p</code>	<code>MemTypes.p</code>	<code>Menus.p</code>
<code>OSEvents.p</code>	<code>OSIntf.p</code>	<code>OSUtils.p</code>
<code>Packages.p</code>	<code>PackIntf.p</code>	<code>PaletteMgr.p</code>
<code>PickerIntf.p</code>	<code>QDOffscreen.p</code>	<code>Quickdraw.p</code>
<code>Resources.p</code>	<code>Scrap.p</code>	<code>SCSIIntf.p</code>
<code>SegLoad.p</code>	<code>Sound.p</code>	<code>StandardFile.p</code>
<code>TextEdit.p</code>	<code>ToolIntf.p</code>	<code>ToolUtils.p</code>
<code>Types.p</code>	<code>VideoIntf.p</code>	<code>Windows.p</code>

Do not redeclare a type that is in one of the built-in interfaces. If you redeclare one of those types and try to call a Toolbox routine that requires an argument of that type, THINK Pascal gives you the

error "Type incompatibility between an actual and formal value parameter." You get this error even if you redeclare the type exactly as it is in the built-in interface. For example, this code will give you an error:

```

program test;
  type
    rect = record
      top, left, bottom, right: integer
    end;
  var
    r: rect;
begin
  SetRect(r, 10, 5, 200, 100);
  ...
end.

```

These interfaces are *not* built into THINK Pascal. If you use them, you must include them in your project and your unit's **uses** clause.

ADSP.p	AIFF.p	Aliases.p
AppleEvents.p	AppleTalk.p	Balloons.p
CommResources.p	Connections.p	ConnectionTools.p
CRMSerialDevices.p	CTBUtilities.p	DatabaseAccess.p
DeskBus.p	Disks.p	Editions.p
ENET.p	EPPC.p	FileTransfers.p
FileTransferTools.p	Finder.p	FixMath.p
Folders.p	Graf3D.p	HyperXCmd.p
Icons.p	Language.p	MIDI.p
Notification.p	ObjIntf.p	Palettes.p
PasLibIntf.p	Picker.p	PictUtil.p
Power.p	PPCToolBox.p	Printing.p
PrintTraps.p	Processes.p	Retrace.p
ROMDefs.p	SANE.p	Script.p
SCSI.p	Serial.p	ShutDown.p
Slots.p	Sound.p	SoundInput.p
Start.p	Strings.p	SysEqu.p
Terminals.p	TerminalTools.p	Timer.p
Traps.p	Video.p	

For each of the built-in interfaces, there is a dummy interface file that contains no definitions. These make porting from other development systems easier, since you don't need to modify a unit that includes one of these interfaces in its **uses** clause.

Calling QuickDraw routines

The standard QuickDraw routines described in *Inside Macintosh I* and the color QuickDraw routines described in *Inside Macintosh V* are predefined in THINK Pascal. You don't need to do anything special to use them.

THINK Pascal lets you use an alternate parameter list for the QuickDraw graphics operations on Rects, RoundRects, Ovals and Arcs. Instead of using a temporary Rect variable, you can supply the rectangle coordinates in the parameter list. So you can replace this:

```
var
  r:rect;
begin
  SetRect (r, left, top, right, bottom);
  FillRect (r, dkGray);
end;
```

with this:

```
FillRect (top, left, bottom, right, dkGray);
```

Note: The order of the arguments to SetRect is different from the order of the arguments to the QuickDraw routines. The alternate parameter list uses the same order as the Rect record declaration.

You can use the alternate parameter list with these 20 procedures:

EraseRect	EraseRoundRect	EraseOval	EraseArc
FillRect	FillRoundRect	FillOval	FillArc
FrameRect	FrameRoundRect	FrameOval	FrameArc
InvertRect	InvertRoundRect	InvertOval	InvertArc
PaintRect	PaintRoundRect	PaintOval	PaintArc

The function GetMouse, which usually takes a point as a parameter, can also take two parameters:

```
procedure GetMouse(var h : integer; var v : integer);
```

Calling AppleTalk routines

If your application uses AppleTalk routines, you should be aware that there are now two versions of AppleTalk. Apple calls the old version (described in *Inside Macintosh II*) the **alternate** set. The new version, described in *Inside Macintosh V*, is called the **preferred** set. You'll use a different library depending on which version of AppleTalk your application uses.

To use...	use this interface file...	and this library...
preferred AppleTalk	AppleTalk.p	nAppleTalk.lib
alternate AppleTalk	AppleTalk.p	ABPackage.lib

Calling Printing Manager routines

There are two ways to use the Print Manager. If you want your project to run under System 4.1 or earlier, use the interface file `Printing.p` and include the library `PrintCalls.lib`. This way, calls to Print Manager routines use glue routines or the printing traps if they exist. If your program runs only on later systems, you should use the interface file `PrintTraps.p` so all calls to the Print Manager routines use the traps directly.

If your program requires...	use this interface file...	and this library...
any System/Finder version	<code>Printing.p</code>	<code>PrintCalls.lib</code>
System Tools 5.0 or later	<code>PrintTraps.p</code>	none

Note: System Tools 5.0 consists of System 4.2 and Finder 6.0. To find out what System version your program is running under, use the Gestalt Manager, described in *Inside Macintosh VI*, Chapter 3, "Compatibility Guidelines," or SysEnviorns, described in *Inside Macintosh V*, Chapter 1, "Compatibility Guidelines."

Calling SANE routines

THINK Pascal lets you choose whether to generate code for the MC68881 or MC68882 floating point unit. If you use the 68881/68882 option, and you use SANE, you need to use a special version of the SANE library.

If you...	use this interface file...	and this library...
don't use the 68881/68882 option	<code>SANE.p</code>	<code>SANELib.lib</code>
use the 68881/68882 option	<code>SANE.p</code>	<code>SANELib881.lib</code>

Calling Time Manager routines

The debugging routines built into THINK Pascal take up so much time that Time Manager tasks may not run often enough when you choose a small interval. In fact, your application may crash when you're running under the environment.

If you're running your application in the THINK Pascal environment, you should schedule tasks to be executed after more than 25 milliseconds. That is, the count parameter to the procedure `PrimeTime` should be greater than 25. When you build your application, you can schedule Time Manager tasks to be executed at any interval.

Note: Twenty-five milliseconds is an approximation. Depending on the kind of Macintosh you're running on, you may be able to use a smaller interval or need to use a larger interval.

To learn more about the Time Manager, see *Inside Macintosh IV*, Chapter 32, "The Time Manager."

Building Projects

12

Introduction

You can write applications, desk accessories, device drivers, and code resources in THINK Pascal. This chapter tells you how to build the different kinds of projects.

What you should know

You should know how to use THINK Pascal. You should know how to create a project and how to add files and libraries to a project. You should know how to run your program in the THINK Pascal environment. If you don't know how to do these things, go back to Chapter 7, "Working with Projects," and to Chapter 8, "Running Programs."

If you want to write Macintosh applications, you should know about the resources that make up an application: menus, window templates, dialog manager, control templates, etc. You should know how to build these resources with a resource editor like ResEdit or with a resource compiler like SAREz. To learn about resources read *Inside Macintosh I*, Chapter 5, "The Resource Manager," and *Inside Macintosh VI*, Chapter 13, "The Resource Manager." Other chapters in *Inside Macintosh* discuss different kinds of resources.

If you want to write desk accessories or device drivers, you should be familiar with the mechanics of DRVr resources. These are a bit more complicated than other resources. See *Inside Macintosh I*, Chapter 14, "The Desk Manager" and *Inside Macintosh II*, Chapter 6, "The Device Manager" to learn about DRVr resources.

If you want to build other kinds of resources (INITs, WDEFs, XCMDs, cdevs, etc.) see "Building Code Resources" later in this chapter.

To learn how to call Macintosh Toolbox routines from THINK Pascal, read Chapter 11, "Using Predefined Routines."

Topics covered in this chapter

- Setting the project type
- Using resource files
- Building applications
- Building desk accessories and device drivers
- Building code resources
- Putting it together

Setting the Project Type

Before you begin working on a project, you set the **project type** to let THINK Pascal know how to build your program into a finished file. To set the project type, choose the **Set Project Type...** command from the **Project** menu. You'll see this dialog box:

The dialog box is titled "Set Project Type". On the left side, there are four icons representing different project types: "Application" (a diamond with a hand), "Desk Accessory" (a calculator), "Driver" (a printer), and "Code Resource" (a computer monitor). The "Application" icon is currently selected. To the right of the icons, there are three sections of information:

- File Information:** Contains "Type:" with a text field containing "APPL", "Creator:" with a text field containing "????", a checked checkbox for "Bundle Bit", and an unchecked checkbox for "Far Code".
- Resource Information:** Contains "Name:" with a text field, "Type:" with a text field, "ID:" with a text field, "Attributes:" with two checkboxes, "Multi-Segment" (unchecked), "Custom Header" (unchecked), and "Segment Type:" with a text field.
- Driver Information:** Contains "Flags:" with a text field, "Delay:" with a text field, and "Mask:" with a text field.

At the bottom right of the dialog are "OK" and "Cancel" buttons.

To set the type of project, click on one of the four icons along the left of the dialog. Depending on the icon you choose, the gray parts of the dialog may become active.

Note: Changing the project type doesn't mean that your program will behave differently. For instance, if you change the project type from Application to Desk Accessory, your application won't turn into a desk accessory automatically. There are different rules for writing the different kinds of projects.

The rest of this chapter goes into detail about the different kinds of projects. For all types of projects, though, you'll need to fill in the file information section. This section lets the Finder know what kind of file you're creating and how to display it on the desktop.

The file type lets the Finder know what kind of file it is. For applications, the type is APPL. Other kinds of files have other types. Text files, for instance, have the type TEXT. The creator field identifies the application that created the file. If the file type is APPL, the creator is the unique four character signature of the application.

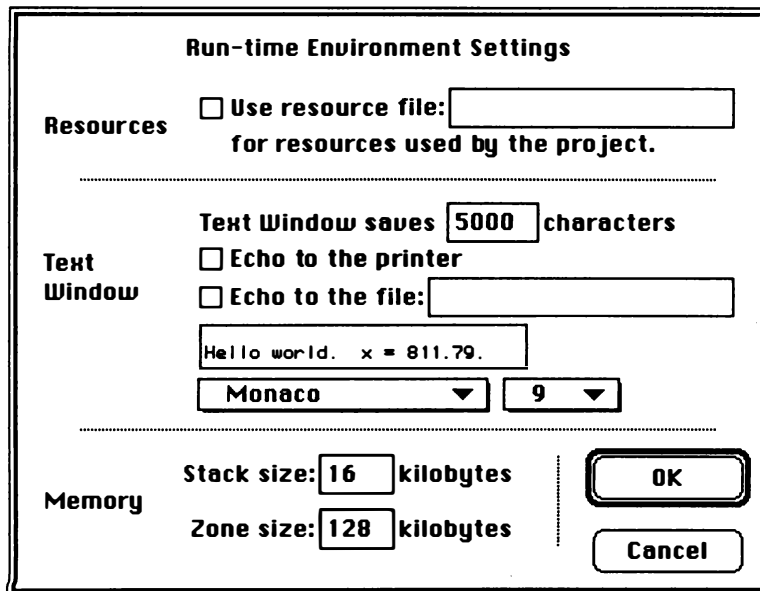
If you check the Bundle bit check box, THINK Pascal will set the bundle bit in the file's Finder information record. The Bundle bit lets the Finder know that your application has a BNDL resource and that it has its own icon.

For more information on how the Finder uses the BNDL resource, see Chapter 7 of the *Resource Utilities Manual*. To learn more about the file type, the file creator, and the bundle bit, see *Inside Macintosh III*, Chapter 1, "The Finder Interface," and *Inside Macintosh VI*, Chapter 9, "The Finder Interface."

Using Resource Files

When you launch an application from the Finder, it looks in the resource fork of the application for its resources. When you run your program in the THINK Pascal environment, it's not quite an application yet, so your program needs to know where to find its resources.

The Resources section of the **Run Options...** command in the **Run** menu lets you specify where your program should expect its resources. When you choose the **Run Options...** command, you'll see this dialog box:



The dialog box is titled "Run-time Environment Settings" and is divided into three sections: Resources, Text Window, and Memory.

- Resources:** Contains a checkbox labeled "Use resource file:" followed by an empty text box. Below this is the text "for resources used by the project."
- Text Window:** Contains a text input field with "5000" and the label "characters". Below this are two checkboxes: "Echo to the printer" and "Echo to the file:" followed by an empty text box. Below these is a text area containing the text "Hello world. x = 811.79." Below the text area are two dropdown menus: the first is labeled "Monaco" and the second is labeled "9".
- Memory:** Contains two text input fields: "Stack size: 16" and "Zone size: 128", both followed by the label "kilobytes".

At the bottom right of the dialog box are two buttons: "OK" and "Cancel".

When you click in the Use resource file: check box, you'll see a standard file dialog. Choose the file that contains the resources for your program. The name of the file you select appears in the box.

Note: THE RESOURCE FILE MUST BE IN THE SAME FOLDER AS THE PROJECT.

When you run your program in the THINK Pascal environment, THINK Pascal opens your resource file so you can access your resources. When you're ready to build your project into a stand-alone application, THINK Pascal copies the resources in your project's resource file into the final application file.

You can use ResEdit or SAREz (in the THINK Pascal 4.0 Utilities folder) to create resource files. To learn more about SAREz, see the Chapter 22, "Using SAREz." To learn more about ResEdit, see *ResEdit 2.1 Reference* (Addison-Wesley) by Apple Computer, Inc.

By convention, files that contain resources to be used as part of a project end in `.rsrc`.

Building Applications

THINK Pascal is set up to build applications by default. When you set the project type, all you need to do is give your application a creator. If the application you're writing is a one-shot — it doesn't need its own icons, and it doesn't create its own files — you can just take the defaults and not bother with the **Set Project Type...** dialog.

Toolbox Initialization

When your program starts up, THINK Pascal inserts calls to the following Toolbox initialization routines for you:

InitGraf	InitDialogs
InitFonts	SetApplLimit
InitWindows	MaxApplZone
InitMenus	MoreMasters (10 times)
TEInit	

If you call these routines while your program is running in the THINK Pascal environment, nothing happens. If you leave them in your final application, your application will crash when it starts.

If you're porting code from other development systems, and that code includes these initialization routines, you can use the `{ $I- }` compiler directive immediately after the **program** statement to disable the automatic initialization. To learn more about this and other compiler directives, see Chapter 15.

Text and Drawing windows

When your program is running in the THINK Pascal environment, you can use the **Text** and **Drawing** commands in the **Windows** menu to display the Text and Drawing windows. If your program uses these windows, you'll need to use the following procedures to work with the Text and Drawing windows in a standalone application:

ShowText	SetTextRect
ShowDrawing	SetDrawingRect
HideAll	

See section 10 of Chapter 17 to learn more about these procedures.

Running the project

When you use one of the execution commands—**Go**, **Step Into**, **Step Over**, **Step Out**, **Go-Go**, or **Step-Step**—in the **Run** menu, THINK Pascal runs your program in an environment that resembles the standard Macintosh software environment.

To learn more about running your project in the THINK Pascal environment, see Chapter 8.

Building applications with large jump tables

If you're building a large application, especially one that uses the THINK Class Library or MacApp, its jump table can grow large quickly. The jump table contains entries for the following:

- Every class
- Every method
- Every routine called by a routine in another segment
- Every routine that you take the address of.

If you need a large jump table, check the "Far Code" option. It lets a jump table grow to 256K, since it uses 32-bit absolute addresses instead of 16-bit relative addresses. Your application will be about 6% larger because of the longer addresses. If the "Far Code" option is off, your jump table can be only 32K.

Far Code lets you make larger applications with more space for your jump table since it handles addresses into the jump table differently. When this option is off, addresses into the jump table are given as offsets from A5. In the MC68000 family, register-relative offsets must be 16-bits long, so the jump table can contain only 32K. When this option is on, addresses to the jump table are absolute addresses and can be 32-bits long. THINK Pascal includes code in your application that adds the value of A5 to these absolute addresses before your code runs.

Note: If you're using Object Pascal and the "Far Code" option is on, the entry «%_MethTables» can be up to 64K., instead of just 32K. However, because of this extended limit, you must put «%_MethTables» in its own segment whenever the "Far Code" option is on. For more information, see "Segmenting a Project" in Chapter 7, "Working with Projects."

If the "Far Code" option is on and your application contains libraries that were compiled with the option off, put those libraries together in their own segment (or segments). THINK Places places the jump table entries for those libraries in first 32K of the jump table. If the libraries are grouped together, THINK Pascal won't waste space in the first 32K by placing in it entries that can be elsewhere. Also, if one of those libraries calls a function in one of your files, put that file in the same segment as those libraries.

Building the application

When you're finished debugging your application, use the **Build Application...** command in the **Project** menu to create your application file. Read the section "Putting it Together" at the end of this chapter to learn how THINK Pascal creates your final application.

Building Desk Accessories and Device Drivers

Desk accessories and device drivers are structurally identical; they're both **drivers**. According to *Inside Macintosh*, drivers don't behave like applications and have a different internal structure. In this section, the word *driver* means either a desk accessory or a device driver.

This section won't teach you how to write a desk accessory or a device driver from scratch. To learn how to write these, read *Inside Macintosh I*, Chapter 14, "The Desk Manager," *Inside Macintosh II*, Chapter 6, "The Device Manager," and *Inside Macintosh V*, Chapter 23, "The Device Manager."

Setting the project type

Set the project type before you start working on a driver. If you set the project type after you've started compiling code, you'll have to recompile your source files.

Choose **Set Project Type...** from the **Project** menu. When the project type dialog appears, click on either the Desk Accessory icon or the Device Driver icon.

The Desk Accessory dialog presets the file type and creator so the resulting file will be a Font/DA Mover file. The ID is set to 12. The Font/DA Mover rennumbers it when you install it in your System. You shouldn't need to change the ID number. All that's left to do is to name the desk accessory and write the code. By convention, desk accessory names begin with a NUL (`chr(0)`). THINK Pascal provides it for you automatically.

Application
 Desk Accessory
 Driver
 Code Resource

File Information
 Type: Creator:
☐ Bundle Bit
☐ Far Code

Resource Information
 Name:
 Type: ID: Attributes:
☐ Multi-Segment ☐ Custom Header
 Segment Type:

Driver Information
 Flags: Delay:
 Mask:

The Device Driver dialog is a little bit different. Device driver names begin with a period. If you don't provide one in the Name field, THINK Pascal automatically provides one for you.

Application

Desk Accessory

Driver

Code Resource

File Information

Type: Creator:

☐ Bundle Bit ☐ Far Code

Resource Information

Name:

Type: ID: Attributes: ☒

☐ Multi-Segment ☐ Custom Header

Segment Type:

Driver Information

Flags: ☒ Delay:

Mask: ☒

OK **Cancel**

Note that the Flags field of the Driver Information section is different for desk accessories and device drivers. Later on you'll see what these two fields mean.

Before you start working on your driver, you need to do one more thing. You need to remove the default `Runtime.lib` library and replace it with the special `DRVRRuntime.lib` library. This library is like `Runtime.lib` except that it uses register A4 to access globals, and it does not contain the Pascal I/O routines. (This means that you can't use `writeln` in a desk accessory, for instance.)

Note: To change the library, hold down the Option key as you double-click on the library name. THINK Pascal displays a standard file dialog that lets you choose the replacement file.

Both device drivers and desk accessories can have more than one segment, just like applications. Just click on the Multi-Segment check box. You can learn more about multi-segment drivers below, but read how a driver works first. For more information on segmentation, see "Segmenting a Project" in Chapter 7.

Note: If your driver uses Object Pascal, you must check the Multi-Segment option even if your driver contains only one segment.

How drivers work

The Device Manager expects drivers to have five entry points and to be written in assembly language. When the Device Manager calls a driver written in THINK Pascal, a short assembly-language

stub translates the Device Manager's request into a call to a Pascal function called `main`. THINK Pascal automatically places the driver glue at the beginning of your driver.

The THINK Pascal driver glue also does two things designed to make writing a driver easier. First, it sets up a data area so that your driver can have its own global variables. Second, it figures out the proper way to return control to the Device Manager automatically. You'll learn more about these services later in this section.

How to write a driver in THINK Pascal

To write a driver in THINK Pascal, you have to follow two rules. The first rule is that a driver must consist only of units. It may not have a file with a main program. The second rule is that a driver must have a function called `main` defined in one of its units, and that function must also appear in its interface section. This is the skeleton for a THINK Pascal driver:

```

unit MyDeskAccessory;
interface

    function main(devCtlEnt: DCtlPtr; paramBlock: ParmBlkPtr;
                  sel: integer): integer;

implementation

    function main;
    begin
        case sel of
            0: { Open      }
            1: { Prime    }
            2: { Control  }
            3: { Status   }
            4: { Close    }
        end
    end;
end.

```

The function `main` takes three arguments that let you know how your driver is being called.

The first argument, `devCtlEnt`, is a pointer to the driver's device control entry. This is the value that is passed in register A1 to the assembly language entry point of the driver.

The second argument, `paramBlock`, is a pointer to an I/O parameter block. This is the value that is passed in register A0 to the assembly language entry point of the driver.

The last argument, `sel`, is a selector that specifies which entry point actually received the call. Use the value of `sel` to dispatch control to the appropriate routine.

Getting the event record pointer from the parameter block

According to *Inside Macintosh I*, Chapter 14, "The Desk Manager," the `csCode` field of the `paramBlock` passed to your driver specifies what kind of action your driver should take. When `paramBlock^.csCode = accEvent`, the `csParam` field of `paramBlock` contains a pointer to an `EventRecord`.

The `csParam` field is defined as an array of integers, so you'd have to cast the first two integers of the array into the pointer to the event record. It's much easier to use the `ioMisc` field, which is defined in another variant of `ParamBlockRec`, because it points to the right place in memory, and it's already a pointer.

```
var
    EventP : ^EventRecord;
...
EventP := Pointer(paramBlock^.ioMisc);
```

Global data in drivers

You can declare global and static variables in drivers. The THINK Pascal driver glue allocates the space for the globals in the heap before it calls `main` to implement the `Open` entry. The glue releases the memory when the driver returns from a `Close` call. (There is a way to keep the global data area allocated after a `Close`; see "Returning from a driver" below.)

Note: All of the globals in your driver are guaranteed to be set to zero on the first `Open` call.

Macintosh applications use register A5 to access their globals. Since drivers co-exist with running applications, they can't use register A5 to access their globals. Instead, drivers use register A4.

The THINK Pascal driver glue stores a handle to the dynamically allocated data area in the `dCtlStorage` field of the driver's device control entry. This handle is dereferenced into address register A4 and locked before each call to `main`. Your `Open` routine must check whether the data area was allocated successfully. If it was not, the `dCtlStorage` field will be 0, and your driver should display some error message (without using any globals!) and close itself. Your code might look like this:

```
if devCtlEnt^.dCtlStorage = nil then
begin
    SysBeep(5);
    exit(main);
end;
```

The data area remains locked between calls to your driver. If you like, you can unlock it yourself before returning. If you unlock the data area, though, make sure that you don't rely on the address of any data item staying the same between calls. Also, make sure that the data area doesn't contain any objects, such as windows, that the Toolbox assumes will not move.

Using driver globals in callback and trap intercept routines

The THINK Pascal driver glue sets up register A4 for you whenever your driver is called from main. If your driver defines callback routines, trap intercept routines, or other functions that might be called when the value of A4 is in doubt, you have to save A4 where your routines can find it.

The special library for drivers, `DRVRRuntime.lib`, contains the procedures `RememberA4`, `SetUpA4` and `RestoreA4` that take care of saving, setting, and restoring A4 for you.

Your main routine must call `RememberA4` before any calls to `SetUpA4`. All calls to `SetUpA4` and `RememberA4` must be from the same segment that contains `main`. This means that the file that contains `main`, `DRVRRuntime.lib`, and any files that contain calls to `SetUpA4` and `RestoreA4` must be in the same segment.

Suppose your driver calls `ModalDialog` with a `filterProc`. Since you're not sure if the value of A4 will be correct when `ModalDialog` calls your `filterProc`, you need to set A4 to the proper value. Your `filterProc` would look like this:

```
function MyFilterProc(dp : DialogPtr; var event: EventRecord;
                    var item : integer) : Boolean;
    var
        result : Boolean;
begin
    SetUpA4;      { main must call RememberA4 first! }
    ...
    MyFilterProc := result;
    RestoreA4;
end;
```

Your main routine would look like this:

```
function main;
begin
    RememberA4;      { Stash A4 where SetUpA4 knows }
                    { where to find it }

    case sel of
        0: { Open      }
        1: { Prime     }
        2: { Control   }
        3: { Status    }
        4: { Close     }
    end
end;
```

Use the same technique for trap intercept routines that need access to a driver's globals. Of course, if your callback or trap intercept routine doesn't use driver globals, you don't need to set up and restore A4.

Using THINK Pascal libraries in drivers

You can use libraries in drivers as long as the libraries don't reference global variables accessed through register A5. The special version of the run-time library for drivers, `DRVRRuntime.lib`, was built so it references its globals from A4.

The QuickDraw globals aren't in `DRVRRuntime.lib`. You can access the real QuickDraw globals in a driver from assembly language by observing that 0 (A5) holds the address of the last of the QuickDraw globals, `thePort`. The remaining QuickDraw globals are at descending addresses from `thePort`; refer to *Inside Macintosh I*, Chapter 6, "QuickDraw" for more information. The value of A5 is stored in the low-memory global `CurrentA5`.

Note: The only reason most people need to use the QuickDraw globals from a driver is to get the bounds of `screenBits`. A common trick to get this information from a driver is to create a new `GrafPort`. The default `portBits` is the same as `screenBits`. Don't forget to get rid of the port once you have what you want.

You can use most of the libraries supplied with THINK Pascal (`PrintCalls`, `nAppleTalk`, `FixMath`, `Graf3D`, etc) in your drivers. You can't use `μRuntime.lib` or `Runtime.lib`.

Using imported libraries in drivers

THINK Pascal honors most initializations from imported THINK C libraries and MPW `.o` files. For example, if a THINK C library contained this initialization:

```
char myString[] = "\psome string";
```

the value of `myString` would be `"\psome string"` when the library is loaded.

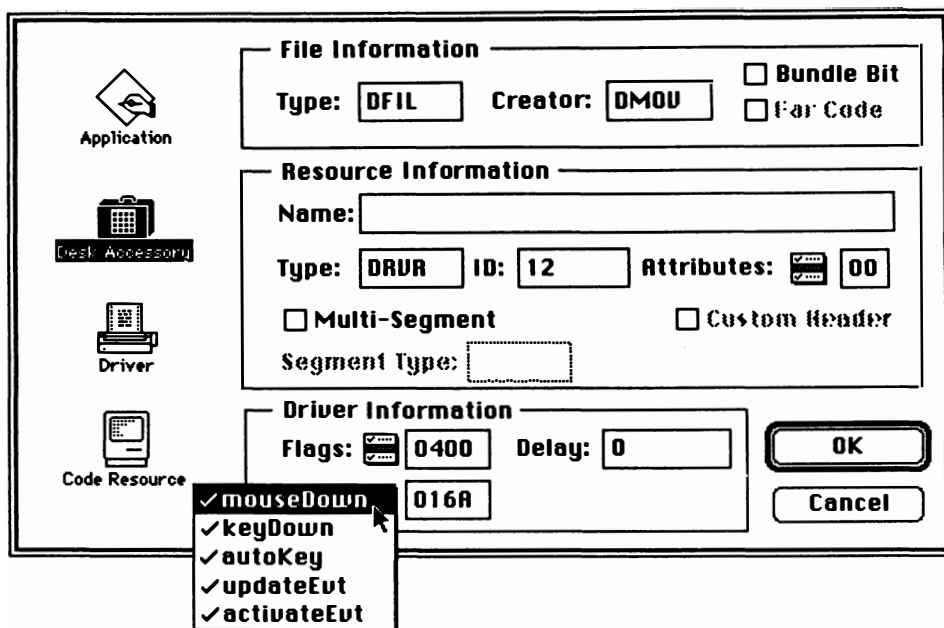
However, in drivers, THINK Pascal does not honor initializations from imported libraries that set a variable to be a pointer to a function or a pointer to another variable (that is, initializations that require runtime relocations). For example, THINK Pascal would not honor this initialization in a THINK C library:

```
static ProcPtr myHook = &myFunction;
```

Setting the fields of a driver's header

A driver begins with a header that contains several flags and other data items (some of which apply only to desk accessories). When the driver is opened, the Device Manager copies these fields to the device control entry before the `Open` entry point is called. After that, the device header is not used. The fields in the driver are used only to initialize the fields in the device control entry.

THINK Pascal lets you set the `drvFlags` and the `drvEMask` fields in the **Set Project Type...** dialog. Use the Flag and Mask pop-up menus to set the flags you want.



These two fields are copied to the `dCtlFlags` and `dCtlEMask` fields of the device control entry.

These are the default settings for desk accessories. The symbols `dReadEnable`, `dWriteEnable`, etc refer to bits of the high byte of the `drvFlags`. `dReadEnable` is bit 0, `dNeedLock` is bit 6. When you set the flag yourself, leave the other bits clear.

Field	Value
<code>dCtlFlags</code>	\$0400
	<code>dReadEnable</code> 0
	<code>dWriteEnable</code> 0
	<code>dCtlEnable</code> 1
	<code>dStatEnable</code> 0
	<code>dNeedGoodbye</code> 0
	<code>dNeedTime</code> 0
	<code>dNeedLock</code> 0
<code>dCtlDelay</code>	N/A because <code>dNeedTime</code> = 0
<code>dCtlEMask</code>	0x016A (<code>mouseDown</code> , <code>keyDown</code> , <code>autoKey</code> , <code>update</code> , <code>activate</code>)
<code>dCtlMenu</code>	0

These are the default settings for device drivers:

Field	Value
dCtlFlags	\$4F00
	dReadEnable 1
	dWriteEnable 1
	dCtlEnable 1
	dStatEnable 1
	dNeedGoodbye 0
	dNeedTime 0
	dNeedLock 1
dCtlDelay	N/A because dNeedTime = 0
dCtlEMask	N/A (desk accessories only)
dCtlMenu	N/A (desk accessories only)

THINK Pascal copies the fields from the driver's header to the device control entry on every Open call. If you want to change the settings of the driver headers on the fly, you'll need to set them on every Open call.

Opening an open driver

The Open entry point of a driver (main's third argument = 0) may be called even if the driver is already open. This happens, for example, when the user selects the name of a desk accessory that's already on the screen. The driver should check to see if it is already open to avoid repeating its initialization sequence.

Since THINK Pascal guarantees that all your driver globals will be set to zero, you can use a global flag to determine whether your driver's been opened before. Your Open routine might look like this:

```

procedure doOpen;

begin
    { AlreadyOpen is a boolean global. }
    if AlreadyOpen then
        exit(doOpen);

    AlreadyOpen := TRUE;
    ...
    { Initialization code }
end;
```

How to return from a driver

If your Open routine was successful, return 0. If the Open routine fails, return a negative result and the driver will not be opened.

Return 0 from Close if it was successful. If your Close routine returns closeErr (-24) the driver won't be closed. If you return a 1, the THINK Pascal driver glue will preserve the dCtlStorage field of the device control entry. This way you can keep your driver globals around until your

driver is reopened. (The driver glue will make it seem as though your `Close` routine returned 0, meaning the `Close` was successful.)

Note: Returning negative values from `Open` and `Close` to prevent opening or closing works only on 128K and later ROMs.

Return 1 from asynchronous calls to `Prime`, `Control`, and `Status` routines if the request could not be completed right away. This result code will be stored in the `ioResult` field of the I/O parameter block, but 0 (no error) will be returned to the Device Manager.

The `jIODone` problem

THINK Pascal always returns from a driver correctly. In other development systems, it's not so easy. Read this section if you want to learn about this problem. Since you don't have to worry about it, you might want to skip this section.

One of the trickiest aspects of returning from a driver is deciding whether to return directly to the Device Manager (via an RTS instruction) or whether to jump to `jIODone`. This is a complex issue, and many existing desk accessories do it wrong (though, fortuitously, they manage to work anyway).

Associated with each driver is an I/O queue, which is a list of I/O parameter blocks waiting for service from the driver. Calls made to a driver fall into one of two categories: *queued*, meaning that the I/O parameter block passed as an argument to the call is in the driver's queue; and *immediate*, meaning that it is not. In the immediate case, the queue may even be (and in fact usually is) empty.

All `Open` and `Close` calls are immediate. All `Control` calls made to desk accessories are immediate, except for the "goodbye kiss" (`csCode=-1`) issued to desk accessories that have requested to be notified when the current application exits out from under them. Other calls may be queued or immediate.

The rules for returning from a driver are: The driver should return directly to the Device Manager from all immediate calls. It should also return directly to the Device Manager from queued calls requesting asynchronous I/O that could not be completed right away. Finally, it should jump to `jIODone` from queued calls if the driver completed the request (or if there was an error).

It is incorrect to violate these rules; in particular, it is incorrect to jump to `jIODone` to return from an immediate call. `jIODone` will attempt to examine the driver's I/O queue, and since the queue is usually empty it will end up examining low-memory locations beginning at \$0000. Apparently, these locations somehow look enough like an I/O parameter block to satisfy the Device Manager, but this is clearly an unsafe situation.

Just to make things difficult, when returning from `Prime`, `Control`, and `Status` calls, it is `jIODone` that unlocks the driver's code and its device control entry so they won't form islands in the heap between calls to the driver (unless, of course, the driver has requested that they remain locked). So the author of a desk accessory, for instance, has to make a difficult decision — to return directly to the Device Manager, leaving the driver's code and its device control entry locked and

potentially interfering with the host application; or to violate the rules and jump to `jIODone`. Most desk accessories seem to take the latter route.

THINK Pascal avoids this dilemma. When a driver written in THINK Pascal returns from `main`, the decision whether to call `jIODone` is made automatically (and correctly). For `Prime`, `Control`, and `Status` calls, if the decision is made to return directly to the Device Manager, and the driver has not requested that its code and device control entry remain locked, they are unlocked.

Multi-Segment drivers

If the Multi-Segment option is on, drivers can contain multiple segments. The `Type` field in the **Set Project Type...** dialog lets you specify the resource type of the owning resource. THINK Pascal gives the owned resources the type `DCOD`.

The `ID` field in the **Set Project Type...** lets you choose the resource ID for the owning resource. THINK Pascal ensures that the resource ID for the owning resource is between 0 to 63. If you enter a number outside that range, you'll see an error message. THINK Pascal numbers the owned resources for you.

The `Attributes` field in the **Set Project Type...** dialog lets you set up the resource attributes of the owning resource. To set the resource attributes for an owned resource, double click on the segment's summary line in the project window's segment view, and use the `Attributes` field in the dialog that appears. For more information, see "Naming segments" in Chapter 7, "Working with Projects."

As with applications, segments are loaded automatically as they are called. In addition, all loaded segments are unloaded automatically upon return to the Device Manager after each call, *unless* the `dNeedLock` bit is set in the driver's device control entry.

To make sure that there is enough memory for a multi-segment driver, you may want to "preflight" it. "Preflighting" means that you load all the segments to see if there's enough memory for them. If there's enough memory, you can continue executing. Otherwise, you exit gracefully.

To preflight your driver, you must first name your segments. To learn how, see "Segmenting a Project" in Chapter 7. Then, use the `GetNamedResource` function to load in each segment like this:

```
var
  h: Handle;
begin
  h := GetNamedResource ('DCOD', 'segment-name');
  if h = nil then
    { exit gracefully }
  ...
end;
```

Be sure to check the return value of `GetNamedResource`. If it's `nil`, you don't have enough memory for your driver, and you should exit gracefully.

Note: Don't use LoadSeg. It loads CODE resources. Driver segments are DCOD resources.

You can unload driver segments manually with this function:

procedure UnloadA4Seg(theProcPtr: ProcPtr);

This function works just like UnloadSeg does in applications. To unload one segment, call UnloadA4Seg with the address of a routine in that segment; for example, UnloadA4Seg(@foo). To unload all segments at once, call UnloadA4Seg with **nil**; that is, UnloadA4Seg(**nil**).

Note: Do not use UnloadSeg instead of UnloadA4Seg by mistake!

Read "Arranging Files in the Project" in Chapter 7 to learn how to break up your project into segments.

Note: If your driver uses Object Pascal, you must check the Multi-Segment option even if your driver contains only one segment.

Running desk accessories

You can't run a desk accessory with one of the execution commands in the Run menu because desk accessories aren't real applications. (They don't even have a main program!) To run a desk accessory, use the special DA Shell program that's included in your THINK Pascal package.

The DA Shell is a small program that simulates an environment for your desk accessory. It takes advantage of the fact that a desk accessory is a unit and that it's called through the function main. The DA Shell creates a fake device control entry and a fake I/O parameter block for your main function.

To use the DA Shell:

- Make a backup copy of the DA Shell file
- Create a new project with the default libraries
- Add your desk accessory units to the project
- Add the DA Shell file to the project
- In the DA Shell file, put your desk accessory's unit name in the **uses** clause

To run your desk accessory, choose one of the execution commands — **Go**, **Step Into**, **Step Over**, **Step Out**, **Go-Go**, and **Step-Step**— from the **Run** menu, then choose **Sample DA** from the **Apple** menu. When you use the DA Shell, you can use any of THINK Pascal's debugging tools to debug your desk accessory.

Desk accessories that work in the DA Shell should work properly as real desk accessories. But since the DA Shell is a simulated environment, you should be aware of several things:

- When your desk accessory is running in the DA Shell, events in your windows look like inContent. When you run your desk accessory as a real desk accessory, these events will be

inSysWindow. At least while you're debugging, your desk accessory should treat inContent hits as inSysWindow hits.

- If your desk accessory uses owned resources, calculate the unit number as $\text{abs}(\text{dCtlRefNum}) - 1$. In a real desk accessory dCtlRefNum is guaranteed to be negative, so the unit name is calculated as $-\text{dCtlRefNum} - 1$, but in the DA Shell it's $\text{dCtlRefNum} - 1$.
- In the DA Shell, your desk accessory will get a goodbye kiss synchronously (immediately). As a real desk accessory, it gets it asynchronously.

If you're writing a multi-segment desk accessory, and you want to use the DA Shell, you may need to define dummy versions of the A4 routines like this:

```
unit DummyA4World;
interface
    procedure SetUpA4;
    procedure RestoreA4;
    UnloadA4Seg(pPtr: ProcPtr);

implementation
    procedure SetUpA4;
    begin
    end;

    procedure RestoreA4;
    begin
    end;

    procedure UnloadA4Seg(pPtr: ProcPtr);
    begin
        UnloadSeg(pPtr)
    end;
end.
```

Just add this unit to your project after your libraries. You don't need to put the unit's name in a uses clause since the compiler will treat the procedure names as externals and resolve them at link time.

Building Code Resources

You can use THINK Pascal to write pure code resources. Code resources don't have the complex structure of drivers. They simply contain code to be called at the entry point, the function or procedure main.

You might want to write code resources for several reasons. You might want to write a window definition function (WDEF) that you can use in several other programs, or you might want to write an INIT to run at startup. You may define your own code resource types to make a function

you've written in THINK Pascal available to a program written in another language. The "client" program simply loads the resource and calls it at its beginning.

This section tells you how to build code resources in THINK Pascal. The specific formats and calling sequences for code resources are given in the various volumes and chapters of *Inside Macintosh*. This list will help you get started.

To learn how to build a...

ADBS resource
CDEF resource
cdev resource
FKEY resource
INIT resource

LDEF resource
MBDF resource
MDEF resource

WDEF resource
XCMD resource
XFCN resource

Read *Inside Macintosh*...

Volume V, Chapter 20, "The Apple Desktop Bus"
Volume I, Chapter 10, "The Control Manager"
Volume V, Chapter 18, "The Control Panel"
Macintosh Technical Note #3 (also see below)
Volume IV, Chapter 29, "The System Resource File"
Volume V, Chapter 19, "The Start Manager"
Volume IV, Chapter 30, "The List Manager Package"
Volume V, Chapter 13, "The Menu Manager"
Volume I, Chapter 11, "The Menu Manager"
Volume V, Chapter 13, "The Menu Manager"
Volume I, Chapter 9, "The Window Manager"
HyperCard Script Language Guide, Appendix A
HyperCard Script Language Guide, Appendix A

Setting the project type

Set the project type before you start working on a code resource. If you set the project type after you've started compiling code, you'll have to recompile your source files and reload your libraries. Choose **Set Project Type...** from the **Project** menu. When the dialog appears, click on the Code Resource icon.

Application

Desk Accessory

Driver

Code Resource

File Information

Type: Creator:

☐ Bundle Bit
☐ Far Code

Resource Information

Name:

Type: ID: Attributes: ☐ ☐ 00

☒ Multi-Segment ☐ Custom Header

Segment Type:

Driver Information

Flags: ☐ ☐ Delay:

Mask: ☐

OK
Cancel

Fill in the Type and ID of the code resource you're building. If you like, you can give your code resource a name.

Use the Attributes pop-up menu to set the resource attributes for your code resource. To learn about resource attributes, see *Inside Macintosh I*, Chapter 5, "The Resource Manager."

If the Multi-Segment option is checked, code resources can contain more than one segment and global data. The Segment Type field lets you specify the resource type of your code resource's owned segments. For more information, see "Multi-segment code resources" later in this section.

Note: If your code resource uses Object Pascal or global data, you must check the Multi-Segment option even if your driver contains only one segment.

If you check the Custom Header check box, THINK Pascal doesn't use the standard code resource header to start the resource. See "Code resource headers" later in this section.

Before you start working on your code resource, you need to replace the default `Runtime.lib` library with the `RSRCRuntime.lib` library. This library is like `Runtime.lib` except it doesn't contain the Pascal I/O routines. (This means that you can't use `writeln` in a code resource, for instance.) `DRVRRuntime.lib` and `RSRCRuntime.lib` differ slightly in how they handle the A4 world.

How to write a code resource in THINK Pascal

To write a code resource in THINK Pascal you must follow some rules:

- It must consist only of units.
- It must not have a main program.
- It must have a unit that contains a function or a procedure called `main`.
- The `main` routine must appear in the interface section of its unit.

The way you write your `main` routine depends on the kind of resource you're writing. An `FKEY`, for example, is called by the Event Manager. It doesn't have any arguments. You would define `main` like this:

```
unit MyFKEY;
interface

    procedure main;

implementation

    procedure main;
    begin
    end;
end.
```

A WDEF resource is a custom window definition. The Window Manager calls `main` with several arguments and expects the window definition function to return a `longint`. This is how you would write `main` for a WDEF:

```
function main (varCode: integer; theWindow: WindowPtr;
               message: integer; param: longint): longint;
begin
    ...
end;
```

Global data in code resources

Code resources can have global data only if the Multi-Segment option in the **Set Project Type...** dialog is on. Unlike drivers and applications, you can't use global data if the Multi-Segment option is off.

Note: Even if the Multi-Segment option is on, you can create a code resource with only one segment. It's a multi-segment code resource with only one segment.

For more information on writing multi-segment code resources, see "Multi-segment code resources" later in this section.

Using THINK Pascal libraries in code resources

You can use libraries in code resources as long as the libraries don't reference global variables accessed through register A5. The library `RSRCRuntime.lib` references its globals from A4

The QuickDraw globals aren't in `RSRCRuntime.lib`. You can access the real QuickDraw globals in a driver from assembly language by observing that 0 (A5) holds the address of the last of the QuickDraw globals, `thePort`. The remaining QuickDraw globals are at descending addresses from `thePort`; refer to *Inside Macintosh I*, Chapter 6, "QuickDraw" for more information. The value of A5 is stored in the low-memory global `CurrentA5`.

Note: The only reason most people need to use the QuickDraw globals from a driver is to get the bounds of `screenBits`. A common trick to get this information from a code resource is to create a new `GrafPort`. The default `portBits` is the same as `screenBits`. Don't forget to get rid of the port once you have what you want.

You can use most of the libraries supplied with THINK Pascal (`PrintCalls`, `nAppleTalk`, `FixMath`, `Graf3D`, etc) in your drivers. You can't use `μRuntime.lib` or `Runtime.lib`. Also, you *cannot* use `DRVRRuntime.lib` in a multi-segment code resource.

Using Imported Libraries In code resources

THINK Pascal honors most initializations from imported THINK C libraries and MPW .o files. For example, if a THINK C library contained this initialization:

```
char myString[] = "\psome string";
```

the value of myString would be "\psome string" when the library is loaded.

However, in drivers, THINK Pascal does not honor initializations from imported libraries that set a variable to be a pointer to a function or a pointer to another variable (that is, initializations that require runtime relocations). For example, THINK Pascal would not honor this initialization in a THINK C library:

```
static ProcPtr myHook = &myFunction;
```

Locking code resources

The Macintosh Toolbox takes care of locking and unlocking the standard code resources like WDEFs. When you write your own code resources, you can either let the caller take responsibility for locking and unlocking them, or you can have the code resource do it itself.

When main is entered, the low memory global ToolScratch contains a pointer to your code resource. If you need to lock it, you would write main like this:

```
procedure main;

  var
    pp: ^Ptr;
    h: Handle;

begin
  pp := Ptr($09CE); { ToolScratch }
  h := RecoverHandle(pp^);
  HLock(h);
  ...
  HUnlock(h);
end;
```

Note: When the Toolbox uses an MDEF, it expects to find a certain value in ToolScratch. To write an MDEF, you'll have to use a Custom Header, described below.

If your code resource can be called reentrantly, don't unlock it unconditionally when it returns. Instead, restore it to the same state of locked-ness it had on entry.

Code resource headers

When THINK Pascal creates a code resource it places this standard header at the beginning:

Offset	Contents
0	BRA.S .+\$10 (branch to header code)
2	\$0000 (unused)
4	'TYPE' (resource type)
8	\$000A (resource ID)
10 (\$A)	\$0000 (unused)
12 (\$C)	\$0000 (unused)
14 (\$E)	\$0000 (unused)

The standard header code puts the address of your code resource in `ToolScratch` and then branches to your main routine. You can do anything you like with the unused words.

If you check the Custom Header option in the **Set Project Type...** dialog, THINK Pascal does not generate this standard resource header. Instead, the file that contains `main` is guaranteed to be the first file in the code resource.

You can use this feature for writing resources that require special headers like PDEFs. The file that contains `main` should be written in assembly language. See Chapter 13 to learn how to write routines in assembly language for use in THINK Pascal.

Note: Multi-segment code resources must use THINK Pascal's default header. If the Multi-Segment option is checked, the Custom Header option is dimmed.

Multi-segment code resources

If the Multi-Segment option is checked in the **Set Project Type...** dialog, code resources can global data and up to 31 segments. Read the section "Segmentation" earlier in this chapter to learn how to break up a project into different segments.

The Type field in the **Set Project Type...** dialog lets you specify the resource type of the owning resource. To specify the resource type of the owned resources, use the Segment Type field. The Segment Type default is CCOD.

The ID field in the **Set Project Type...** lets you choose the resource ID for the owning resource. THINK Pascal ensures that the resource ID for the owning resource is between 0 to 63. If you enter a number outside that range, you'll see an error message. THINK Pascal numbers the owned resources for you.

The Attributes field in the **Set Project Type...** dialog lets you set up the resource attributes of the owning resource. To set the resource attributes for an owned resource, double click on the segment's summary line in the project window's segment view, and use the Attributes field in the dialog that appears. For more information, see "Naming segments" in Chapter 7, "Working with Projects."

The Custom Header option in the **Set Project Type...** is dimmed. Multi-segment code resources must use THINK Pascal's default header.

Before you start working on your code resource, you need to replace the default `Runtime.lib` library with the `RSRCRuntime.lib` library. This library is like `Runtime.lib` except that it uses register A4 to access globals and doesn't contain the Pascal I/O routines. (This means that you can't use `writeln` in a code resource, for instance.)

Note: To change the library, hold down the Option key as you double-click on the library name. THINK Pascal displays a standard file dialog that lets you choose the replacement file.

In multi-segment code resources, variables and jump table entries are addressed as offsets from A4. Unlike drivers, however, A4 isn't set up automatically for you when your main routine is called. You have to do this yourself. Immediately after you enter `main`, you must call `RememberA4` and `SetUpA4`. `RememberA4` saves the value of A4 where the `SetUpA4` can find it. You must call `RestoreA4` before you return from `main`. This is what the main routine for your code resource should look like:

```
procedure main;
begin
    RememberA4;
    SetUpA4;
    ...
    RestoreA4;
end;
```

All calls to `SetUpA4` and `RememberA4` must be from the same segment that contains `main`. This means that the file that contains `main`, `RSRCRuntime.lib`, and any files that contain calls to `SetUpA4` and `RestoreA4` must be in the same segment.

As with the other project types, segments are loaded automatically as they are called. However, unlike the other project types, you must unload the segments yourself when the code resource exits. You can unload individual segments with this function:

```
procedure UnloadA4Seg(routine: ProcPtr);
```

This function works just like `UnloadSeg` does in applications. To unload one segment, call `UnloadA4Seg` with the address of a routine in that segment; for example, `UnloadA4Seg(@foo)`. To unload all segments at once, call `UnloadA4Seg` with `nil`; that is, `UnloadA4Seg(nil)`. Before you call `RestoreA4`, it's a good idea to call `UnloadA4Seg(nil)` to make sure that all your code resource's segments are unloaded.

Note: Do not use `UnloadSeg` instead of `UnloadA4Seg` by mistake!

In code resources, the segment that contains the main routine is special. The jump table and global data for all the segments are appended to the segment that contains `main`, so the resource's global data, the jump table, and the main segment's code must be less than 32K.

Note: If your code resource uses Object Pascal or global data, you must check the Multi-Segment option even if your driver contains only one segment.

Using code resource globals in callback and trap intercept routines

If your multi-segment code resource uses callback, trap intercept routines, or other functions that might be called when the value of A4 is in doubt, you have to save A4 where your routines can find it. The special library for multi-segment code resources, `RSRCRuntime.lib`, contains the procedures `SetUpA4` and `RestoreA4` that take care of setting and restoring A4 for you.

All calls to `SetUpA4` must be from the same segment that contains `main`. This means that the file that contains `main`, `RSRCRuntime.lib`, and any files that contain calls to `SetUpA4` and `RestoreA4` must be in the same segment.

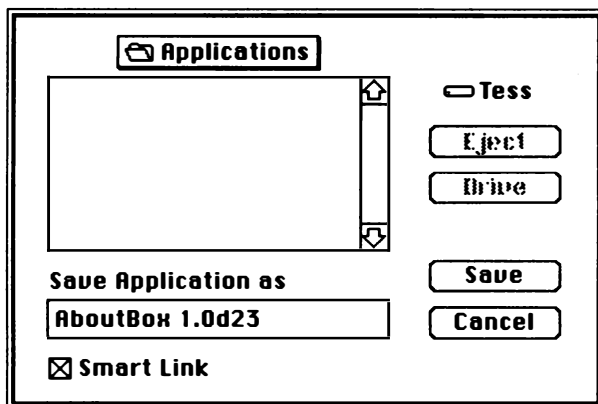
Suppose your code resource calls `ModalDialog` with a `filterProc`. Since you're not sure if the value of A4 will be correct when `ModalDialog` calls your `filterProc`, you need to set A4 to the proper value. Your `filterProc` would look like this:

```
function MyFilterProc(dp : DialogPtr; var event: EventRecord;
                    var item : integer) : Boolean;
    var
        result : Boolean;
begin
    SetUpA4;
    ...
    MyFilterProc := result;
    RestoreA4;
end;
```

Use the same technique for trap intercept routines. Of course, if your callback or trap intercept routine doesn't use code resource globals, you don't need to set up and restore A4.

Putting It Together

When you're finished developing your application, desk accessory, device driver, or code resource, choose one of the **Build...** commands in the **Project** menu. The actual name of the **Build...** command could be **Build Application...**, **Build Desk Accessory...**, **Build Driver...**, or **Build Code Resource...**, depending on the project type. When you choose one of the commands, you'll see a dialog box like this:



When THINK Pascal puts your final file together, it compiles all the files that need to be recompiled. If you're building an application, and you have the Debug option on, THINK Pascal recompiles all the files with the Debug option turned off.

Note: THINK Pascal does not change any of the other compiler options during a build. You will probably want to disable Range and Overflow checking yourself.

To make your final file as small as possible, THINK Pascal uses a technique called **smart linking**. THINK Pascal examines all the routines that your project uses. If there are functions, procedures, or methods that your program doesn't reference, it doesn't incorporate their code in the final file.

Smart linking yields a smaller final application, but it takes longer to produce it. If you're doing a lot of builds you might want to turn smart linking off. Just click on the Smart Link check box when you build your final file.

After THINK Pascal links your program, it produces the application, desk accessory, device driver, or code resource file. At the end, it copies the resources from the resource file you specified in the **Run Options...** dialog into the final file.

Assembly Language

13

Introduction

Most of the Macintosh Toolbox routines expect to be called from Pascal. The data types used by the routines are usually described as Pascal records, pointers, and arrays. That's why working in Pascal on the Macintosh is a natural fit.

Of course, deep inside the Macintosh works in MC68000 machine language. For some specialized tasks, assembly language is the only way to get the job done.

This chapter shows you how to use assembly language routines in your THINK Pascal programs. It begins with an overview of the Macintosh software environment and a description of how Pascal data structures exist within that environment. Next is a description of Pascal calling conventions so you can be sure that your assembly language routines are well-behaved. Finally, this chapter shows you how to use assembly language routines written in other development environments with THINK Pascal.

Even if you don't plan on writing or using assembly language routines in your programs, this chapter will give you enough background to help you use the built in LightsBug debugger as well as low level debuggers like TMON or Macsbug.

What you should know

This chapter assumes you know about MC68000 assembly language. But you don't have to be a proficient assembly language programmer to understand this chapter. And although this chapter gives you an overview of the Macintosh software architecture, it doesn't go into too much detail. To learn more about how the Macintosh manages memory and how it loads your program, read *Inside Macintosh II*, Chapter 1, "The Memory Manager," *Inside Macintosh II*, Chapter 2, "The Segment Loader," and *Inside Macintosh VI*, Chapter 28, "Memory Management."

If you plan to write assembly language routines to use in THINK Pascal, you can use THINK C or Apple's Macintosh Programmer's Workshop.

Topics covered in this chapter

- The runtime environment
- Pascal data types
- Pascal calling conventions
- Using assembly language

The Runtime Environment

While your program is running, all memory is divided into three parts: the stack, the heap, and the A5 world.

The Stack

The **stack** is an area of memory that is dynamically allocated and deallocated in a strict last-in-first-out fashion, like a stack of trays in a cafeteria. The MC68000's A7 register is reserved as a stack pointer. It always contains the address of the top of the stack.

Note: The stack in the MC68000 actually grows downward, towards lower memory addresses, so the top of the stack is actually the byte in the stack with the lowest address.

The stack contains information about the activation and deactivation of procedure and functions. Each time a routine is called, a **stack frame** is allocated. The stack frame contains all of the routine's parameters, local variables and temporaries, and the return address. When the routine exits, the stack frame is released and the context of the calling routine is restored. The register A6 is the frame pointer of the currently active procedure or function.

The Heap

The **heap** (also called a **zone** in Macintosh terminology) is the area of memory that the Macintosh uses to allocate dynamic data structures like windows, menus, resources, the code for the program itself, and other heap zones.

The Macintosh Memory Manager takes care of all the housekeeping chores. It knows where a new block can be allocated, what to do with deallocated blocks, and how to compact the heap when it needs space.

The memory within a heap is divided into three kinds of blocks:

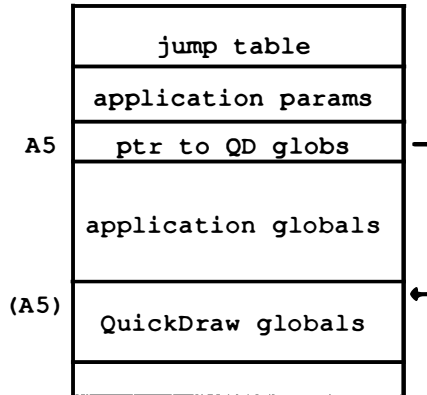
- **Free Blocks.** These blocks represent unused heap memory that may be allocated to satisfy a memory request.
- **Non-relocatable Blocks.** These are allocated blocks of memory that reside at a fixed location. They are referenced by a pointer to the block.
- **Relocatable Blocks.** These are blocks that the Memory Manager can move around to make more room in the heap for larger blocks. Because a relocatable block can move, your application can't keep a pointer to it. Instead, the Memory Manager maintains and updates a master pointer that points to the block of memory. The Memory Manager gives you a **handle**, a pointer to the master pointer, so you can access the block. To access a relocatable block, you dereference the handle twice.

The Memory Manager is one of the fundamental parts of the Macintosh operating system. There is a great deal of myth and lore concerning its effective use. For more information, see *Inside*

Macintosh II, Chapter 1, "The Memory Manager" and Scott Knaster's *How to Write Macintosh Software*.

The A5 World

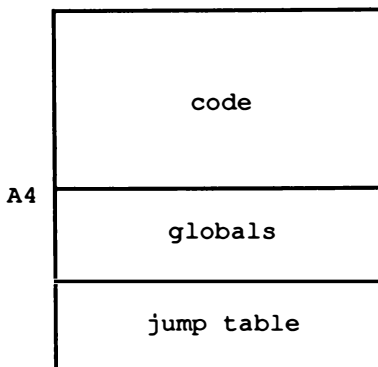
The A5 world is a colloquial term for the area of memory that is referenced through register A5. The A5 world looks like this:



- The 32 bytes from the memory location specified in register A5 contain the **application parameters**. This is information used and maintained by the system. The only part of this section you might use is 0(A5) which contains a pointer to the beginning of the QuickDraw globals.
- Immediately following the application parameters is the **jump table**. The Segment Loader uses the jump table to load and unload segments of code. Every procedure and function that is referenced across segments has an entry in this table. For more information about the Segment Loader see *Inside Macintosh II*, Chapter 2, "The Segment Loader." To learn how to break up your THINK Pascal program into segments, see Chapter 7.
- The area of memory below A5 is reserved for the **application globals**. All of the globals that your application defines are stored here, as are the QuickDraw globals.

The A4 World

If you build a multi-segment desk accessory, device driver, or code resource, you use the A4 world. The A4 world is a colloquial term for the area of memory that is referenced through register A4. The A4 world looks like this:



- A4 points to the beginning of your **globals**. All of the globals that your program defines are stored here.
- Immediately following your globals is the **jump table**. The Segment Loader uses the jump table to load and unload segments of code. Every procedure and function that is referenced across segments has an entry in this table. For more information about the Segment Loader see *Inside Macintosh II*, Chapter 2, "The Segment Loader." To learn how to break up your THINK Pascal program into segments, see Chapter 7.

Pascal Data Types

When you're writing an assembly language routine, you need to know how THINK Pascal stores the various Pascal data types in memory.

Integer Types

Integers of type `integer` are represented as a 16-bit two's complement number with a range of -32,768 to 32,767.

Integers of type `longint` are represented as a 32-bit two's complement number with a range of -2,147,483,648 to 2,147,483,647.

Subranges of integer in the range -128 to 127 are represented in 8 bits. Subranges with bounds outside this range occupy 16 bits.

If an integer subrange is a component of a packed array and it is in the range 0 to 255, it is represented as an unsigned byte (8 bits). If an integer subrange is a component of a packed record and it is in the range 0 . . 65535, it is represented with the least number of bits possible. (For example, the subrange 0 . . 15 would be represented in 4 bits.)

Chars

The type `char` is represented as a 16-bit value with the extended ASCII code of the character in the low-order byte, and 0 in the high-order byte.

If a `char` is a component of a packed structured-type, it is represented as an unsigned byte (8 bits). If a character subrange is a component of a packed record, it is represented with the least number of bits possible. (For example, the subrange 'a'.. 'z' would be represented in 7 bits.) Other packed structured-types represent character subranges with 8 bits.

Booleans

The type `boolean` is represented as a byte. It may assume only the values 0 (false) or 1 (true).

If a `boolean` is a component of a packed structured-type, it is represented as a bit (1 bit).

Enumerated Types

Enumerated types are represented as unsigned bytes. They may assume ordinal values in the range 0 to 255 depending upon the number of enumerated constants in the type.

If an enumerated type is a component of a packed record, it is represented with the least number of bits possible. (For example, the enumerated type (mon, tue, wed, thu, fri, sat, sun) would be represented with 3 bits.)

Real Types

The real-types `real`, `double`, and `extended` are represented as IEEE-format floating-point numbers of 32, 64 and 80 bits respectively. The real type `COMPUTATIONAL` is represented as a 64-bit two's complement integer.

Note: If you compile your program with the 68881/68882 option, extended values are 96 bits long. To learn more about the 68881/68882 option, see Chapter 15.

The floating-point formats are described in complete detail in *Apple Numerics Manual, Second Edition* (Addison-Wesley)

Pointers

Pointer-types are represented as 32-bit address values. Only the low order 24 bits contain the address. The Macintosh Memory Manager uses the high-order 8-bits, and they are not guaranteed to be 0. Use the Toolbox routine `StripAddress` to turn a Memory Manager address into a canonical address before doing any pointer arithmetic.

Strings

A **string** of size `n` has a 1 byte length field followed by `n` bytes containing the character components of the string (each occupying a single byte as if packed). An unused byte is added to the end if needed to ensure that the total number of bytes is even.

Arrays

An array with index $[L..H]$ is represented as if $H-L+1$ variables of the component type were laid end to end. If the size of the component type is not 1, it is first rounded to an even number of bytes so that each element of the array is on an even byte boundary. An unused byte is appended to the array if necessary to ensure that it occupies an even number of bytes.

A multidimensional array of indices $[L1..H1, L2..H2, \dots, Ln..Hn]$ is represented as if it were declared as an array of index $[L1..H1]$ with a component array of index $[L2..H2]$ with a component array of index $[L3..H3]$, etc.

A packed array is identical to the corresponding array type unless the component type is packable (e.g., CHAR or integer subrange in the range $0..255$) in which case the components are allocated in their packed format.

Records

A record with fields $f1: T1, f2: T2, \dots, fn: Tn$ is represented as if each field were a single variable and all fields were laid end to end. If a field's size is not 1, the field is first aligned to an even boundary. An unused byte is appended to the rest of the record if necessary to ensure that it occupies an even number of bytes.

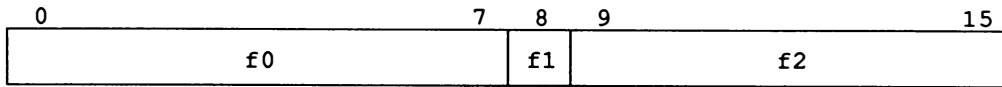
A packed record is identical to the corresponding record type unless the types of one or more fields are packable (e.g., CHAR or integer subrange in the range $0..255$) in which case the fields are allocated in their packed format. These are the types THINK Pascal packs:

Type	Example	Range	# Bits
Any boolean type	Boolean	0..1	1
Any character type or subrange	char 'a'..'z'	0..255 0..122	8 7
Any enumerated type	(mon, tue, wed, thu, fri, sat, sun)	0..6	3
Any integer subrange in the range 0..65535	0..15 0..65535	0..15 0..65535	4 16

To make a packed record, THINK Pascal packs the fields into the byte starting from the most significant bit to the least significant bit. It packs the fields as tightly as the word-alignment rules let it. For example, this record:

```
packed record
  f0: Char;      { 8 bits }
  f1: Boolean;   { 1 bit  }
  f2: 0..127;    { 7 bits }
end;
```

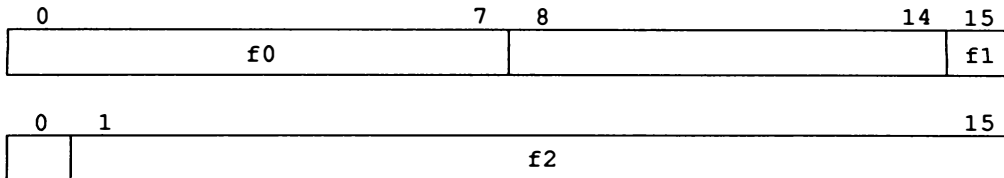
is packed neatly into two bytes:



But this record:

```
packed record
  f0: Char;      { 8 bits }
  f1: Boolean;   { 1 bit  }
  f2: 0..32767; { 15 bits }
end;
```

must be packed into four bytes (empty fields are unused):



To avoid straddling an odd byte boundary, THINK Pascal must place the 15-bit field f2 in a different byte from f1. Note that if the fields of a packed record do not completely fill a word, THINK Pascal aligns the fields to the least significant bit-position where possible to access the field more efficiently.

Sets

Sets are represented as bit arrays where each bit indicates whether the corresponding element is in the set or not. Sets occupy an even number of words, and are always allocated as if the set origin were 0: a set of 0..255 occupies the same number of bytes as a set of 100..255.

Note: A set with elements in the ordinal range 0..7 is stored as a byte. However, when passed as an actual value parameter, the set is first extended to a word, so the value ends up in the low-order byte rather than the high-order byte.

Files

A file is represented as a record of 58 bytes of status information followed by a component buffer whose size is equal to the size of the component type of the file.

Files of type text are represented identically to **packed file of char**. However, the predefined routines `eoln`, `writeln`, and `readln` can only be used with files of type text. See Section 9 of Chapter 17.

Pascal Calling Conventions

Most of the Macintosh Toolbox routines expect to be called as if from Pascal. When you write your assembly language routines, be sure that they follow the Pascal calling conventions.

Pascal calling sequence

If the routine is a function, it is the caller's responsibility to reserve space on the stack for the return value. The caller then pushes the arguments in left-to-right order and calls the function. Upon return, the result (if any) may be found on the stack. The caller's code looks something like this:

```

SUBQ      #n, SP      ; reserve space for result
MOVE      arg1, -(SP)  ; first argument
...
MOVE      argn, -(SP)  ; last argument
JSR       routine-addr
MOVE      (SP)+, result ; result
    
```

If the called routine is a stack-based Toolbox trap, THINK Pascal generates the appropriate trap word instead of the JSR instruction. For all other routines, the compiler generates a JSR instruction.

Note: For register-based traps, for routines dispatched through a single trap, or for routines marked [Not in ROM] the JSR goes to glue code in `Interface.lib`.

If the "Far Code" option is on, there might be a NOP (no-op) instruction after the JSR instruction. The instruction is needed because the linker replaced a 32-bit address with a 16-bit address. When the Far Code option is on, the code generator uses a 32-bit absolute address to a jump table entry for *routine-addr*. If the called routine is in the same segment as the caller, the linker replaces the 32-bit *routine-addr* with a 16-bit relative address off A5 and a 16-bit NOP instruction. This replacement makes it more likely that the linker can delete the jump table entry for that routine. For more information on the Far Code option, see "Building applications with large jump tables" in Chapter 12.

If the called routine is in a nested scope, the caller also provides the static link, the frame pointer of the most recent activation of the nesting procedure or function. It's very unlikely that any assembly language routines you write will need a static link.

Pascal routine entry

Just after the call to the Pascal routine, the return address is on the top of the stack. Most of the time, the routine will create a stack frame with the LINK instruction, and it will save any non-scratch registers.

As the routine begins, the stack looks like this:

	return value
	first argument
	...
	last argument
	static link (opt)
	return address
A6	previous A6
	local variables
SP	saved registers

The function's code looks something like this:

```
LINK      A6, #...      ; (optional)
MOVEM.L   ..., -(SP)    ; (optional) save registers
...       ; code for the routine
```

The last argument can be found at 8 (A6). If there was a static link, the last argument is at 12 (A6). You can find the first local variable at a negative offset from A6. The value of the offset depends on the size of the variable.

Note: You generally don't need to worry about static links.

All arguments occupy 2 or 4 bytes on the stack. A byte argument appears in the high byte of its word and is found at an even offset from A6.

Pascal routine exit

When a Pascal routine exits, it is responsible for deallocating its stack frame and for removing any arguments from the stack. So the end of Pascal routine looks like this:

```
MOVEM.L   (SP)+, ...      ; (optional) restore registers
UNLK      A6              ; (optional)
MOVE      (SP)+, A0        ; return address in A0
ADD        #..., SP        ; total size of arguments
                        ; including static link if necessary
MOVE      ..., (SP)        ; store return result
JMP        (A0)            ; return to the caller
```

The code that THINK Pascal actually generates to return from a function may be slightly different because the compiler optimizes the stack cleanup. If you have the MC68020/MC68030 option on, for instance, THINK Pascal uses an RTD instruction to clean up the stack and return to the caller.

Parameter passing

The way you push parameters on the stack depends on the size and kind of parameter.

For this parameter...

Do this...

var parameter

Push the address of the actual parameter on the stack.

Value parameter

If the size of the actual parameter is 4 bytes or less, push the value of the parameter. All values take up either 2 or 4 bytes on the stack. A byte value appears in the high byte of the word, and you can find it at an even offset from register SP or A6.

If the size of the parameter is greater than 4 bytes, pass the address of the actual parameter. It is the responsibility of the called routine to make a copy of the parameter in case it is modified.

Procedural parameter

Push the address of the procedure or function, then push the static link to be used when the routine is actually called. If the procedure or function is declared in the outermost scope, pass a 0 for the static link.

Note: A procedural parameter is not the same thing as passing a pointer to a procedure or function.

Return Values

If the size of the return value of a function is 4 bytes or less, the caller allocates 2 or 4 bytes on the stack for it. When the function returns, this value is left on the stack.

If the size of the return value is more than 4 bytes, the caller allocates a temporary variable of the appropriate size and pushes its address as a "hidden parameter." When the function returns, the caller discards the address of the hidden parameter.

Register saving conventions

You can use registers D0, D1, D2, A0, and A1 freely in your assembly language routines. If you need to use other registers, be sure to save and restore them. If the machine you're programming for has a MC68881 or MC68882 floating point unit, you can use registers FP0, FP1, and FP2.

Note: Because registers A5, A6, and A7 are used to maintain vital state information between calls to subroutines, you should try to avoid using them except to access global variables, local variables, frame pointers, etc. If you push something on the stack, be sure to pop it off.

Using Assembly Language

You can use THINK C and Apple's Macintosh Programmer's Workshop (MPW) to write assembly language routines that you can use in THINK Pascal. You can add THINK C libraries and MPW object files directly to your THINK Pascal project. Read the sections below to learn how to use code from these development systems with THINK Pascal.

Note: Even though they may use the same format, THINK Pascal can't use some .o files created by some high-level language compilers, like MPW C or Pascal. You can use .o files created with the MPW assembler, as well as libraries created with THINK C. For more information, see "Using .o Files," Appendix C.

When you write a routine in assembly language, it's up to you to make sure that you follow the proper Pascal calling conventions and that you call it correctly from Pascal. For example, suppose that you wrote a function, `NewSysHandle`, that creates a new handle in the system heap. The function takes a `longint` that is the size of the handle, and it returns the new handle. To use the function, you would put this declaration in your Pascal program:

```
function NewSysHandle(size : longint) : Handle;
external;
```

The external directive tells THINK Pascal to look for `NewSysHandle` when it links your program. Presumably, the routine is in a library that you've added to your project.

THINK Pascal honors data initializations from imported THINK C libraries and MPW .o files. For example, take these declaration from a THINK C library:

```
char myString[] = "\psome string";
static ProcPtr myHook = &myFunction;
```

When the library is loaded, the value of `myString` will be `"\psome string"`, and `ProcPtr` will point to `myFunction`.

However, in desk accessories and drivers, libraries and .o files may not initialize variables to be pointers to other variables or to functions (that is, they may not use initializations that require run-time relocations). For example, THINK Pascal would honor the first initialization above in a desk accessory or device driver, but it would not honor the second.

Using THINK C with THINK Pascal

To write the `NewSysHandle` function in THINK C, write it like this:

```
pascal void Handle NewSysHandle (long size)
{
    asm {
        move.l    size, d0        ; size of handle in d0
        _NewHandle SYS            ; call the trap
        move.w    d0, 0x220       ; put result in MemErr
        move.l    a0, d0         ; put return in D0 for THINK C
    }
}
```

The `pascal` keyword instructs THINK C to use Pascal calling conventions. This way, you can let the compiler worry about following the rules while you write the meat of your routine in assembly language (or, if you prefer, in C).

If you'd rather take care of managing the stack yourself, you can write the routine this way:

```
void NewSysHandle(void)
{
    asm {
        movea.l    (sp)+, a1    ; get return address
        move.l     (sp)+, d0    ; size of handle in d0
        _NewHandle SYS          ; call the trap
        move.w     d0, 0x220    ; put result in MemErr
        move.l     a0, (sp)     ; put result on stack
        jmp        (a1)         ; return to caller
    }
}
```

To use Pascal global symbols from THINK C source files, just declare them `extern` and use them as you would any other C symbol.

After you compile the file, use the THINK C **Build Library...** command to create a library. Since you can load THINK C libraries directly into THINK Pascal projects, that's all you have to do.

To learn more about writing Pascal compatible routines in THINK C, see Chapters 10, "The Compiler," and 13, "Assembly Language," of the *THINK C User's Manual*.

Using Apple's Macintosh Programmer's Workshop

To write the `NewSysHandle` function with Apple's Macintosh Programmer's Workshop, write the assembly language file like this:

```
                INCLUDE    'SysEqu.a'
                INCLUDE    'Traps.a'

NewSysHandle    PROC      EXPORT
    movea.l     (sp)+, a1    ; get return address
    move.l      (sp)+, d0    ; size of handle in d0
    _NewHandle  SYS         ; call the trap
    move.w      d0, $220    ; put result in MemErr
    move.l      a0, (sp)    ; put result on stack
    jmp         (a1)        ; return to caller

                ENDP
                END
```

To access Pascal global symbols from MPW assembly language files, declare them `external` with the `IMPORT` directive and reference them from register A5 for applications or register A4 for desk accessories and device drivers.

Once you assemble the file, you can use the **Add File...** command to add an MPW object file to your THINK Pascal project.

LightsBug 14

Introduction

This chapter shows you how to use LightsBug, THINK Pascal's powerful debugging tool. LightsBug lets you get a close-up view of your program. When you stop your program, you can see all the procedures and functions that called the routine you're stopped in, the values of all the local and global variables, all the blocks in the application and system heaps, the CPU registers, and any part of memory.

What you should know

LightsBug knows about things like procedures, functions, arrays, and records, and it knows about low-level Macintosh objects like heap zones. For an overview of the Macintosh Memory Manager, see *Inside Macintosh I*, Chapter 3, "Macintosh Memory Management: An Introduction." To learn about the technical details of the Memory Manager, see *Inside Macintosh II*, Chapter 1, "The Memory Manager," and *Inside Macintosh VI*, Chapter 28, "Memory Management."

You'll find LightsBug more useful if you know something about memory organization and Pascal calling conventions. You can learn about these things in Chapter 13, "Assembly Language."

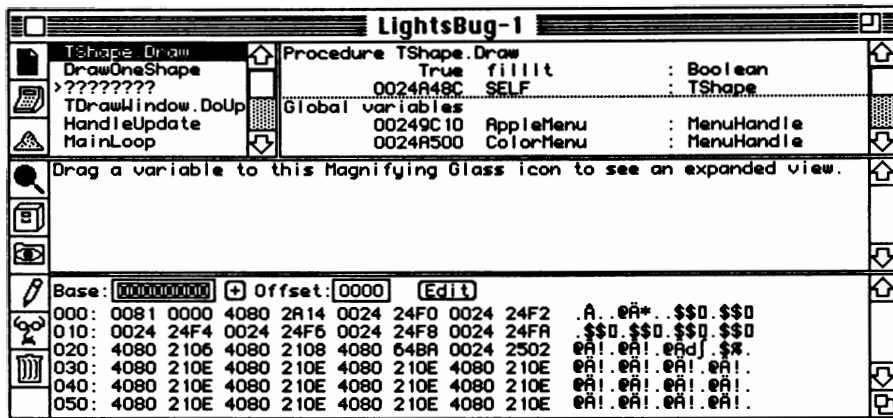
Topics covered in this chapter

- Using LightsBug
- Subroutine call chain
- Examining variables
- Examining structured variables
- Examining many variables
- Using watchpoints
- Type casting variables
- Examining registers
- Examining heap zones
- Displaying memory
- Editing memory
- Debugging Toolbox routines

Using LightsBug

LightsBug works best when you have both the Debug and the Names options turned on. To learn about these options, see Chapter 7, "Working with Projects," and Chapter 15, "Compiler Directives."

When you choose the **LightsBug** command from the **Debug** menu, you'll see a LightsBug window like this:



The LightsBug window looks more complicated than other THINK Pascal windows because it lets you work with a running program in several different ways.

The LightsBug window is divided into four **panes**. The upper right pane and the middle pane can have different **displays** depending on how you use LightsBug. It may sound complicated, but once you experiment a little, using LightsBug will become as easy as using other parts of THINK Pascal.

The upper left pane always shows you the subroutine call chain. The upper right pane displays subroutine variables, the CPU registers, and heap zones. The middle pane displays expanded views of variables, variables that you've put into a collected view, and watch points. The bottom pane always shows you the contents of your Macintosh's memory.

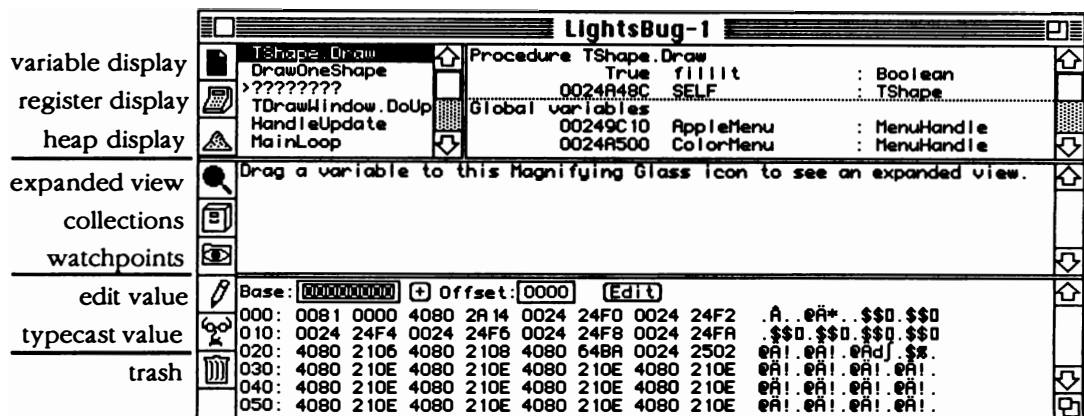
Working with panes

To make a pane larger or smaller, drag the double line that separates the panes. To get rid of a pane altogether, drag the double line to the edge of the window. To get a pane back, just drag the double line back from the edge of the window.

THINK Pascal lets you open up to four LightsBug windows. To create another LightsBug window, choose the **New LightsBug** command. To see the **New LightsBug** command, hold down the Shift key as you select the **Debug** menu. If there is more than one LightsBug window on the screen, choosing the **LightsBug** command cycles through all the open windows.

The LightsBug Icons

The icons along the left edge of the LightsBug window control the displays in the LightsBug panes. The icons are divided into four groups.



The first group of icons controls the display in the upper right pane of the LightsBug window.



Variable display. When you choose this icon, the pane displays all the variables visible to the routine selected in the subroutine call chain.



Register display. When you choose this icon, the pane displays all the CPU registers in hexadecimal. If you have the 68881/68882 option on, the pane also displays the floating point registers.



Heap display. When you choose this icon, the pane displays your application heap zone or the system heap zone.

The second group of icons controls what you see in the center pane. These icons work like containers — like folder icons in the Finder. You drag values into them.



Expanded view. When you drag a variable name from the variable display into this icon, the center pane displays its value. If the variable is a structured type (a record, an array, or a set), the display shows you all the fields or elements.



Collected views. This container is like the expanded view container, but it can hold more than one value at a time. You can place several different values here so you can keep track of them all at once.



Watchpoints. This container also holds several values at once. Whenever one of the watchpoints changes, THINK Pascal stops your program.

The third group of icons lets you edit values. These icons work like filters. You drag something into them and get something else.



Edit value. When you drag a value into this icon, LightsBug displays a dialog box that lets you edit the value.



Type cast value. When you drag a value into this icon, LightsBug displays a dialog box that lets you change the type of a value. For example, you can change a generic handle into a `ControlHandle`.

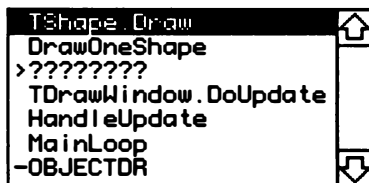
The last icon is like the Trash icon in the Finder. It lets you remove values from any of the containers.



Trash. When you drag a value from any of the containers — expanded view, collection views, or watchpoints — to this icon, the value is removed from the container. Unlike the Finder, you can't drag things out of the trash in LightsBug.

Examining Subroutines

The upper left pane of the LightsBug window is the **subroutine call chain**. It shows you the list of all the subroutines that are currently active — all the subroutines that have been called but haven't returned yet. The most recently called subroutine is at the top of the list.



LightsBug does its best to find the names of the subroutines in the call chain. When the subroutine is compiled with the Debug option on, and it is in an open window, LightsBug uses the full name of the routine. Otherwise, LightsBug uses the name stored in the code when the Names option is on. (See Chapters 7 and 15 to learn more about the Names option.)

LighsBug uses a system of case conventions and prefixes to let you know how the subroutine was compiled.

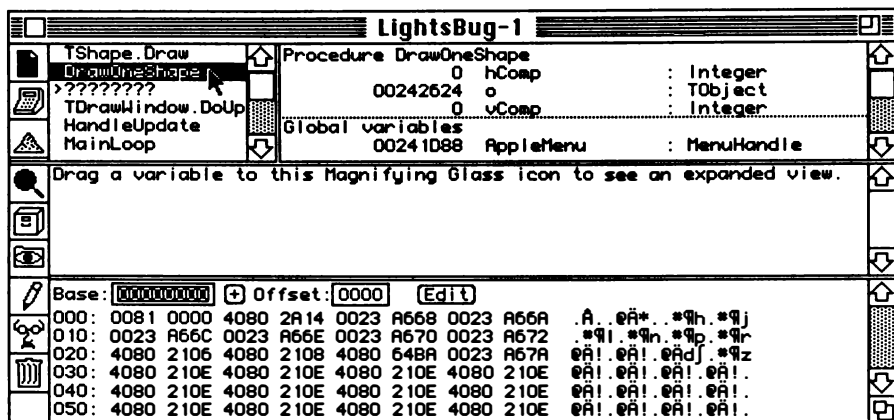
Name Display	Meaning
MyProc_Name	The subroutine is in an open editing window and was compiled with the Debug option on, or the subroutine was compiled with both the Names and the Full Length names options on.
MYPROCNA	The subroutine is not in an open editing window, or the subroutine was compiled with both the Names on and the 8-Character names options on.
????????	The routine was compiled with the Names option off, and either it is not in an open editing window or it was compiled with the Debug option off.
no prefix	The subroutine is in an open editing window and was compiled with the Debug option on.
-	The subroutine is not in an open editing window and was compiled with the Debug option on.
>	The subroutine was compiled with the Debug option off.
*	There was a gap in the call chain. This usually happens when an assembly language routine uses register A6 in an unusual way.

In the picture above, OBJECTDR is the program name. OBJECTDR called a procedure MainLoop which called HandleUpdate which called TDrawWindow.DoUpdate. TDrawWindow.DoUpdate called a routine which was compiled with the Names option off. It is listed as ????????. That routine in turn called DrawOneShape which called TShape.Draw. Note that OBJECTDR is marked as not being in an open editing window. ???????? is marked as having been compiled with the Debug option off.

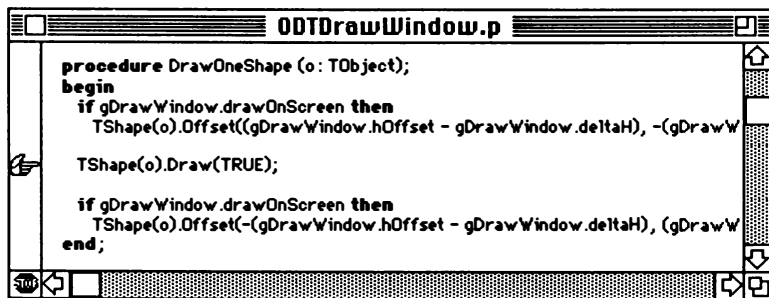
When a subroutine is called recursively, its name will appear more than once in the subroutine call chain.

Finding a routine definition using the call chain

You can use the subroutine call chain to find the definition of a procedure, function, or method. When you double-click on a routine, THINK Pascal opens the file that contains the routine and points to the line it last executed (that is, where it called the next routine in the chain). For example, if you double click on DrawOneShape:

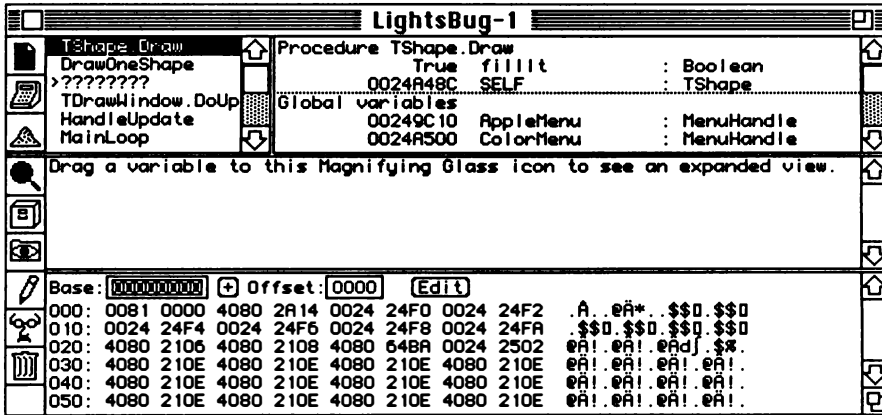


you'll see this editing window:



Examining Variables

When you select a subroutine name from the subroutine call chain, the upper right pane becomes the **variable display**. This display shows you the names, types, and values of all variables (including parameters and global variables) visible to the subroutine.



The variables are listed in alphabetical order.

Variables and scope

LightsBug displays all the variables visible to the selected subroutine according to Pascal's scoping rules. That means that every variable you could reference in the selected routine appears in the variable display. If the selected routine is nested within another routine, you'll see the variables for the enclosing routine.

A dotted line separates the variables for each routine. The first line of text under the dotted line tells you whether the routine is a procedure or a function. If the routine is a function, LightsBug displays its return value right under the name.

Global variables appear at the end of the variable display. Since the display shows only the variables that are visible to the selected subroutine, you'll see only the global variables that are in units known to the subroutine.

Variable display formats

LightsBug displays the value of a variable in the most natural format for its type:

Variable type	Display format
INTEGER	12, -73 (decimal)
CHAR	'F', CHR(13)
String	'A String'

Variable type

Enumerated, BOOLEAN
 Pointer, Handle, Object
 Array, Record, Set

Display format

TRUE, FALSE, RED, WHITE
 00031F36 (hexadecimal value)
 <array>, <record>, <set>

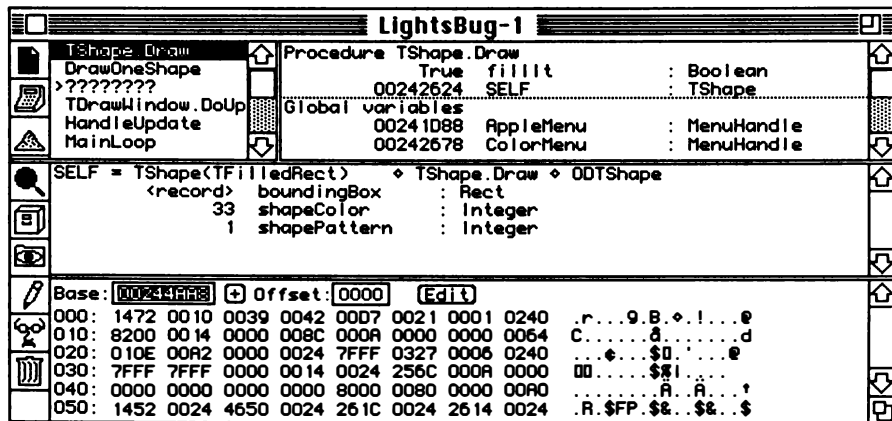
If a character isn't in the printable range, LightsBug uses the form CHR (n) so you can see its value. A carriage return, for instance, would be displayed as CHR (13) .

Note: When you click on a variable, the memory pane displays the memory it occupies. See "Displaying Memory" below.

Examining Structured Variables

When you double-click on a variable in the variable display, LightsBug displays its **expanded view** in the middle pane. If the variable is a structured type (a record, an array, a set, or an object), LightsBug shows you all the fields of the record or all the elements of the array. For example, if you double-click on the value of a record, you'll see all the fields of the record and their values in the expanded view pane.

This is what the LightsBug window looks like if you double-click on the self variable in the variable display pane:



The top line in the expanded view gives you the name of the variable and its type. If the variable is an object, LightsBug displays it using its declared type and shows its runtime type in parentheses, as in the example above. If the variable is a handle or a pointer, LightsBug dereferences the variable to its base type. The top line also tells you the name of the routine and unit that your variable is defined in.

Note: LightsBug dereferences generic handles and pointers as if they were of the type SignedByte. If you know the type of the object a generic pointer or handle

refers to, you can use type coercion to display it. You can also use type coercion to display an object using its runtime type. See "Type Casting Variables" below.

The fields of records appear in declaration order, and the elements of arrays appear in order. The expanded view displays values the same way as the variable display, so if a record contains another record, its value is <record>.


Note: If an array has more than 32,000 elements, LightsBug displays only the first 32,000. The array has not been truncated.

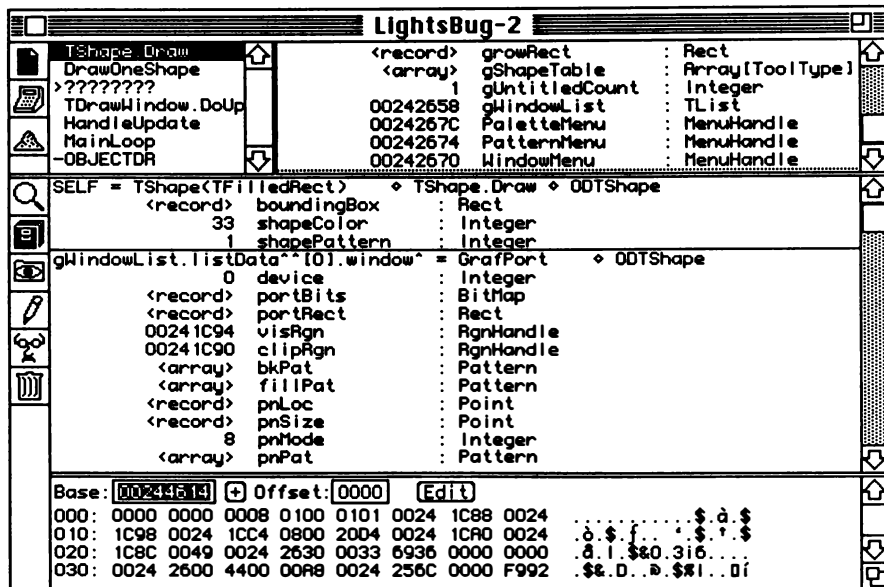
To see an expanded view of a field of a record, an element of an array, or a set, just double-click on it. The field or element's expanded view replaces the current expanded view.

LightsBug remembers each time you expand a value. You can use the arrow keys to follow the chain of expanded views. For example, suppose you're looking at an event record in the expanded view. If you double-click on the where field of the event record, the expanded view displays it as a point. Pressing the Up or Left Arrow key would return you to the event record display. From there, pressing the Down or Right Arrow key would show you the point display.

Note: You can also use the < and > (or Command and Period) keys. Use the < (or Comma) key instead of the Left arrow and use the > (or Period) key instead of the Right Arrow.

Examining Many Variables

The expanded view lets you see only one variable at a time. To see several variables at a time, you can save them in the **collected views**. To put a variable in the collected views, just drag it from either variable display or the expanded view to the collected view icon . When you click on the collected view icon, you'll see all the variables in the middle pane.



You can have global variables and variables from different procedures and functions in the collected view. As your program runs, variables that belong to some routines will go out of scope. When this happens, their entry in the collected view says no context. If your program enters that routine again, you'll see their full values again.

To remove a variable from the collected views, just click on the first line of the variable and drag it to the trash icon.

Note: You can't drag items out of the trash like you can in the Finder.

Using Watchpoints

The watchpoints container is just like the collected views container. The difference is that whenever the value of a variable in the watchpoints container changes, THINK Pascal stops your program and displays a "thumbs down" next to the line where the change was detected. Usually, the "thumbs down" points to the line *following* the line that actually changed the value.


To turn off the watchpoints feature, drag the watchpoints to the trash.

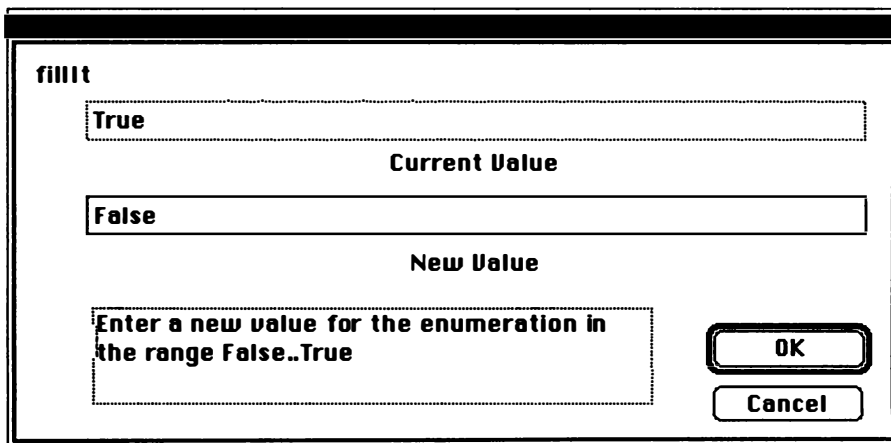
Use watchpoints to debug strange behavior in your program. For instance, suppose you have a pointer that somehow ends up pointing to the wrong thing while your program is running. If you put the pointer in the watchpoints container, your program will stop at the statement that's changing the value of the pointer.

To make the watchpoints feature work, THINK Pascal has to look at all the values in the watchpoints container after every statement to see if they've changed. This checking will make your program run significantly slower. When you drag a variable to the watchpoints container, THINK Pascal makes a copy of it. After each step, it compares the current value of the variable to the saved value. If you drag large variables to the watchpoints container (for instance, an array that takes up 100K), you might find that THINK Pascal runs out of memory.

Note: You can double-click on a value displayed in the watchpoints pane to see its expanded view.

Editing Variables

You can use LightsBug to change the values of variables. To change a value, drag the variable into the edit value icon . You'll see a dialog box like this:




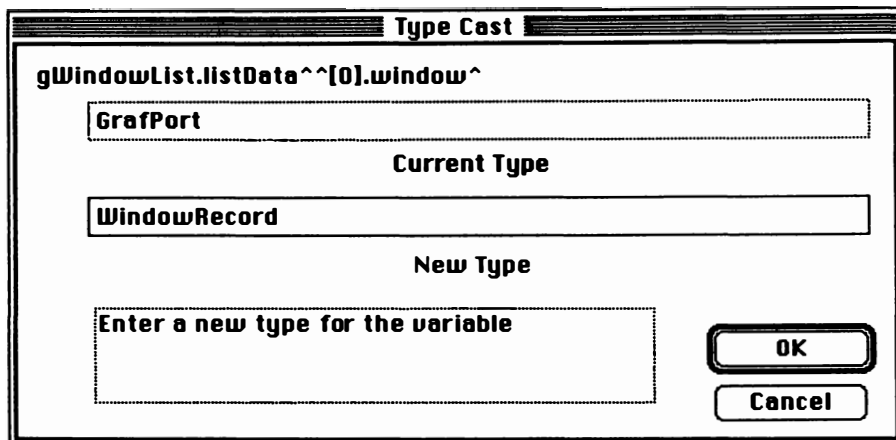
Type the new value of the variable in the New Value field, click on the OK button. LightsBug will change the value if the new value falls within the legal range.

Type Casting Variables

LightsBug lets you look at objects of one type as if they were objects of another type. This operation is called **type casting**. For example, *Inside Macintosh* defines a `WindowPtr` as a pointer to a `GrafPort` instead of as pointer to a `WindowRecord`. To examine the `WindowRecord` fields of a variable declared as a `WindowPtr`, you need to type cast it. Type casting is also useful when you want to look at an object using its runtime type instead of its declared type.

Note: Sometimes you cannot coerce an object to its runtime type because the definition of the runtime type is outside the scope of the function you're debugging.

To type cast a variable, drag it from the variable display or any view in the center pane to the type cast icon . You'll see a dialog box like this:









Type the new type for the variable in the New Type field, and click OK. LightsBug changes the type to the new type and displays its expanded view.

Note: When you type cast a variable, the type of the original variable doesn't change. What changes is the way it's displayed in the expanded view

Examining Registers

To display the values of the CPU registers, click on the register icon  in the LightsBug window. The upper right pane becomes the **register display**. The values of the registers appear in the register display like this:


	TShape Draw		A0=00238938	D0=00000401	
	DrawOneShape		A1=00321AE6	D1=00280000	
	>????????		A2=00000000	D2=00241C34	
	TDrawWindow.DoUp		A3=00000000	D3=00000000	
	HandleUpdate		A4=00242624	D4=00000000	
	MainLoop		A5=00266E36	D5=00000000	

If your Macintosh has either the MC68881 or MC68882 floating point unit, and you have the 68881/68882 option on, the values of the floating point registers appear in the register display, too.

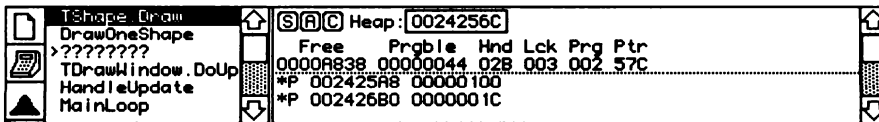
When you select a register, the value of the register becomes the new Base for the memory display. (See "Displaying Memory" below.) Usually you'll want to select register A7 (the stack pointer) to see what's on the stack.

Note: If you need to edit the values of the registers, use a low level debugger like Macsbug or TMON. Unless you know *exactly* what you're doing, it's not a good idea to change the values of the registers. Be careful. See "Examining Compiled Code" at the end of Chapter 9, "Debugging Programs."

Examining Heap Zones

When you select the heap icon  in the LightsBug window, the upper right pane becomes the **heap display**. The heap display lets you examine the pointers and handles allocated in any heap zone. To learn about heap zones, see *Inside Macintosh I*, Chapter 3 and *Inside Macintosh II*, Chapter 2.

The heap display looks like this.



The three buttons at the top of the pane let you choose which heap to display:

Button	Heap zone displayed
S	System heap
A	Application heap
C	Current heap

You can also enter the address of a heap zone in the box next to the buttons. If you enter an address that's not a heap zone or if the heap is damaged, LightsBug gives you an error message.

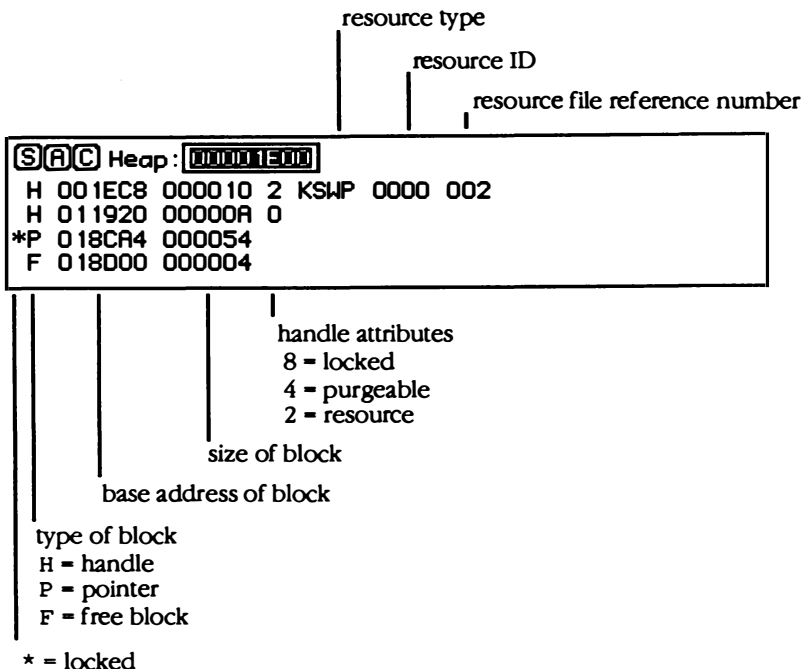
Note: All values in the heap display are in hexadecimal.

Unless you changed the heap zone with the Toolbox routine SetZone, the current zone is the same as the application zone.

Right under the heap zone buttons you'll see a summary of what's in the heap:

Heading	Meaning
Free	total size of all free objects in bytes
Prgble	total size of all purgeable objects in bytes
Hnd	number of handles allocated
Lck	number of locked handles
Prg	number of purgeable handles
Ptr	number of pointers allocated

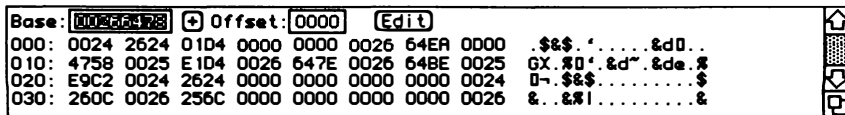
After the summary, every line in the heap display describes an item in the heap.



To see the contents of an item in the heap, click on it. Its contents will appear in the memory display. The next section tells you how the memory display works.

Displaying Memory

The bottom pane of the LightsBug window is the **memory display**. Memory appears in hexadecimal starting at the memory location Base+Offset. An ASCII display of the same memory appears to the right of the hex display.



There are several ways you can choose which memory is displayed. They all involve changing the base address or the offset.

To display a certain location in memory, type in its address in the Base field and press the Return or Enter key. You can also specify an offset in the Offset field. If you click on the + button, THINK Pascal adds the offset to the Base and resets the offset to zero.

Note: You can press the ~ key to undo changes you made to the Base or Offset field.

To display an item in the heap, click on it in the heap display pane. If the item is a pointer, the pointer becomes the Base address. If the item is a handle, the dereferenced handle becomes the Base address. The offset is set to zero.

To display the memory a register points to, click on a register value in the register display pane. The value of the register becomes the Base address, and the offset is set to zero.

To display the memory a variable occupies, click on a variable in the variable display pane or in any of the views in the center pane. The location in memory that holds the beginning of the variable becomes the Base address, and the offset is set to zero.

To see the frame pointer for a subroutine or function, click on its name in the variable display pane. The address of the stack frame becomes the Base address, and the offset is set to zero.

You can use the mouse to change the offset field. If you click on any byte in the memory display, it becomes the first byte displayed. The Base address doesn't change, but the Offset is updated accordingly. If you hold down the Shift key as you click on a byte in the memory display, the first byte in the display slides to the point you clicked on.

You can dereference objects in memory. Click on a byte in memory as you hold down the appropriate keys. Depending on the keys, THINK Pascal treats the four bytes after the cursor as either a pointer or a handle and shows you what it refers to. This chart shows you which keys to hold down:

To deference a...	Click on a byte as you hold down the...
Pointer	Option key
Handle	Option and Command keys

THINK Pascal sets the Offset to zero. If you dereference a pointer, THINK Pascal sets the Base address to the four bytes after the cursor. If you dereference a handle, THINK Pascal sets the Base address to the address that the handle points to.

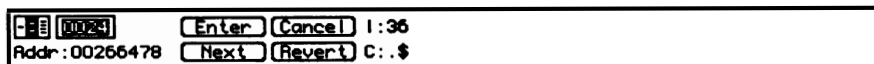
LightsBug remembers the last eight changes to the base address. You can recall them with the < and > keys.

Note: The offset field is sign extended (i.e. if you type in a minus sign followed by a hex number, it will compute the proper value).

Editing Memory

You can edit memory displayed in LightsBug. Be sure you know what you're doing before you change any byte in memory.

To edit memory, click on the small Edit button in the memory display pane or type Shift-E. A small window appears on top of the memory display:



The Addr field displays the address of the memory you're editing. Initially, this address is the same address as the base address of the memory display.

The rectangles above the Addr field let you edit one, two, or four bytes at a time. The rectangles contain one, two, or four lines to indicate the number of bytes being edited. In this example the rectangle with two lines is selected, so the edit box displays the two bytes of memory starting at Addr.

To edit memory just type a value in the box. When you are done typing you can click on any of the four buttons in the edit box:

Button	Action
Enter	This button causes the change to take effect. This changes memory at Addr to the value just typed. You can also use the Return and Enter keys.
Cancel	This button cancels editing mode. You can also use Shift-~.
Next	This button causes the change to take effect, and displays the next byte, word, or longword of memory. You can also use the Tab key.
Revert	This button restores the edited value to what it was previously. You can also use the ~ key.

On the right side of the editing box, memory is displayed in different formats, with a letter identifying each format. Some formats may not be displayed depending on whether you're editing one, two, or four bytes.

Symbol	Format
C	characters
I	integer
L	longint
S	signed byte
U	unsigned byte

You can use the < and > keys to move through memory when you're in edit mode. These keys change the offset in the memory display by one, two, or four bytes.

Compiler Directives

15

Introduction

This chapter describes the compiler directives you can use in your THINK Pascal programs and the options you can set to control code generation. A **compiler directive** is an instruction that tells the compiler how to compile a specific file or part of a file.

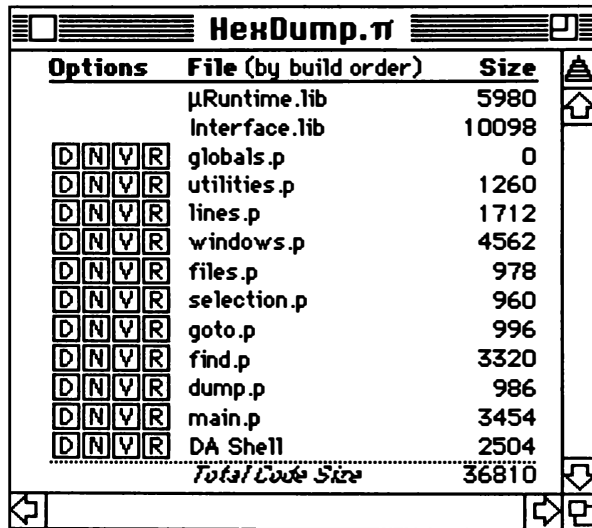
You've already seen some of the THINK Pascal compiler directives as **compiler options** in the project window — the four letters D, N, V, and R. In this chapter you'll learn more about what these directives do and how you can control them from within your source code. You'll also learn about other directives that not only let you control how THINK Pascal compiles your source files but also what part of your source files it will compile.

Topics covered in this chapter

- What are compiler directives?
- Using compiler directives
- Using conditional compilation
- Using the Compile Options command

What Are Compiler Directives?

You're already familiar with four compiler directives from Chapter 7. These are the four letters in boxes in the Options column of project window. These **compiler options** control code generation for the file. When you add files to a project, the D and N options are already on. As you work on your program, you can turn the options on and off for specific files.



Options	File (by build order)	Size
	μRuntime.lib	5980
	Interface.lib	10098
D N V R	globals.p	0
D N V R	utilities.p	1260
D N V R	lines.p	1712
D N V R	windows.p	4562
D N V R	files.p	978
D N V R	selection.p	960
D N V R	goto.p	996
D N V R	find.p	3320
D N V R	dump.p	986
D N V R	main.p	3454
D N V R	DA Shell	2504
	<i>Total Code Size</i>	36810

Option Meaning

- D** **Debug.** Generates additional information between each Pascal statement to support stepping, stopping, and observing.
- N** **Names.** Inserts the name of all routines into the code. This is useful for debugging with LightsBug or low-level debuggers.
- V** **Overflow Checking.** Generates code that checks for integer arithmetic overflows.
- R** **Range Checking.** Generates code that does range checking for array indexing, assignments, and parameter passing. It also generates code that checks for **nil** pointer dereferencing.

To disable an option, click on the appropriate letter. The box around the letter disappears. To enable the option again, click on the letter. The box reappears.

Note: If you hold down the Option key as you click on a compiler option, THINK Pascal will change the option for all the files in the project.

Using compiler directives in your source files

You can also turn these compiler options on and off from your source file. When you use the compiler options in your source files, they are called **compiler directives** because they direct the compiler to generate different code. In addition to the four directives you can set as compile options, THINK Pascal provides seven more.

A compiler directive looks like this:

```
{ $D+ }
```

A compiler directive is a comment that begins with a \$. The text after the \$ specifies which directive and whether it's on or off. In this example, D+ means that you want to turn the setting on. A minus sign would mean that you want the setting off.

Note: There are no spaces between the { and the \$.

THINK Pascal supports these nine compiler directives:

Directive	Meaning
{ \$D+ } or { \$D- }	Turns Debug option on or off
{ \$N+ } or { \$N- }	Turns Names option on or off
{ \$N++ }	Turns Tracing and Names options on.
{ \$V+ } or { \$V- }	Turns Overflow Checking option on or off
{ \$R+ } or { \$R- }	Turns Range Checking option on or off
{ \$I+ } or { \$I- }	Turns automatic initialization on or off (only in main program)
{ \$J+ } or { \$J- }	Turns External Variable handling on or off.
{ \$Z+ } or { \$Z- }	Turns External Routine handling on or off.
{ \$S <i>name</i> }	Puts the following code into the segment <i>name</i> .

Note: The MPW Pascal { \$D± } directive corresponds to the THINK Pascal { \$N± } directive, not the THINK Pascal { \$D± } directive. Some MPW directives don't have analogous directives in THINK Pascal.

THINK Pascal also supports these two related directives:

Directive	Meaning
{ \$PUSH }	Save the current settings of the D, N, V, J and Z compiler directives
{ \$POP }	Restore the settings of the D, N, V, R, J and Z compiler directives from the previous { \$PUSH }.

Finally, THINK Pascal recognizes this directive when it prints a file:

Directive	Meaning
{ \$P }	Start printing on a new page.

The first four directives — D, N, V, and R — correspond to the four compile options in the project window. The difference between the compiler options and the compiler directives is that the

embedded compiler directives let you control code generation at the procedural level or even the statement level. For example, one common technique is to turn off the Range Checking option when you want to change the length byte of string:

```
{ $R- }
myString[0] := 0;    { pretend it's a null string }
```

To use a compiler directive, you put it right before the line you want to affect. That directive remains in affect until the end of the file or until you turn it off (or on) again. In the example above, the Range Checking option will remain off until the end of the file or until you turned it on again with { \$R+ }.

If a compiler option — D, N, V, or R — is turned off for a file in your project, you can't turn it back on with a compiler directive in your file. In other words, the compile options in the project work like a "master switch" for the compiler directives in your file.

If you use the Debug, Name, External Routine, or Tracing directives, you must change their state before the first begin of a procedure or function, or they won't have any effect until the *next* procedure or function. For example, to turn off debugging for a procedure, you'd write this:

```
{ $PUSH }
{ $D- }
{ Don't generate debugging code for this procedure }
procedure SalmonSushi;
begin
    ...
end;
{ $POP }
```

But this won't work:

```
procedure Pakora;
begin
    { $PUSH }
    { $D- }
    ...
end;
{ $POP }
```

THINK Pascal will still generate debugging code for procedure Pakora.

How compiler directives work

When you turn on one of the D, N, V, or R compiler directives, THINK Pascal weaves "invisible" code into your program. For example, the D directive inserts code that lets THINK Pascal know what your program is doing, so you can use its powerful debugger. Other options, V and R, for instance, insert error checking routines to check for overflow or out-of-range conditions. When you turn on more than one option, you get the extra code for each option.

When you change a compile option for a file, THINK Pascal recompiles the file to generate the new code before you run.

Using Compiler Directives

This section describes each compiler directive in detail. You'll learn exactly what the directive does and when you should use it. Since every compiler directive generates extra code, you'll also learn what it costs you — in terms of speed and memory — to use the directive.

Debug {\$D±}

When you turn the Debug option on, THINK Pascal generates code that lets you use all of the THINK Pascal debugging features. This code makes it possible for THINK Pascal to display a thumbs-down icon next to an error that occurs while your program's running. For example, if your program tries to divide by zero, or tries to write beyond the end of a file, or tries to assign an out-of-range value to a variable, the debugging code lets THINK Pascal know which file and which line contains the error.

This directive also generates code that looks out to make sure your stack doesn't collide with the heap. If the stack were to run into the heap, you'd get a System Error ID=28. The error would damage the THINK Pascal environment, and you'd have to reboot and start again. Typically, these errors are hard to track down.

The debugging directive acts as a kind of safety net, so it's a good idea to leave this option on while your program is still unstable.

If you use the {\$D±} directive in your source code, be sure to put it before the first **begin** of the procedure or function you want to affect.

Cost	The Debug directive increases your code size by about 30%. Your program runs 2 to 10 times slower.
Limits	You can't have the Debug option on in a standalone application or in a library. When you choose one of the Build... commands from the Project menu, THINK Pascal turns the Debug directive off automatically.
Uses	If you plan to use a low-level debugger like TMON or Macsbug to debug a piece of code, it's best to turn the Debug directive off so the additional debugging code doesn't get in your way when you look at the assembly language listing.

Names {\$N±}

The Names directive embeds the subroutine name into the code right after the end of the procedure or function. LightsBug uses this name when the source file that contains the routine isn't open. Other low-level debuggers like TMON or Macsbug use these names as well.

THINK Pascal can store the names two ways, depending on how you set the Long names option:

- If the Long names option is off, names are limited to eight uppercase characters. Underscores are stripped off. Names shorter than eight characters are padded with spaces. For methods (e.g., `TDrawWindow.FileWrite`), only the first eight characters of the class and method names are stored (e.g., `TDRAWWIN.FILEWRIT`). LightsBug, TMON, and all versions of Macsbug can use eight-character names.
- If the Long names option is on, names can be any length, all uppercase. For methods (e.g., `TDrawWindow.FileWrite`), the full class and method names are stored (e.g., `TDRAWWINDOW.FILEWRITE`). LightsBug and Macsbug 6.0 and later can use full-length names.

To see the Long names option, choose **Compile Options...** from the **Project** menu. For more information on that command, see "Using the Compile Options Command" at the end of this chapter

Cost	Up to 255 extra bytes per procedure or function. No speed penalty.
Limits	Eight-character names are limited to eight characters. Full-length names are limited to 254 characters.
Uses	Turn the Names option on whenever you're using LightsBug, TMON, or Macsbug to help you find your way around your routines. It's usually a good idea to turn this option off before you create your final release.

Tracing {\$N++}

The Tracing {\$N++} directive is included for compatibility with MacApp. It lets you trace your procedures and functions with the MacApp debugger. The {\$N++} directive also turns on the Names option, since the MacApp debugger's tracing routines print out function and procedure names.

Note: MacApp turns on the Tracing and Names options automatically when you use the MacApp debugger. Most of the time, you won't need to turn on the Tracing directive yourself. For more information on the MacApp debugger, see your MacApp documentation.

If your project doesn't use MacApp, you must define these procedures to do your own tracing:

Declaration	Description
procedure %_BP;	Called at the beginning of every procedure.
procedure %_EP;	Called at the end of every procedure.
procedure %_EX;	Called before an exit or a non-local goto.

To turn off the Tracing option, use either { \$N+ } or { \$N- }. The { \$N+ } directive leaves the Names option on, and the { \$N- } directive turns the Names option off. This chart summarizes what these directives do:

This directive...	turns the Names option...	and the Tracing option...
{ \$N++ }	On	On
{ \$N+ }	On	Off
{ \$N- }	Off	Off

Overflow { \$V± }

The Overflow directive generates code that detects errors resulting from numeric overflow in arithmetic operations. The result of each intermediate operation must be within the bounds of numerical representation. For type `integer`, the bounds are -32768 to +32767. For type `longint`, this range is -2147483648 to +2147483647.

For example, consider this procedure:

```

procedure Boom;
var
    i, j, k : integer;
begin
    i := 20000;
    j := 21000;
    k := i + j;
end;

```

You might expect that `k = 41000`, but since the largest integer is 32767, `k` would overflow (-24536), and you'd have an error. If the Overflow directive is on, THINK Pascal will catch this kind of error.

The Overflow directive doesn't check floating point operations, but the SANE library does. The SANE environment lets you control the effects of overflow in floating point operations. See the *Apple Numerics Manual, Second Edition* to learn more about SANE.

Cost	Two extra bytes generated for every arithmetic expression. Your program runs 5% to 10% slower.
Limits	The Overflow works best while your program is still under development, and you're still working in the THINK Pascal environment. You can examine the variables that caused the overflow and correct their values in the Observe or LightsBug windows. In a standalone program, recovery is more difficult, if not impossible.
Uses	Use the Overflow directive to track down potential problems and to make sure that your program deals with errors appropriately.

Range {\$R±}

The Range directive makes sure that values fall within legal bounds. For variable assignment, this directive checks that a value assigned to a variable is in the legal range for the variable. For instance, when your program assigns a value to a variable of type `integer`, it makes sure that the value falls in the range `±Maxint` (-32768 to +32767).

The Range directive also makes sure that the index of an array falls within the bounds of the index type. For instance, suppose your program contains this declaration:

```
var
    SomeArray : array [1..25] of INTEGER
```

The Range Checking directive would catch an illegal error like:

```
SomeArray[27] := 1961;
```

Note: If the index is a constant, the Range directive catches the error at compile time. Otherwise, it catches the error while the program is running.

Strings are a special case. When the Range directive is on, it makes sure that the character being accessed is in the range of the string's current length. For example:

```
type
    s25 = string[25];
var
    MyString : s25;
begin
    MyString := 'abc';
    MyString[4] := x;    { ERROR.}
    ...
end;
```

In fact, the most common use of the {\$R±} directive is to turn it off while your program manipulates strings. For instance, if you wanted to write your own version of the predefined `length` function, you'd have to write it like this:

```
{ $PUSH }
{ $R- }
function MyLength (s:string): integer;
begin
    MyLength:= ord(s[0]); {Compiles ONLY if Range option is off}
end;
{ $POP }
```

For more information about how THINK Pascal deals with strings, see Sections 3.3 and 4.3.1 of Chapter 17.

Finally, the Range directive checks to see if a pointer is `nil` before it dereferences it. The directive won't keep you from dereferencing an odd or invalid pointer, though.

Cost	Four to 24 bytes extra code generated for assignments, array accesses, and pointer dereferences. Your program runs about 10% to 20% slower.
Limits	Like Overflow checking, Range checking is easy to fix while you're still in the THINK Pascal environment. Error recovery is difficult and impractical in standalone programs.
Uses	Turn the Range Checking directive on while your program is still under development. You'll probably want to turn it off when you want to manipulate strings directly.

Initialization {\$I+}

As mentioned in Chapter 12, THINK Pascal usually takes care of initializing all the Toolbox managers for you. You can use the Initialization directive to turn this feature off if you prefer to initialize the managers yourself. The {\$I+} directive tells THINK Pascal to initialize the managers automatically. The {\$I-} directive lets THINK Pascal know that you're going to handle the initializations.

When the Initialization directive is on, THINK Pascal calls these Toolbox routines:

```
InitGraf(@thePort);
InitFonts;
InitWindow;
InitMenus;
TEInit;
InitDialogs(nil);
SetApplLimit (value of A7 - value of Run Options... stack size);
MaxApplZone;
for i := 1 to 10 do
    MoreMasters;
```

The {\$I-} directive should appear before the begin of your main program. THINK Pascal ignores this directive in units.

Note: THINK Pascal never does automatic initialization for desk accessories, device drivers, or code resources.

You'll usually use the {\$I-} directive when you port Pascal code that contains the Toolbox manager initializations.

Note: You need to insert {\$I-} in most programs you port from other Macintosh compilers.

If you checked the 68881/68882 option in the **Compile Options...** dialog, THINK Pascal also initializes the state of the MC68881 or MC68882 floating point unit (FPU). This means that if you want your application to check for a FPU, you should turn the Initialization directive off. Otherwise, your program will crash at startup on a machine that doesn't have a FPU. To learn more

about the 68881/68882 option, see "Using the Compile Options Command" at the end of this chapter.

To initialize the FPU state yourself, use this procedure:

```
procedure InitFPState;
inline
    $42A7,      { clr.l      -(sp)          }
    $42A7,      { clr.l      -(sp)          }
    $F21F, $9800; { fmovem.l  (sp)+, fpcr/fpsr }
```

External Variable {\$J±}

The External Variable directive {J±} tells THINK Pascal not to allocate space for the following variables. Use it to access variables defined elsewhere, such as in a library or in an assembly language file. If you don't define a variable elsewhere with the same name, you'll get a link error. For example, this is the interface file for a library that declares the external variable ioStatus:

```
unit io;
interface
{$PUSH}
{$J+}
    var
        ioStatus: integer;
{$POP}
...
```

Note: At the end of the variable declaration section, you must turn off the External Variable directive (with either {\$J-} or {\$POP}).

You can also use the external variable directive to create libraries of objects. When THINK Pascal comes across a class declaration, it generates code to help resolve calls to its methods. Without the external variable directive, THINK Pascal generates this code twice: once in the library and again in the interface file. Use the external variable directive in the interface file to avoid generating the code a second time. For example, this class declaration will not generate it:

```
type
{$PUSH}
{$J+}
    MyWindow = object (Window)
        windowData: Handle;
        subWindow: ToolWindow;

        procedure Hit (where: Point);
        override;
        procedure Draw;
        override
    end;
{$POP}
```

External Routines {\$Z+}

The External Routine directive `{ $Z+ }` makes the procedure or function that follows it externally visible. When a routine is externally visible, any unit in a project — even a unit that comes before the routine's unit in the build order — can declare the routine as `external` and use it.

This directive is useful when you're writing a large project that contains circular dependencies; for example, unit A requires a function in unit B which requires a function in unit A. If you've never come across a circular dependency, you should use a unit's interface section to declare the functions and procedures that other units need. For more information on the interface section and the best way to write units, see Chapter 10, "Units and Libraries."

Push {\$PUSH} and Pop {\$POP}

The Push directive saves the current settings of the D, N, V, R, J, and Z compiler directives. The Pop directive restores these settings. This lets you change the setting of a compiler directive for one or more routines, and then restore it to its original setting, without having to know the original setting. Here's an example:

```
{ $PUSH }
{ $R- }
procedure MyProc;
  begin
    ...
  end;
{ $POP }
```

This turns off the Range Checking option for the procedure `MyProc`, and restores it to its original setting when the procedure is done.

Segmentation {\$S name}

The Segmentation directive, `{ $S name }`, places the code for all following procedures and functions (up to the next segmentation directive or the end of the file) into the segment *name*. This directive lets a single unit contribute code to several Macintosh segments.

Note: Don't use `%_SelProcs` or `%_MethTables` as segment names. They are reserved by THINK Pascal.

Case is significant in segment names. For example, `InitSeg` and `initseg` are different segments.

The linker places functions and procedures into the file's segment if

- no Segmentation directives appear before the routines
- a `{ $S Main }` directive is before the routines
- a `{ $S }` directive (with no segment name) is before the routines.

Note: Don't place a segmentation directive in a unit's interface section. If you do, that will place the routines of any file that uses that unit into that segment.

For more information on segmentation, see Chapter 7, "Working with Projects."

Page Break {\$P}

THINK Pascal recognizes the Page Break directive { \$P } only when it's printing a file. When it sees the directive, it starts printing on a new page. The Page Break directive is the first thing on the new page.

Using Conditional Compilation

The compiler directives described above let you control how THINK Pascal generates code in specific instances. The **conditional compilation** directives, though, let you generate different code for different circumstances. For instance, while you're debugging your application, you might want to have additional debugging menus or particular routines that help you see what your application is doing. Or you might want to create two versions of your application: one that runs on any Macintosh and another one that runs only on a Macintosh with a floating point processor.

The four conditional compilation directives are:

```
{ $SETC identifier = expression }
{ $IFC expression }
{ $ELSEC }
{ $ENDC }
```

These compiler directives work like Pascal statements. An identifier can be any legal Pascal identifier. An expression can consist of identifiers, integer constants, boolean constants and these operators: +, -, *, DIV, MOD, =, <>, <=, >=, <, >, AND, OR, NOT.

An expression can also be of the form UNDEFINED *identifier*. This expression evaluates to TRUE if the identifier is undefined and FALSE if the identifier is defined.

To find out which compiler options are set, use the OPTION (*optionName*) function. The argument can be one of MC68881, MC68020, DEBUG, RANGE, OVERFLOW, or NAMES.

The { \$SETC } directive lets you set values to **compile-time variables**. These are not variables that you can use in your program. Instead, you use the variables to define conditions.

Note: You can use := as well as = in a { \$SETC } directive.

By combining compile-time variables with the { \$IFC }, { \$ELSEC }, and { \$ENDC } directives, you can tell THINK Pascal which parts of your program you want it to compile and which parts you want it to ignore in specific cases.

Here's an example. Suppose that you were writing an application that had a special menu that you wanted on only during development. You might write this:

```

program Toast;
{$SETC development = TRUE}
const
    FileMenuID = 1;
    ...
    {$IFC development}
        DebugMenuID = 7;
    {$ENDC}
    ...

procedure StartMenus;
    var
        h : MenuHandle;
begin
    ...
    {$IFC development}
        h := GetMenu(DebugMenuID);
        InsertMenu(h, 0);
    {$ENDC}
end;
    ...

```

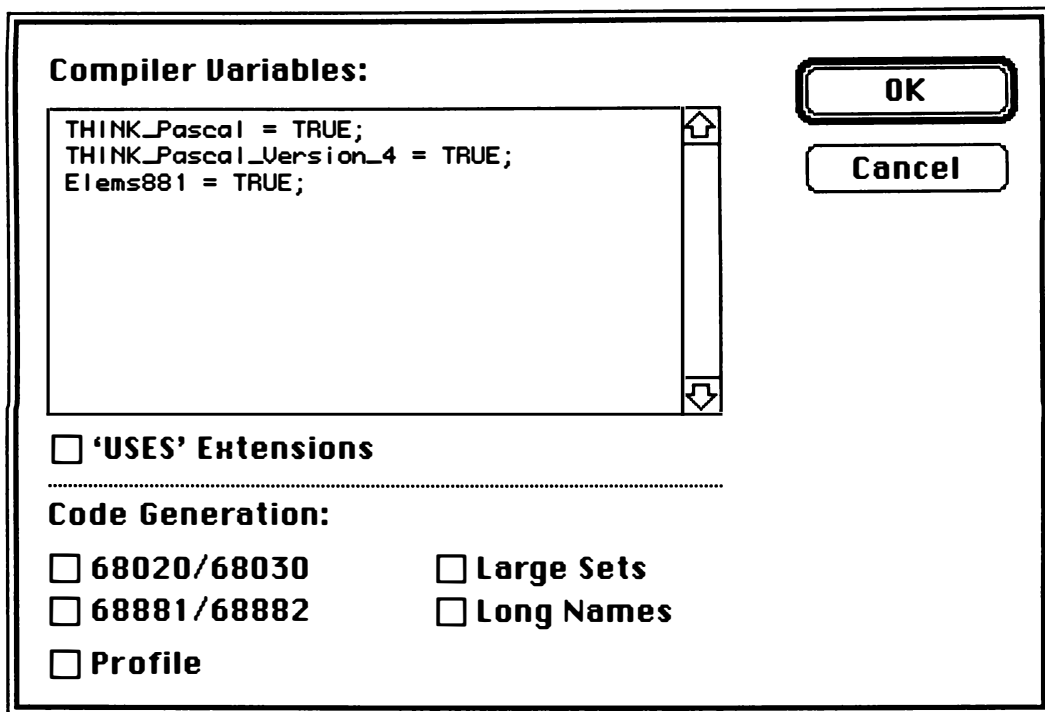
Note: In an actual case, you would probably set the value of the compile-time variable `development` with the **Compile Options...** command described in the next section.

You can use conditional compilation directives anywhere in your source file, even in constant and type declarations. If you use a `{ $SETC }` directive in a unit, you'll be able to use it in any file that **uses** the unit.

Using the Compile Options... Command

The **Compile Options...** command in the **Project** menu lets you predefine symbols that you can use for conditional compilation as well as set up some other code generation options.

When you choose **Compile Options...** from the **Project** menu, you'll see this dialog box:



Compiler variables

You can define any compile-time variable in the text-edit box in the left side of the dialog just as if you had typed:

```
{ $SETC identifier = value }
```

THINK Pascal predefines three compile-time variables for you, `THINK_Pascal`, `THINK_Pascal_Version_4` and `Elms881`. Use the `THINK_PASCAL` compile-time variable when you need to know if your program is being compiled by THINK Pascal. Use the `THINK_Pascal_Version_4` when you need to know whether your program is being compiled by the latest version of THINK Pascal.

The `Elms881` compile-time variable is used by the `SANE.p` interface file to indicate that it should use the faster (but less accurate) transcendental functions built into the MC68881 and MC68882 floating point units if the 68881/68882 option is on. If you set the `Elms881` compile-time variable to `FALSE`, THINK Pascal will use the slower, but more accurate, software versions of the transcendental functions.

USES Extensions

If you turn on the "USES Extensions" option, THINK Pascal lets you use these features:

- **Propagated `uses`.** If your unit uses other units, any unit that uses your unit also uses those units automatically.
- **Implementation `uses`.** You can put a uses clause in a unit's implementation section.

For more information on these features, see "The uses clause," in Chapter 10, "Units and Libraries."

68020/68030 option

When the 68020/68030 option is on, THINK Pascal generates code that uses the MC68020 or MC68030 CPU found in some Macintosh models (the Macintosh SE/30, II, IIfx, IIfx, and IIfx) and accelerator boards. If this option is on, your code will run only machines that have a MC68020 or MC68030. Specifically, THINK Pascal uses MC68020 and MC68030 instructions for 32 bit multiplies and divides and the RTD instruction to strip arguments off the stack.

Note: THINK Pascal doesn't check to see whether the machine you're writing your program on or whether the machine your application runs on has a MC68020 or MC68030. Use Gestalt Manager check the features of the Macintosh you're program is running on.

68881/68882 option

The 68881/68882 option controls whether THINK Pascal generates code that directly calls the MC68881 or MC68882 floating point unit (FPU) for floating point operations. If this option is on, your code will run only on machines that have a FPU.

Note: THINK Pascal doesn't check to see whether the machine you're writing your program on or whether the machine your application runs on has a FPU. If you aren't sure that your program will always run on a Macintosh with a FPU, turn off automatic initialization with the `{SI-}` directive and use the Gestalt Manager to check the features of the Macintosh you're program is running on.

Using the 68881/68882 option and the `ElEmS881` compile-time variable, described above, you can choose among three ways of handling floating-point operations. Choose an option depending on which of these are important to you: speed, accuracy, and portability.

To make your program...	Turn 68881/68882...	and set <code>ElEmS881</code> to...
Portable	Off	true or false
Fastest	On	true
Fast and Accurate	On	false

- **Portable.** Use this option for most programs, unless your program does a lot of intensive calculation and you can afford to have it run only on machines with a FPU.
 - Your program will run on all Macintosh models
 - Include `SANELib.lib` in your project.
 - All floating-point operations use the SANE library. If a FPU is present, the SANE library will use it.
 - All functions comply with the SANE accuracy standards.
 - All variables declared extended are 80 bits long.
- **Fastest.** Use this option for programs that do a lot of intensive calculation and that need to run only on machines with a FPU.
 - Your program will run only on Macintosh models with a FPU.
 - You don't need to include a special library.
 - All floating-point operations call the FPU directly.
 - Most functions comply with the SANE accuracy standards. The transcendental functions (e.g., `arctan`, `exp`, `ln`) comply with the IEEE accuracy standards.
 - All variables declared extended are 96 bits long.
- **Fast and Accurate.** Use this option for programs that do a lot of intensive calculation and need to be as accurate as possible.
 - Your program will run only on Macintoshes with a FPU.
 - Include `SANELib881.lib` in your project.
 - Most floating-point operations call the FPU directly. The transcendental functions (e.g., `arctan`, `exp`, `ln`) call the more accurate and slower routines in the SANE library.
 - All functions comply with the SANE accuracy standards.
 - All variables declared extended are 96 bits long.

Note: The FPU's built-in routines for the transcendental functions comply with IEEE accuracy standards, so they should be adequate in most cases. The library routines that comply with SANE's more stringent standards can run over 100 times slower. For more information on SANE, see *Apple Numerics Manual, Second Edition* (Addison-Wesley).

If your program doesn't use all the math functions available in THINK Pascal, you may not need to include the library listed for your option. This table lists exactly which library you need, depending on the option you choose and the functions you use. If the space for a library contains "n/a," that function is not available for that choice. If the space for a library contains "built-in," that version of the function is built into THINK Pascal, and you don't need to include a library to use it.

To use...	Include the library for the option you chose...		
	Portable	Fastest	Fast and Accurate
<code>arccos</code>	n/a	built-in	built-in
<code>arcsin</code>	n/a	built-in	built-in
<code>arctan</code>	<code>Runtime.lib</code>	built-in	<code>SANELib881.lib</code>
<code>arctanh</code>	n/a	built-in	built-in
<code>cos</code>	<code>Runtime.lib</code>	built-in	<code>SANELib881.lib</code>

To use...	Include the library for the option you chose...		
	Portable	Fastest	Fast and Accurate
cosh	n/a	built-in	built-in
exp	Runtime.lib	built-in	SANELib881.lib
expl	SANELib.lib	built-in	SANELib881.lib
exp2	SANELib.lib	built-in	SANELib881.lib
expl0	n/a	built-in	built-in
ln	Runtime.lib	built-in	SANELib881.lib
ln1	SANELib.lib	built-in	SANELib881.lib
log2	SANELib.lib	built-in	SANELib881.lib
log10	n/a	built-in	built-in
round	Runtime.lib	Runtime.lib	Runtime.lib
sin	Runtime.lib	built-in	SANELib881.lib
sinh	n/a	built-in	built-in
sqrt	Runtime.lib	built-in	built-in
tan	SANELib.lib	built-in	SANELib881.lib
tanh	n/a	built-in	built-in
trunc	Runtime.lib	Runtime.lib	Runtime.lib

Large sets option

THINK Pascal lets you specify whether sets of integer include all integers (-32768..32767) or just the range 0..255. Most of the time, you'll use the smaller set range. The larger set range takes up considerably more space in your program and takes much longer to access than the smaller range. Sets of the range 0..255 can take up at most 32 bytes. Sets of the range -32768..32767 can take up to 8192 bytes.

Long names option

When the Names option is on, THINK Pascal embeds subroutine names into the code right after the end of the procedure or function. Debuggers such as LightsBug, Macsbug, and TMON use these names. In the **Compile Options...** dialog, you choose how THINK Pascal stores them:

If this option is...	Names can be
Off	Only eight characters long
On	Any length

Note: You can turn on the names option in the project window or with the names directive { \$N+ }. See "Using Compiler Directives" above for more information.

Profile option

When the Profile option, THINK Pascal profiles your code. It collects statistics about your program, including the time spent in each routine. For more information, see Chapter 19, "The Profiler."

THINK PascalTM

P A R T F O U R

Reference

- 16 THINK Pascal Menus
- 17 Language Reference

THINK Pascal Menus

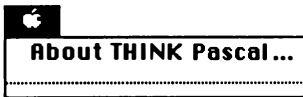
16

Introduction

This chapter describes each of the THINK Pascal menu commands. It is organized by menu, from left to right along the menu bar. Within each menu, commands are described in the order in which they appear.

The Apple Menu

The **Apple** menu gives you information on THINK Pascal and contains your desk accessories (or the items in your Apple menu folder under System 7.0).



The Apple menu

About THINK Pascal...

This command tells you what version of THINK Pascal you're using. Click the mouse once to see the credits for THINK Pascal and click the mouse again to end the display.

The File Menu

Use the **File** menu commands to work with files that you open and edit with the THINK Pascal editor. This menu also has commands that let you launch other applications and that let you quit THINK Pascal. When you hold the Option key down, the **File** menu has commands to close and save all the open files, and to print all the files in the open project.

File	
New	⌘N
Open...	⌘O
Close	⌘W
<hr/>	
Save	⌘S
Save As...	
Save a Copy As...	
Revert	
<hr/>	
Page Setup...	
Print...	⌘P
<hr/>	
Delete...	
<hr/>	
Transfer...	
Quit	⌘Q

The File menu

File	
New	⌘N
Open...	⌘O
Close All	⌘W
<hr/>	
Save All	⌘S
Save As...	
Save a Copy As...	
Revert	
<hr/>	
Page Setup...	
Print All Files	⌘P
<hr/>	
Delete...	
<hr/>	
Transfer...	
Quit	⌘Q

The Option File menu

New

This command opens a new Untitled Pascal edit window. Use the **Save As...** command to save new files.

Open...

This command lets you open an existing Pascal file in an edit window. You can have up to 16 files open at once. You can also use this command to open Instant and Observe windows that you've saved.

Close

Close All

The **Close** command lets you close the active window. Clicking in the window's close box does the same thing. If you try to close an edit window, and the file has been modified since it was last saved, THINK Pascal asks you if you want to save the changes, discard them, or cancel the **Close** command.

Note: When you hold down the Command key and click in an editing window's close box, THINK Pascal hides the window without closing the file.

The **Close All** command lets you close all the open files. To see this command, hold down the Option key and choose the **File** menu.

Note: When you hold down the Option key and click in an editing window's close box, THINK Pascal closes all the editing windows. When you hold down the Command and Option keys and click in an editing window's close box, THINK Pascal hides all the editing windows without closing the files.

Neither of these commands close the project window. Use the **Close Project** command in the **Project** menu to do that.

Save Save All

The **Save** command saves the file in the active edit window to disk. This command is dimmed if the current edit window is untitled or hasn't been changed.

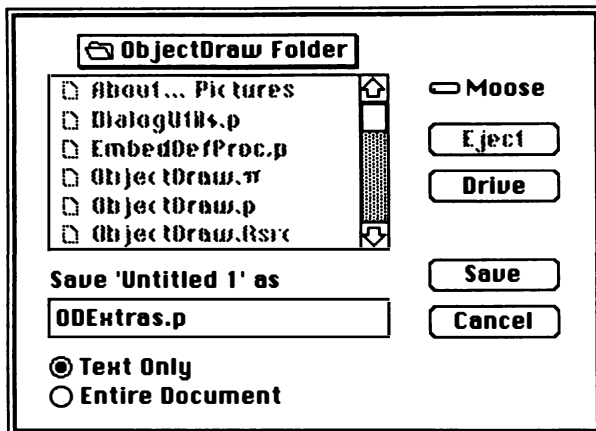
The **Save All** command saves the files in all the open windows to disk. To see this command, hold down the Option key and choose the **File** menu.

Save As...

This command lets you save the current file under another name. If you have made changes in the current session, THINK Pascal saves them under the new name. The original file remains unchanged, and as you continue editing, you are editing the new file. This feature is useful for switching to a new version of a file, leaving the old file as a backup.

The **Save As...** command tries to preserve the tie between the file you're editing and its entry in the project window. If the file is in the project window, THINK Pascal changes the name of the file in the project window to the new name. This command won't let you overwrite a file that's already in the project window.

When you choose this command, you'll see a standard save dialog box with two additional radio buttons. When you choose Text Only, THINK Pascal saves the file as a text file that you can open with any text editor or word processor. When you choose Entire Document, THINK Pascal saves the file in a special format that loads faster and preserves any Stop signs you've set in the file.



You can use **Save As...** to save the Instant and Observe windows to a file. When either of those windows is the active window, you can use the **Save As...** command to save them.

Save a Copy As...

Unlike **Save As...**, this command does not affect the status of the file currently being edited; it simply snapshots it to another file. This is a good way to make backups without finding yourself editing the backup. **Save a Copy As...** doesn't let you overwrite a file that's already in the project window.

Revert

This command restores the last-saved version of the current file, discarding any edits made since the last **Save** or **Save As...**

Page Setup...

This command displays the standard Page Setup dialog that lets you specify the size of the paper you're printing on, and whether the file should be printed upright on the page (tall orientation) or sideways (wide orientation). See your Macintosh owner's manual for details.

Print...

Print All Files

The **Print...** command lets you print the current file. The standard Print dialog box lets you set the page range among other options. When you press the OK button, your file begins to print. Each page of the file has a header showing the name of the file and the last modification date. To cancel printing, press Command-Period.

Note: If the active window is the project window, the **Print...** command prints the project window.

The **Print All Files** command lets you print the all the files in your project. You'll see a dialog listing your files as it prints them. To see this command, hold down the Option key and choose the **File** menu.

Note: The **Print All Files** command doesn't show you the Print dialog, but uses the default settings for its options.

Delete...

This command lets you delete a file from your disk without leaving THINK Pascal.

Transfer...

This command lets you launch another application without first returning to the Finder. When you're running under MultiFinder, this command launches applications without quitting THINK Pascal. When you're running under the Finder, holding down the shift key and selecting **Transfer...** launches an application and returns to THINK Pascal when the application exits.

Quit

This command quits THINK Pascal and returns to the Finder.

If you choose **Quit** while a project is open, it is automatically closed. If any editing windows have been modified, THINK Pascal asks if you want to save or discard your edits, or cancel the **Quit** command.

The Edit Menu

The **Edit** menu has the standard Macintosh editing commands (**Cut**, **Copy**, **Paste**) as well as commands that let you specify how THINK Pascal should format your Pascal source files.

Edit	
Undo	⌘Z
Cut	⌘H
Copy	⌘C
Paste	⌘V
Clear	
Select All	
Show Clipboard	
Source Options...	
✓Auto-Reformat	
Projector Aware	

The Edit menu

Undo

The Undo command reverses the last edit operation. The actual name of this command changes to let you know exactly which operation you'll be undoing. After a paste, for instance, the name of this command changes to **Undo Paste**. Once you've undone something, the name of this command changes to **Redo**.

If there isn't anything to undo, this command is dimmed. If the operation to undo doesn't belong to the frontmost window, the name of the command indicates that there is something to do, but the command is dimmed.

You can't undo a **Replace All**, or a **Revert**.

Cut

This command removes selected text and places it in the Clipboard. It replaces the current contents of the Clipboard (if there are any). Use the Paste command to insert text from the Clipboard into your file at the insertion point.

Copy

This command copies the selected text and places it in the Clipboard. The copy can be pasted somewhere else using the **Paste** command.

Paste

This command copies the contents of the Clipboard into the file being edited at the insertion point. If text is currently selected, it is replaced.

Clear

This command clears the selected text. The selection is not placed on the Clipboard. The Clear key on your keyboard has the same effect as the Clear command.

Select All

This command selects all the text in the current edit window.

Show Clipboard

This command shows you the contents of the Clipboard .

Source Options...

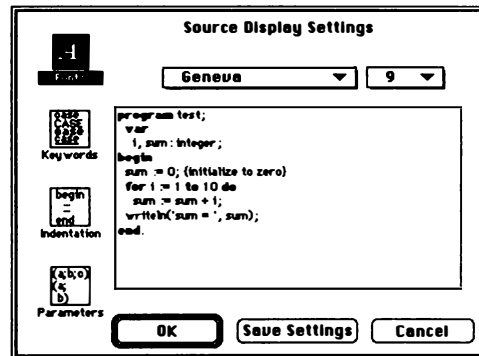
The **Source Options...** command lets you change the way the THINK Pascal editor pretty-prints your source code. You can also change the tab spacing, the indentation spacing, and the font that the editor uses to display your file.

The four icons along the left let you choose different aspects of the formatting you can change with this command. When you click on the OK button, any changes you made in any section of this dialog applies to your project. Clicking on the Save Settings button saves the settings as the THINK Pascal defaults.

FONTS

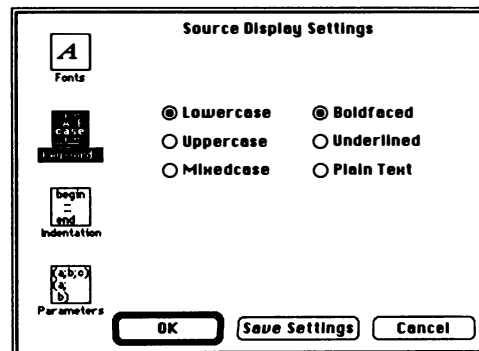
Lets you choose the font used to display your source code.

The pop-up menus lets you choose the font and size that THINK Pascal uses to display your source file.



KEYWORDS

Lets you choose how THINK Pascal displays Pascal reserved words like **begin** and **end**.



The case of the keywords can be one of these:

Lowercase	<code>begin...end</code> , for example.
Uppercase	<code>BEGIN...END</code> , for example.
Mixed Case	<code>Begin...End</code> , for example.

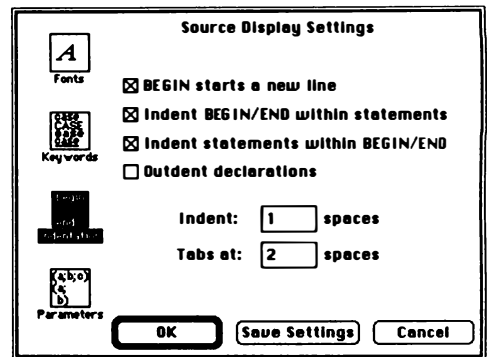
And the style of the keywords can be one of these:

Boldfaced	<code>begin...end</code> , for example.
Underlined	<u><code>begin...end</code></u> , for example.
Plain Text	<code>begin...end</code> , for example.

The most common keyword styles are lowercase bold, uppercase plain, and lowercase underline.

INDENTATION

Lets you choose the indentation style that THINK Pascal uses to pretty-print your program.



BEGIN starts a new line The reserved word **begin** appears at the beginning of a new line. When this option is off, **begin** appears at the end of a line. This is what your code looks like when you don't check this option:

```
for i := 1 to 10 do begin
...
end;
```

Indent BEGIN/END within statements If this option is checked, **begin/end** pairs are indented like this:

```
for i := 1 to 10 do
  begin
    ...
  end;
```

If this option is not checked, they're indented like this:

```
for i := 1 to 10 do
begin
...
end;
```

**Indent
statements
within
BEGIN/END**

This option controls whether statements within a **begin/end** pair are indented or lined up with the **begin/end** pair. Here's what code looks like when this option is checked:

```
begin
  writeln(i);
end;
```

Here's what code looks like when the option isn't checked:

```
begin
writeln(i);
end;
```

**Outdent
declarations**

This option controls whether the **const**, **type**, and **var** declarations in program, procedure, or function declarations should be indented or lined up with the **begin/end** pair. This is what outdented declarations look like:

```
program Outdent;

procedure Delicate;
  var
    aVar : aType;
  begin
  end;

begin
end.
```

And this is what code looks like with outdented declarations off:

```
program NoOutdent;

  procedure Delicate;
    var
      aVar : aType;
    begin
    end;

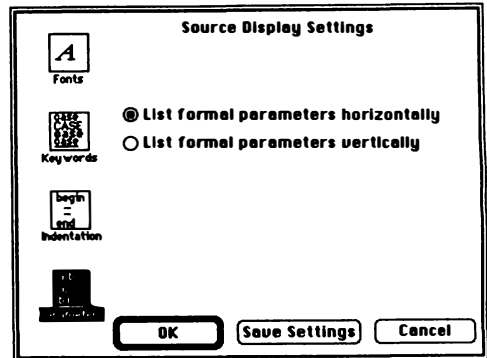
  begin
  end.
```

Indent: This is how many spaces added to the front of a line each time it's indented.

Tabs at: This is how many spaces are inserted when you press the Tab key.

PARAMETERS

Lets you choose how THINK Pascal displays parameter lists.



List formal parameters horizontally

This is how a parameter list looks like with this option on:

```
procedure foo (a: integer; b: integer; c:integer);
```

List formal parameters vertically

This how a parameter list looks like with this option on:

```
procedure foo (a: integer;
               b: integer;
               c: integer);
```

Auto-Reformat

This command lets you specify when THINK Pascal pretty-prints your program. This option is initially checked, and THINK Pascal reformats your program every time you press the Return key or the semicolon (;) key. If you uncheck this option, THINK Pascal pretty-prints your program only when you press the Enter key or when you move the insertion point to another line.

Projector Aware

This option lets you work on a large project with people who use MPW Projector. When you select this option, THINK Pascal displays an icon in the lower-left-hand corner of every editing window that tells you whether Projector marked the file as read-only. If the file is not read-only, its window has a pencil icon. If the file is read-only, its window has a crossed-out pencil icon. For more information, see "Using MPW Projector with THINK Pascal" in Chapter 6.

The Search Menu

The commands in the **Search** menu let you find and replace strings in your source files. THINK Pascal also lets you search all the files in your project for a string. When you hold down the Option key, you can use a command that lets you search through your project's files quicker.

Search	
Find...	⌘F
Find Again	⌘A
Find In Next File	⌘T
Enter Selection	⌘E
<hr/>	
Replace	⌘R
Replace and Find Again	⌘D
Replace All	
<hr/>	
Show Selection	
Show Error	

The Search menu

Search	
Find...	⌘F
Find Again	⌘A
Find in All Files	⌘T
Enter Selection	⌘E
<hr/>	
Replace	⌘R
Replace and Find Again	⌘D
Replace All	
<hr/>	
Show Selection	
Show Error	

The Option Search menu

Find...

This command lets you specify the string to search for. If the string is found, THINK Pascal highlights it. If it's not found, you'll hear a beep.

At the start of an editing session, only the **Find...** command is active. The dialog box that appears when you choose this command lets you specify the search string as well as the replacement string:

The two check boxes on the left let you specify how the editor looks for your string.

Whole Words

If you check this option, the search matches your string only with whole words. If you uncheck this option, the search matches even if your string is embedded in another string. For example, if the **Whole Words** option is off, the search string `rect` matches `rect`, `rectangle`, and `myrect`. If the **Whole Words** option is on, the search string `rect` matches only `rect`.

Match Case

If you uncheck this option, the search treats uppercase and lowercase letters as if they were the same. If you check this option, the search matches only if the string matches exactly. For example, if the **Match Case** box is off, the search string `rect` matches `rect`, `Rect`, and `RECT`. If the **Match Case** box is on, the search string `rect` matches only `rect`.

Note: Turning on the **Whole Words** option speeds up your search substantially .

The **Multi-File Search** check box lets you look for the search string in all the source files in your project. See Chapter 6 and the description for the **Find in Next File** command for more information about searching through more than one file.

In addition to the **Find** button ("Go ahead with the search") and the **Cancel** button ("Pretend I never invoked this command"), there is a **Don't Find** button. Clicking on this button sets up the search and replacement strings without actually doing the search. The **Don't Find** button is useful for setting things up for a **Replace All...** command.

Find Again

This command searches for the next occurrence of a previously specified string.

Find in Next File Find in All Files

These commands let you search for a string through more than one file. To use these commands, you must check the **Multi-File Search** check box in the **Find...** dialog.

The **Find in Next File** command looks for the string specified in the **Find...** dialog box through each of the files in your project. If it finds the strings, THINK Pascal opens an edit window containing the file, and selects the search string. At this point, you can go on and make any edits you choose. If you want to search further in the current file, you can use the **Find Again**, **Replace**, and **Replace All** commands, which work within the current file. When you're ready to go on with the multi-file search, use the **Find in Next File** command.

Using the **Find in All Files** command is like selecting the **Find in Next File** command repeatedly. To see **Find in All Files**, hold down the Option key as you select the **Search** menu. Like **Find in Next File**, **Find in All Files** looks for your search string through each file in your project. However, when **Find in All Files** finds your search string, it opens an edit window containing the file and continues searching in the next file in your project. **Find in All Files** stops only after it's searched each file in your project.

Multi-file search is useful when you are writing a program, and decide to modify a routine that is used in several files. You can open each of the files containing the search string, so you can switch back and forth between the various edit windows as necessary. (This feature is also helpful when you are tracking down link errors due to undefined or multiply defined symbols.)

Enter Selection

This command sets the search string to the current selection. You can then use **Find Again** to begin searching, or **Find...** to set search options. This command clears Multi-File Search if it was set.

Replace

This command replaces the current selection with a replacement string. If you haven't provided a replacement string, this command is dimmed.

Replace and Find Again

This command replaces the current selection with the replacement string, then finds the next instance of the search string, but does not replace it. Use this command to step through a series of replacements. After each replacement, you see the next instance of the search string, so you can decide whether you want to replace it. If you want to replace the string, use the **Replace** or **Replace and Find Again** commands. If not, use the **Find Again** command to find the next occurrence of the string.

If you haven't provided a replacement string, this command is dimmed.

Replace All

This command replaces every instance of the search string from the current cursor position to the end of the file. Use this command when you don't want to give your approval for every replacement.

Note: Be careful with this command. You cannot undo it with the **Undo** command.

Show Selection

This command lets you get back to the insertion point or the beginning of the current selection if you've scrolled away from it.

Show Error

This command lets you see where THINK Pascal has detected an error in your program if you've scrolled away from it.

The Project Menu

The commands in the **Project** menu work with the current project. You can open and create projects, set the project type, make sure all the files in the project are compiled and loaded. This menu also contains the commands you use when you're ready to build a file containing your application, desk accessory, device driver, or code resource. When you hold down the Option key, this menu lets you add several files at once.

Project	
New Project...	
Open Project...	⌘O
Close Project	
Add "Utils.p"	
Add File...	
Remove	
Build Library...	
Build Application...	
Remove Objects	
Set Project Type...	
Compile Options...	
View Options...	
Get Info...	

The Project menu

Project	
New Project...	
Open Project...	⌘O
Close Project	
Add "Utils.p"	
Add Files...	
Remove	
Build Library...	
Build Application...	
Remove Objects	
Set Project Type...	
Compile Options...	
View Options...	
Get Info...	

The Option Project Menu

New Project...

This command creates a new project document and opens an empty project window. The project window includes entries for the default libraries `Runtime.lib` and `Interface.lib`. You can then add files to the project with the **Add File...** or **Add Window** commands. Only one project can be open at a time.

Open Project...

This command opens an existing project.

Close Project

This command closes the current project. If a project still has open files that haven't been saved, THINK Pascal asks you if you want to save the files before closing the project.

Note: When you click in the project window's close box, THINK Pascal hides the project window without closing the project. To close the project, use this command or hold down the Command key when you click in the project window's close box.

When you close a project, THINK Pascal lets you open another project. If you don't want to open another project, click on the Cancel button.

Add "Utils.p"

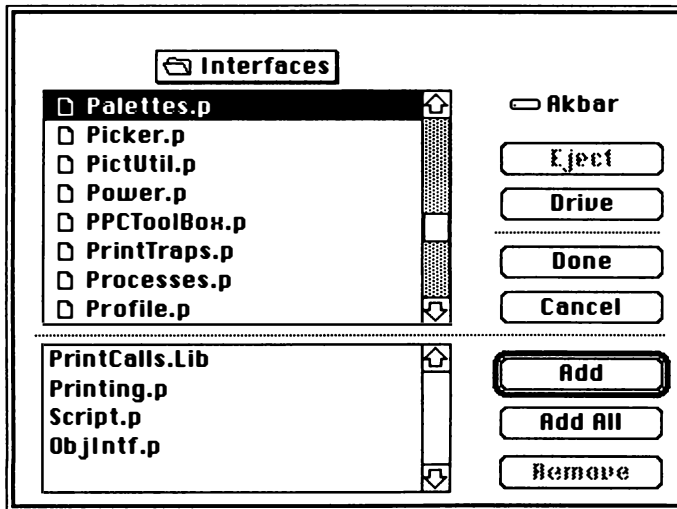
This command adds the current THINK Pascal edit window to the project. If the current edit window is an Untitled window, THINK Pascal asks you to save the file first. The command name contains the name of the current edit window.

Add File...

Add Files...

The **Add File...** command adds one file to your project. It displays a standard file dialog. Select the file you want to add, and click on the Add... button. After THINK Pascal adds the file to the project, it lets you choose another file to add. When you're done adding files, click on the Done button.

The **Add Files...** command lets you add several files to your project at once. To see **Add Files...**, hold down the Option key as you select the **Project** menu. You'll see a dialog box like this:



The top list displays the contents of the folder you're in. The bottom list displays the files that will be added to your project when you click Done. You add files to the bottom list with these commands:

- To add a file, select it and click Add, or double-click on the file name.
- To add all the files from the folder you're in, click Add All.
- To open a folder, select it and click Open, or double-click on the file name. (Add becomes Open when you click on a folder.)
- To remove a file from the bottom list, select it and click Remove.
- When you finish selecting files to add, click Done.
- If you change your mind and don't want to add any files, click Cancel.

The **Add File...** and **Add Files...** commands let you add add four kinds of files to your project:

- THINK Pascal source files
- THINK Pascal libraries
- THINK C libraries
- Macintosh Programmer's Workshop object files (.o files)

Remove

This command lets you remove the selected source file or library from the project. It's especially useful if you're project is corrupted. For more information, see "Recovering Corrupted Projects," in Chapter 7, "Working with Projects."

Bulld Library...

This command saves the current project as a library that you can add to other projects. A dialog box prompts you for the name of the library file. By convention, library files end in .Lib, but any valid file name is OK.

To learn how to create libraries, see Chapter 10.

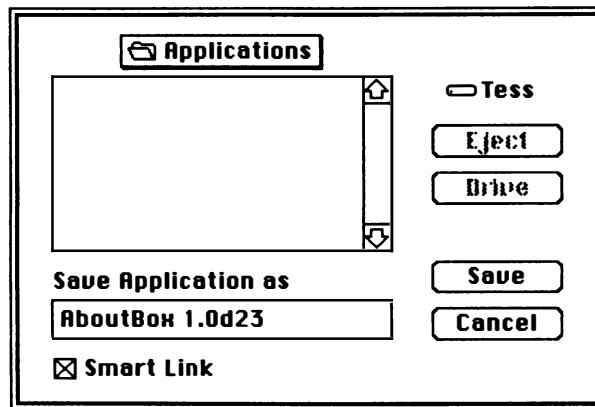
Bulld Application...

Bulld Desk Accessory...

Bulld Driver...

Bulld Code Resource

This command saves the current project as an application, desk accessory, device driver, or code resource. A dialog box lets you name the resulting file:



If the Smart Link option is checked, THINK Pascal uses only the referenced object code of the source and libraries to create the resulting file. It takes a little longer to put the application or resource together, but the resulting file is as small as possible. Uncheck this option if you're building frequently for testing.

When THINK Pascal builds your project with one of these commands, it merges the resource from the resource file you specified in the **Run Options...** dialog box into the resulting file.

Remove Objects

This command removes all the compiled code from a project. All source files must be recompiled, and all libraries must be reloaded.

Use the **Remove Objects** command when you need to make the project document as small as possible for archiving or for transmitting to someone else.

Set Project Type...

This command lets you set the project type. The default project type is Application, but you can change it to Desk Accessory, Driver, or Code Resource. All project types let you specify the file type and creator of the file created by one of the **Build...** commands. To learn the details of each project type, see Chapter 12.

Note: It's best to set the project type before you compile any of your source files. THINK Pascal needs to recompile all your source files if you change the project type once there is compiled code in the project.

APPLICATION

The Application presets the type to APPL and lets you fill in the creator.

Bundle Bit

If this option is on, THINK Pascal sets the bundle bit in the resulting file when you use one of the **Build...** commands. The bundle bit lets the Finder know that the file contains a BNDL resource, which is used to display the application's icons.

Far Code

If this option is off, THINK Pascal lets you write large applications: applications with a jump table as large as 256K. If this option is off, your jump table can be only 32K. The jump table includes an entry for every routine that your program calls from another segment or takes the address of. For more information, see "Building applications with large jump tables" in Chapter 12, "Building Projects."

Note: For more information on how the Finder uses the BNDL resource, see Chapter 7 of the *Resource Utilities Manual*.

DESK ACCESSORY DRIVER

The Desk Accessory and Device Driver dialogs are similar. For desk accessories, THINK Pascal presets the file type and creator so the resulting file is a Font/DA Mover file. There are other differences between desk accessories and device drivers. See Chapter 12 for details.

- Name** This field is the name of your desk accessory or device driver. By convention, desk accessory names begin with a null byte and device driver names begin with a period. THINK Pascal automatically inserts the null for you in desk accessories and adds the period to device driver names if you've left it out.
- Type** Desk accessories and device drivers are resources of type DRVr. You can change the type if you have some reason for doing so.
- ID** This field is the number of the DRVr resource. Desk accessories default to 12. The Font/DA Mover rennumbers it (and its owned resources) for you if there is an ID conflict with an installed desk accessory, so you shouldn't have to change the number. If the Multi-Segment option is on, the ID must be between 0 and 63.
- Attributes** This pop-up menu let you set the attributes for the DRVr resource. To learn about resource attributes see *Inside Macintosh I*, Chapter 5, "The Resource Manager."
- Multi-Segment** When this option is checked, your desk accessory can have up to 31 segments. If your desk accessory uses Object Pascal, this option must be on.

- Flags** This pop-up menu lets you set the `drvFlags` field of the driver's header. These flags let the operating system know what driver calls your driver responds to. The default for desk accessories is \$0400. For device drivers the default is \$4F00. For more information about driver headers, see *Inside Macintosh II*, Chapter 6, "The Device Manager."
- Delay** This field lets you set the `drvDelay` flag of the driver's header. The value in this field lets the operating system know how often it needs to call your driver for some periodic action. For this field to make sense, you must have the `dNeedTime` bit set in the `drvFlags` field.
- Mask** This pop-up menu lets you set the `drvEMask` field of the driver's header. The mask lets the system know which events your desk accessory responds to. (This field isn't used in device drivers.)

CODE RESOURCE

The Code Resource dialog lets you specify the type, name, id, and attributes of your code resource.

The Code Resource dialog box is shown with the following fields and options:

- File Information:**
 - Type:
 - Creator:
 - ☐ Bundle Bit
 - ☐ Far Code
- Resource Information:**
 - Name:
 - Type:
 - ID:
 - Attributes:
 - ☐ Multi-Segment
 - ☐ Custom Header
 - Segment Type:
- Driver Information:**
 - Flags:
 - Delay:
 - Mask:

Buttons: OK, Cancel

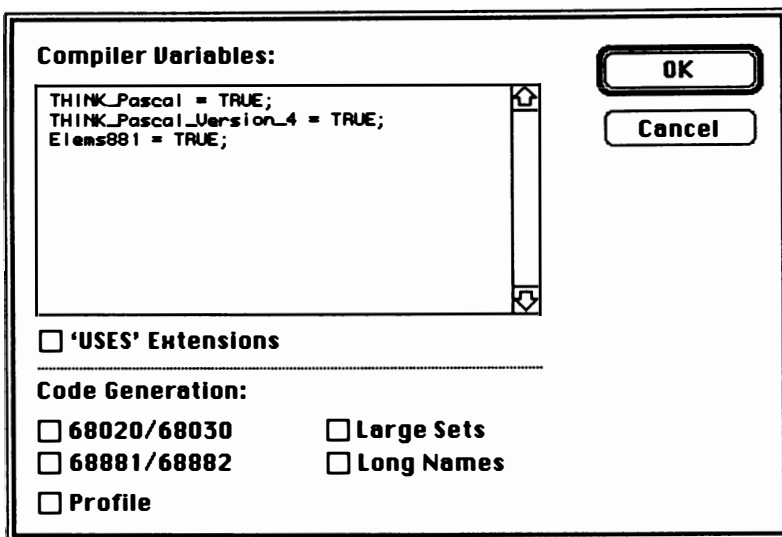
- Name** This field lets you name your code resource. For most code resources, the name is optional.
- Type** This field lets you specify the type of the code resource.
- ID** This field lets you specify the ID of the code resource.
- Attributes** This pop-up menu lets you specify the attributes of your code resource. To learn about resource attributes see *Inside Macintosh I*, Chapter 5, "The Resource Manager."
- Multi-Segment** When this option is checked, your code resource can have up to 31 segments. If your desk accessory uses Object Pascal, this option must be on. If the Multi-Segment option is on, the ID must be between 0 and 63.
- Segment Type** When the Multi-Segment option is checked, this field lets you specify the resource type of the owned segments.

Custom Header

When this option is on, THINK Pascal does not generate the standard resource header. Instead, the object code for the file that contains `main` is guaranteed to appear first in the resource. See Chapter 12 for details.

Compile Options...

This command lets you define compile-time variables for the project, and it lets you specify some code generation options. When you choose this command, you'll see this dialog box:



Compiler Variables

You can define any compile-time variables in this large edit box just as if you had used the `{ $SETC }` compiler directive.

'USES' Extensions

If this option is on, you can use these features:

- **Propagated *uses*.** If your unit uses other units, any unit that uses your unit also uses those units automatically.
- **Implementation *uses*.** You can put a *uses* clause in a unit's implementation section.

For more information, see "The *uses* clause," in Chapter 10, "Units and Libraries."

68020/68030

When this option is on, THINK Pascal generates code for the MC68020 and MC68030 CPUs found in some Macintosh models and accelerator boards.

Note: THINK Pascal doesn't check to see whether the machine you're writing your program on or whether the machine your application runs on has an MC68881 or MC68882 floating point unit. Use the Gestalt Manager, described in *Inside Macintosh VI*, Chapter 3, "Compatibility Guidelines," or `SysEnvi`rons,

described in *Inside Macintosh V*, Chapter 1, "Compatibility Guidelines." If you want to check for the machine's capabilities, you'll need to turn off automatic initializations.

68881/68882 When this option is on, THINK Pascal generates code for the MC68881 and MC68882 floating point units for all floating point operations. For more information on how you can use this option together with the Elems881 compiler variable, see "68881/882 option" in Chapter 15, "Compiler Directives."

Note: THINK Pascal doesn't check to see whether the machine you're writing your program on or whether the machine your application runs on has an MC68881 or MC68882 floating point unit. Use the Gestalt Manager, described in *Inside Macintosh VI*, Chapter 3, "Compatibility Guidelines," or SysEnviroms, described in *Inside Macintosh V*, Chapter 1, "Compatibility Guidelines." If you want to check for the machine's capabilities, you'll need to turn off automatic initializations.

Long Names When the Names option is on, THINK Pascal embeds subroutine names into the code right after the end of the procedure or function. Debuggers such as LightsBug, Macsbug, and TMON use these names. If the "Long Names" option is on, names can be any length. If it's off, THINK Pascal stores only the first eight characters of the name. For more information on the Names option, see Chapter 15, "Compiler Directives."

Large Sets THINK Pascal lets you specify whether sets of integer include all integers (-32768 . . 32767) or just the range 0 . . 255. Most of the time, you'll use the smaller set range. The larger set range takes up considerably more space in your program and takes much longer to access than the smaller range. Sets of the range 0 . . 255 can take up at most 32 bytes. Sets of the range -32768 . . 32767 can take up to 8192 bytes.

Profile When the Profile option, THINK Pascal profiles your code. It collects statistics about your program, including the time spent in each routine. For more information, see Chapter 19, "The Profiler."

For more information on this command, see "Using the Compile Options Command" in Chapter 15.

View Options...

This command lets you customize what appears in the project window. When you choose this command, you'll see this dialog box:

Options	File (by build order)	Size
D N V R	MemHacks	32767
D N V R	Extremely w-i-d-e...	10

☒ filename ☐ unit name
☒ options ☐ volume name
☒ code size ☐ date file saved

Geneva 9

OK Cancel

Options - Boxed indicates enabled

D - Debug. Allows stepping, stopping, stack checking, Observing.
N - Names. Insert Macsbug names into the code.
U - Integer arithmetic overflow checking.
R - Range checking.

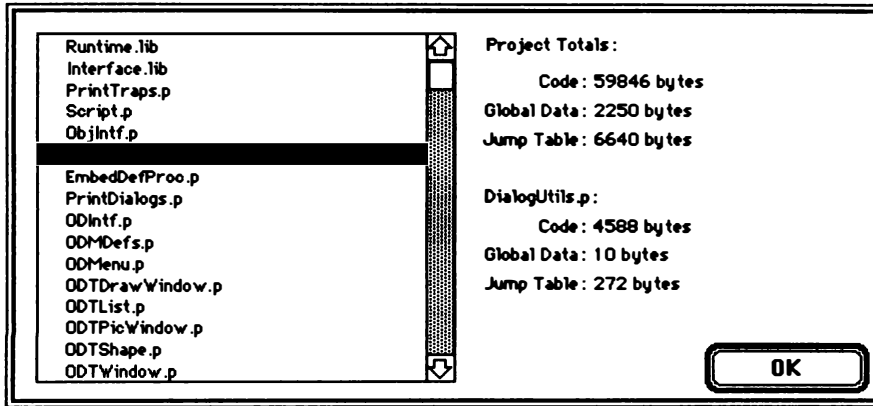
The two pop-up menus let you choose the font THINK Pascal uses to display the file names in the project window. The check boxes let you choose the information that appears for each file. The box at the top of the dialog box shows you what the project window would look like with the current settings.

Option	Meaning
filename	Displays the name of the file.
options	Displays the compiler options.
code size	Displays the size of the compiled code.
unit name	Displays the unit name if the file has been compiled.
volume name	Displays the volume or folder the file is in.
date file saved	Displays the last date and time you saved the file.

The order that you click the check boxes determines the order of the display.

Get Info...

This command shows how much code and data each file in your project produces. Choosing the command brings up this dialog:



The list on the left contains all the files in your project. To examine a file, click on it. To examine several files in a range, hold down the Shift key and select the range. To examine several scattered files, hold down the Command key and click on each file.

The right side displays the amount of code and data. On top, the "Project Totals" displays total amount of code and data in your project. Underneath is the amount of code and data that the selected file (or files) contributes to the total.

This table explains what the dialog displays:

Title	Meaning
Code	The amount of object code. This is same number in the Size column of the project window.
Global Data	The amount of global data.
Jump Table	The size of the jump table, which contains the addresses of functions and procedures in your application.

The Run Menu

The commands and options in the **Run** menu let you compile and run your program. When you hold down the Option key, this menu has automatic versions for some commands. When you hold down the Shift key, this menu lets you compile individual files.

Run	
Check Syntax	⌘K
Build	⌘B
Check Link	
Reset	
<hr/>	
Go	⌘G
Step Over	⌘J
Step Into	⌘I
Step Out	⌘U
<hr/>	
Auto Save	
✓Confirm Saves	
Don't Save	
<hr/>	
Run Options...	

The Run menu

Run	
Check Syntax	⌘K
Build	⌘B
Check Link	
Reset	
<hr/>	
Go-Go	⌘G
Step Over	⌘J
Step-Step	⌘I
Step Out	⌘U
<hr/>	
Auto Save	
✓Confirm Saves	
Don't Save	
<hr/>	
Run Options...	

The Option Run menu

Run	
Compile	⌘K
Build	⌘B
Check Link	
Reset	
<hr/>	
Go	⌘G
Step Over	⌘J
Step Into	⌘I
Step Out	⌘U
<hr/>	
Auto-Save	
✓Confirm Saves	
Don't Save	
<hr/>	
Run Options...	

The Shift Run menu

Check Syntax Compile

The **Check Syntax** command checks the Pascal syntax of the current edit window or of the Instant window.

The **Compile** command compiles the current edit window and updates the project document. To see the **Compile** command, hold down the Shift key as you select the **Run** menu.

Build

This command compiles all the files that have changed or that use a unit whose interface section has changed.

The execution commands — **Go**, **Step Into**, **Step Over**, **Go-Go**, and **Step-Step** — do an implicit **Build** before executing the program.

Check Link

This command links all the files in the project. If THINK Pascal finds an undefined symbol or a symbol defined more than once, it reports an error.

The execution commands — **Go**, **Step Into**, **Step Over**, **Go-Go**, and **Step-Step** — do an implicit **Check Link** before executing the program. If there are files that need to be recompiled, THINK Pascal asks you if you want to recompile them.

Reset

This command resets a paused program. The next execution command — **Go**, **Step Into**, **Step Over**, **Go-Go**, and **Step-Step** — starts the program from the beginning.

Go Go-Go

The **Go** command runs your program, pausing at the Stop Signs you placed in your code.

You can think of **Go-Go** as the automatic version of **Go**, choosing **Go** again and again. To see the **Go-Go** command, hold down the Option key and choose the **Run** menu.

The **Go-Go** command runs your program without stopping. It pauses at Stop Signs to move the execution finger and update the Observe and LightsBug windows.

Both commands build the project first, if necessary.

If you hold down the Shift key when you choose either command, THINK Pascal lets you examine your compiled code with a low-level debugger, such as TMON or Macsbug. It breaks into the low-level debugger at an RTS instruction just before the actual code for the statement.

Step Over

This command executes the next statement in your program. It then moves the execution finger, and updates the Observe and LightsBug windows. If the statement has a function or procedure call, this command executes the routine without moving the execution finger into it.

This command builds the project first, if necessary.

If you hold down the Shift key when you choose this command, THINK Pascal lets you examine your compiled code with a low-level debugger, such as TMON or Macsbug. It breaks into the low-level debugger at an RTS instruction just before the actual code for the statement.

Step Into Step-Step

The **Step Into** command executes the next statement in your program. It then moves the execution finger and updates the Observe and LightsBug windows. If the statement has a function or procedure call, this command moves the execution finger to the first line of the routine.

You can think of **Step-Step** as the automatic version of **Step Into**, choosing **Step Into** again and again. To see the **Step-Step** command, hold down the Option key and choose the **Run** menu.

The **Step-Step** command executes your program statement by statement. It pauses after each statement to move the execution finger and update the Observe and LightsBug windows. If a

statement has a function or procedure call, this command moves the execution finger into the routine.

Both commands build the project first, if necessary.

If you hold down the Shift key when you choose either command, THINK Pascal lets you examine your compiled code with a low-level debugger, such as TMON or Macsbug. It breaks into the low-level debugger at an RTS instruction just before the actual code for the statement.

Step Out

This command continues executing until it returns from the current routine. You'll find this command especially useful if you accidentally step into a routine with the **Step Into** command.

This command builds the project first, if necessary.

If you hold down the Shift key when you choose this command, THINK Pascal lets you examine your compiled code with a low-level debugger, such as TMON or Macsbug. It breaks into the low-level debugger at an RTS instruction just before the actual code for the statement.

Auto-Save Confirm Saves Don't Save

These three options determine what THINK Pascal does with files that you've changed but haven't saved when you choose one of the execution commands in the **Run** menu.

The three options are mutually exclusive. You can choose only one.

Auto-Save When this options is checked, THINK Pascal automatically saves your files before running your project.

If you hold down the Shift key and then choose Auto-Save, THINK Pascal saves all your unsaved files without changing the current option.

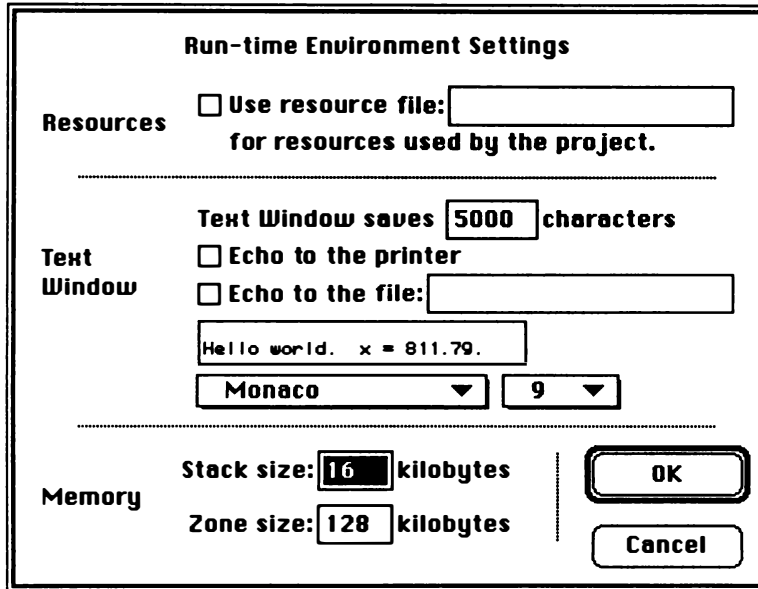
Confirm Saves When this option is checked, THINK Pascal asks if you want to save your unsaved files before running your project.

If you hold down the Shift key and then choose Confirm Saves, THINK Pascal asks you if you want to save each of your unsaved files.

Don't Save When this options is checked, THINK Pascal doesn't do anything with your unsaved files before running your project.

Run Options...

This command lets you set up certain run-time environment settings. When you choose this command you'll see this dialog box:



The dialog box is titled "Run-time Environment Settings". It is divided into three sections: Resources, Text Window, and Memory. The Resources section has a checkbox labeled "Use resource file:" followed by an empty text field and the text "for resources used by the project." below it. The Text Window section has a label "Text Window" on the left. It contains a text field "Text Window saves" with the value "5000" and the text "characters" to its right. Below this are two checkboxes: "Echo to the printer" and "Echo to the file:" followed by an empty text field. A sample text field shows "Hello world. x = 811.79.". Below the sample text are two pop-up menus: one showing "Monaco" and another showing "9". The Memory section has a label "Memory" on the left. It contains two text fields: "Stack size:" with the value "16" and "kilobytes" to its right, and "Zone size:" with the value "128" and "kilobytes" to its right. On the right side of the dialog are two buttons: "OK" and "Cancel".

Resources

When you check this box, THINK Pascal lets you choose the resource file your program uses. To learn more about using resource files with your THINK Pascal projects, see Chapter 8 and Chapter 12.

NOTE: THE RESOURCE FILE MUST BE IN THE SAME FOLDER AS THE PROJECT AND CANNOT BE AN ALIAS.

Text Window

This part of the dialog lets you configure the THINK Pascal Text window. THINK Pascal uses the Text window for all standard output, for instance write, writeln, read, etc..

You can specify how many characters THINK Pascal saves in the Text window. The default is 5000 characters. If you write more than that to the text window, the earlier text disappears.

The Echo to printer and Echo to a file check boxes let you send the standard output to the printer or to a file as well as to the Text window.

The pop-up menus let you choose the font and size of text in the Text window.

Memory

These options let you choose how much memory THINK Pascal allocates for your project's stack and heap zone. If your program is running out of memory, try making the zone size bigger. If your program allocates a lot of local variables, or if it uses very deep recursion, you might want to make the stack size bigger. In most cases, 16K is more than enough.

Note: When you build your application, THINK Pascal uses the stack size value you specify, unless you turn off initialization with the {SI-} directive. The zone size value applies only while you're running in the THINK Pascal environment.

The Debug Menu

The **Debug** menu commands and options let you control THINK Pascal's debugging features. When you hold down the Option key, you can remove all the Stop Signs in your project at once. When you hold down the Shift key, you can create new LightsBug windows and use your low-level debugger.

Debug	
LightsBug	⌘L
Instant	
Observe	

Show Finger	
Pull Stops	

Auto-Show Finger	
Stops In	
Break at A-Traps	

Use Second Screen	
Quietly Auto-Reset	

Monitor	⌘M

The Debug menu

Debug	
LightsBug	⌘L
Instant	
Observe	

Show Finger	
Pull All Stops	

Auto-Show Finger	
Stops In	
Break at A-Traps	

Use Second Screen	
Quietly Auto-Reset	

Monitor	⌘M

The Option Debug menu

Debug	
New LightsBug	⌘L
Instant	
Observe	

Show Finger	
Pull Stops	

Auto-Show Finger	
Stops In	
Break at A-Traps	

Use Second Screen	
Quietly Auto-Reset	

Use Monitor	⌘M

The Shift Debug menu

LightsBug New LightsBug

The **LightsBug** command opens a LightsBug debugging window. You can open up to four LightsBug windows. Choosing the **LightsBug** command (or its Command key equivalent) when LightsBug windows are open cycles through all the open LightsBug windows.

The **New LightsBug** command creates a new LightsBug window. To see the New LightsBug command, hold down the Shift key as you select the **Debug** menu.

Instant

This command opens the Instant window, which is used for executing statements when your program is paused.

Observe

This command opens the Observe window, which displays the values of expressions during program execution.

Show Finger

This command makes the window that contains the execution finger the active window. If necessary, it scrolls the window so you can see the execution finger. If the window that would contain the execution finger is not open, the project window becomes the active window, and the execution finger points at the file name.

Pull Stops

Pull All Stops

The **Pull Stops** command removes all Stop Signs from the current edit window. The **Pull All Stops** command removes all Stop Signs from all the files in the project.

Auto-Show Finger

You can think of this command as an automatic version of **Show Finger**. When this option is checked, and THINK Pascal is executing a **Step Into**, **Step Over**, **Go-Go**, or **Step-Step** command, the window that contains the execution finger becomes the active window. If the file that would contain the execution finger isn't opened, the execution finger points to the file name in the project window.

Since this command changes the active window, it's a good idea to turn this option off when you're debugging the code that handles activate and update events in your application.

Stops In

This option lets you add Stops to your Pascal code. To add a Stop to your program, move the cursor to the left column of your program. The cursor changes to a Stop sign. When you click the mouse, a Stop sign appears to the left of the line of code.

To remove a Stop Sign, click on it.

When you choose the **Go** or **Step-Step** command, THINK Pascal stops execution right before the Stop sign.

Break at A-Traps

When this option is on, THINK Pascal stops execution before calling a Macintosh Toolbox routine. Use this option to make sure that you're passing the proper values to Toolbox routines.

Use Second Screen

If this option is on and if you have two screens, THINK Pascal places the Instant, Observe, and LightsBug windows on your second screen.

Quietly Auto-Reset

If you're running a program under THINK Pascal and you try to change your source files or the project, THINK Pascal displays a warning and asks if you want to cancel your change or reset the program. If this option is on, THINK Pascal won't warn you and automatically resets your program.

Monitor Use Monitor

The **Monitor** command drops you into the low-level debugger (Macsbug or TMON) if it's installed.

The **Use Monitor** command lets you choose which debugger you use when THINK Pascal comes to an exception. If the **Use Monitor** option is on, you use the low-level debugger, like TMON or Macsbug. If the **Use Monitor** option is off, you use LightsBug. A diamond appears next the the **Use Monitor** and **Monitor** commands when this option is on. To see the **Use Monitor** command, hold down the Shift key as you select the **Debug** menu.

The Windows Menu

The commands in the **Windows** menu help you work with the THINK Pascal windows.

Windows	
MyProject.p	⌘0
Arrange...	
✓Auto-Reopen	
✓Save Positions	
Class Browser	⌘H
Text	
Drawing	
File1.p	⌘1
File2.p	⌘2
File3.p	⌘3
Available	⌘4

The Windows menu

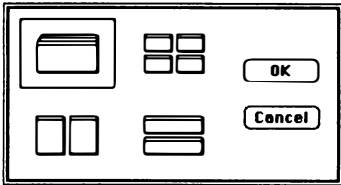
Project name

This command makes the project window the active window.

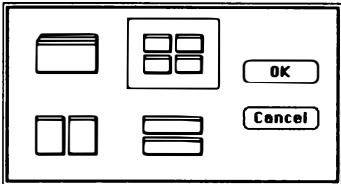
Arrange...

This command arranges the editing windows. The **Arrange...** dialog box lets you choose one of four ways. To choose an option, either click on its icon and click OK or just double-click on its icon. If you change your mind, click Cancel.

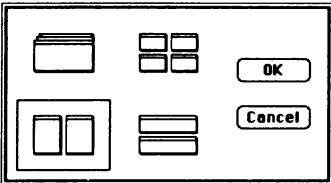
These are the four options:



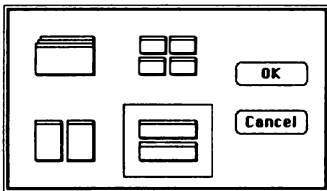
Overlapping. The first window is the about the length of the screen. The top of each subsequent window is placed just below the previous window's title bar. After five windows, the cycle starts over.



Tiled. All windows are completely visible. The screen is split into as many sections (approximately equal in size) as there are editing windows. Each window is placed in a section.



Side-by-side horizontally. One window is in the left half of the screen; the rest are in the right. Only two editing windows are visible.



Side-by-side vertically. One window is in the top half of the screen; the rest are in the bottom. Only two windows are visible

Auto-Reopen

If checked, opens a project's windows automatically whenever a project is opened. (This affects all windows: the editing windows, and the project, Observe, Instant, and LightsBug windows.)

Save Positions

If checked, saves the positions of the windows when the project is closed. Next time the project is opened, the windows open in the same place. (This affects all windows: the project window, the editing windows, and the Observe, Instant, and LightsBug windows.)

Class Browser

This displays the Class Browser window, which shows the hierarchy of your classes and helps you explore it. For more information, see the *Object-Oriented Programming Manual*, Chapter 6.

Text

This command opens the Text window which THINK Pascal uses for standard output like `write` and `writeln`.

Drawing

This command opens the Drawing window which THINK Pascal uses as the default drawing port for your program.

File Windows

Use these commands to activate one of the currently open editing windows. A diamond appears next to files which have been changed but haven't been saved. If you hid a window by clicking in its close box, choosing its name in this menu makes it visible.

Language Reference

17

Introduction

This chapter describes the Pascal language implemented by THINK Pascal. THINK Pascal is intended to be compatible with Macintosh Pascal, Apple's Macintosh Programmer's Workshop Pascal, and American National Standard Pascal. Naturally, it's impossible to be completely compatible with all three implementations. Appendixes B and C detail the differences between THINK Pascal and other implementations of Pascal.

Related Documents

Pascal User Manual and Report, Third Edition (ISO Pascal Standard) by Jensen and Wirth, revised by Mickel and Miner, Springer-Verlag, New York, 1985.

American National Standard Pascal Computer Programming Language, ANSI/ IEEE770X3.97-1983, IEEE/Wiley-Interscience, 1983.

Object Pascal Report by Larry Tesler. Apple Technical Report No. 1. Apple Computer 1985.

Inside Macintosh I-VI, Addison Wesley, 1985-1991.

Apple Numerics Manual, Second Edition (Addison-Wesley)

Definitions

This chapter uses these definitions for the terms error, undefined, and unspecified:

- | | |
|--------------------|--|
| Error | Any misuse of the Pascal language described in this chapter. Most of these errors are detected at compile time or at run time. Other errors are not detected; these are listed in Appendix B. The behavior of a program that contains undetected errors is unspecified. |
| Undefined | A value of a variable or of a function that is not meaningful. It is an error to use an undefined value. |
| Unspecified | When a situation arises in the execution of a program where several courses of action are possible, the specific course chosen is said to be unspecified. This means that the program should not depend on any specific course being chosen, as the result may be unpredictable. This leaves the implementation free to choose the course that is most convenient at the time. |

Notation and syntax diagrams

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

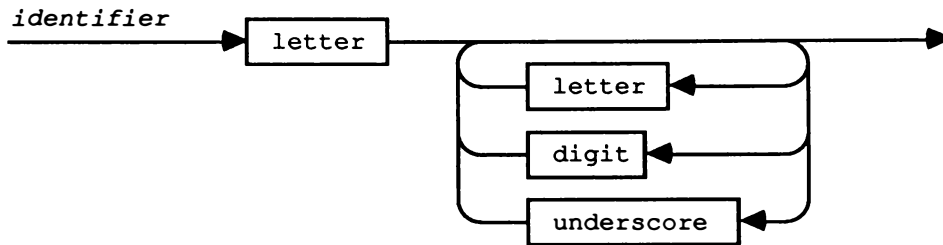
In this section, Pascal text is in typewriter text, and Pascal reserved words are in **bold typewriter text**. For example:

```
sqr(n div 16)
```

Sometimes the same word appears both in plain text and in typewriter text. For example, "The declaration of a Pascal procedure begins with the word **procedure**."

Technical terms appear in **bold face** when they're introduced.

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

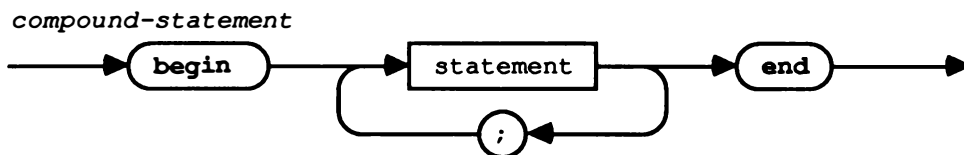


Start at the left and follow the arrows through the diagram. There are several paths you can take. Every path begins at the left and ends at the arrow-head on the right, and represents a valid way to construct an identifier. The boxes represent the elements that can be used to construct an identifier. The diagram represents the following rules:

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

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

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



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

So this diagram means that a compound-statement consists of the reserved word **begin**, followed by any number of statements separated by semicolons, followed by the reserved word **end**.

1.0 Tokens and Constants

Tokens are the smallest meaningful units of text in a Pascal program. Structurally, they correspond to the words and punctuation of an English sentence. The tokens of Pascal are classified into **special symbols**, **identifiers**, **numbers**, **labels**, and **character-strings**.

The text of a Pascal program consists of tokens and **separators**, where a separator is either a **blank** or a **comment**. Two adjacent tokens must be separated by one or more separators if each token is an identifier, number, or word-symbol.

Separators cannot be embedded within tokens except in character-strings.

1.1 Character set and special symbols

THINK Pascal uses the Macintosh character set. Letters, digits, hex-digits, and blanks are subsets of the character set:

- **Letters** are the characters A through Z and a through z.
- **Digits** are the Arabic numerals 0 through 9; the **hex-digits** are the Arabic numerals 0 through 9, the letters A through F, and the letters a through f.
- The **blanks** are the space character and the end-of-line character (CR).

Special-symbols and **word-symbols** (also called **reserved words**) are tokens that have one or more fixed meanings. The following single characters are special-symbols:

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

The following **character-pairs** are special-symbols:

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

Note: The special symbol "(." is an alternate representation for the special symbol "[". Both actually denote the same special symbol. If you type "(." in a THINK Pascal program, it will always be displayed as "[". The same is true for ".)" and "] ".

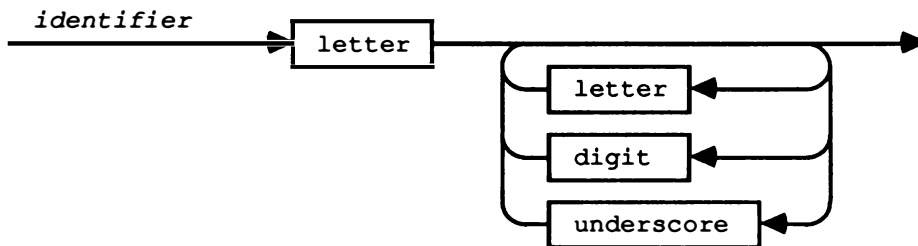
The following are the **word-symbols**:

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

Upper and lower case letters are equivalent in word-symbols.

1.2 Identifiers

Identifiers serve to denote constants, types, variables, procedures, functions, programs, and fields in records. Identifiers can be up to 255 characters long. All characters are significant. Upper and lower case letters are equivalent in identifiers. No identifier can have the same spelling as a word-symbol.



Examples of identifiers:

X Rome gcd SUM get_byte

1.3 Directives

Directives are identifiers that have special meanings in specific contexts. They can be used as identifiers in all other contexts. For example, the word `forward` is interpreted as a directive if it occurs immediately after a procedure-heading or function-heading, but in any other position it is interpreted as an identifier.

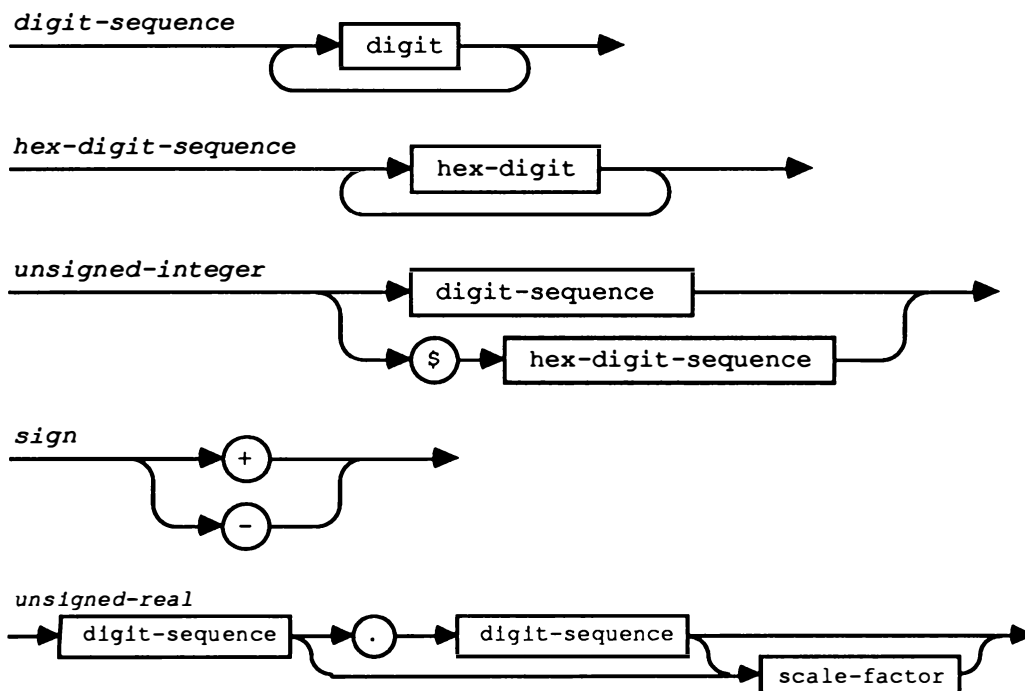
THINK Pascal recognizes these three directives:

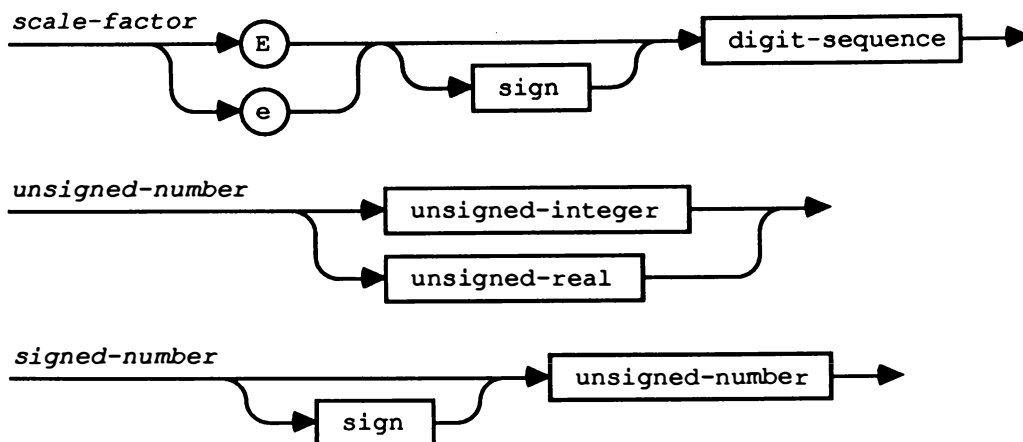
```
external
forward
override
```

Note: Don't confuse these directives with compiler directives.

1.4 Numbers

The usual decimal notation is used to represent numbers that are constants of the data types `integer`, `longint`, `real`, `double`, `extended`, and `computational` (see § 3.1). A hexadecimal integer constant uses the `$` character as a prefix (1-4 hex-digits for `integer`, 5-8 hex-digits for `longint`).





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

Examples of numbers:

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

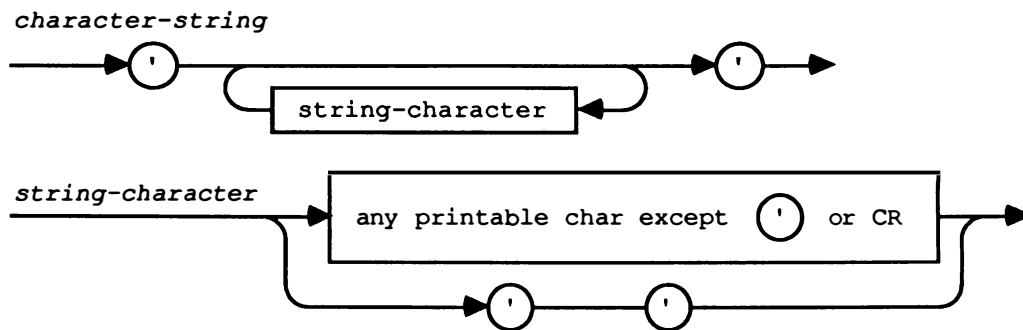
Note that 5E-3 means 5×10^{-3} , and 87.35e+8 means 87.35×10^8 .

1.5 Labels

A **label** is a digit-sequence whose value is in the range 0 through 9999. Leading zeros in a label are insignificant. The labels 1 and 0001 are equivalent.

1.6 Character-Strings

A **character-string** is a sequence of zero or more printing characters on the same line in a program and enclosed by single quotes. The maximum number of characters that can be in a character-string is 255. A character-string with nothing between the apostrophes is a **null-string** value (see §3.3). Two adjacent apostrophes in a character-string denotes a single apostrophe character.



A character-string represents a value of a **string-type**. As a string-type, a character-string is compatible not only with other string-types, but also **char-types** (see §3.1.1.1) and **packed-string-types** (§3.2.1).

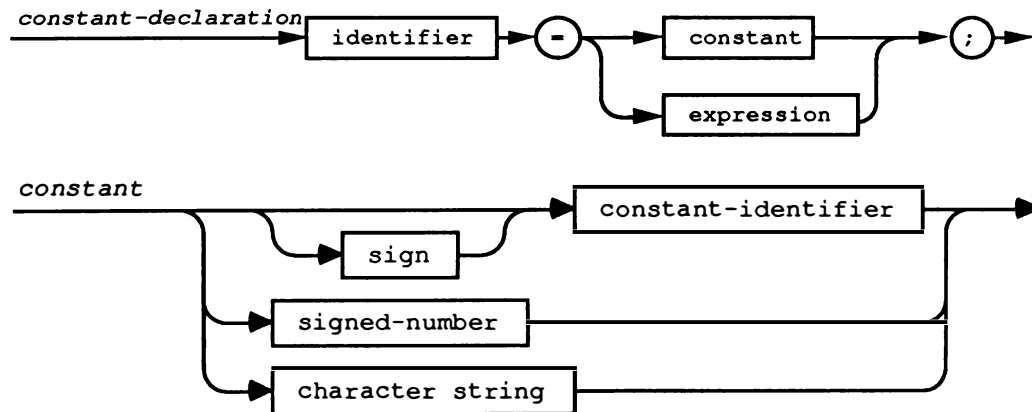
All string-type values have a **length** attribute. In the case of a character-string, the length is fixed; it is equal to the number of characters in the string as enclosed within apostrophes. Two adjacent apostrophes within a character-string count as a single apostrophe and thus count as a single character in the string's length.

Examples of character-strings:

```
'Pascal'    'THIS IS A STRING'    'Don''t worry!'
'A'        ',,'                ''
```

1.7 Constant-Declarations

A **constant-declaration** declares an identifier to denote a constant within the block that contains the declaration. A constant-identifier may not be included in its own declaration.



A constant-identifier following a sign must denote a value of type set, character-string, integer, longint, boolean, char, real, double, computational or extended (see §3.1).

A constant expression must evaluate to a set, character-string, integer, longint, boolean or char. You can use the following operators and standard functions in a constant expression:

* (multiplication)	+ (addition)	- (subtraction)
* (set intersection)	+ (set union)	- (set difference)
div	mod	abs
chr	ord	sizeof
sqr		

1.8 Comments

The constructs:

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

are called **comments**.

You can use a comment anywhere you can use a blank. THINK Pascal moves any comments embedded in a line of code to the end of the line. Comments that appear alone in a line are left alone. If you try to extend a comment over more than one line, THINK Pascal adds a } or *) at the end of one line and a { or (* at the beginning of the next.

You can't nest comments within other comments. A } or a *) always terminates a comment.

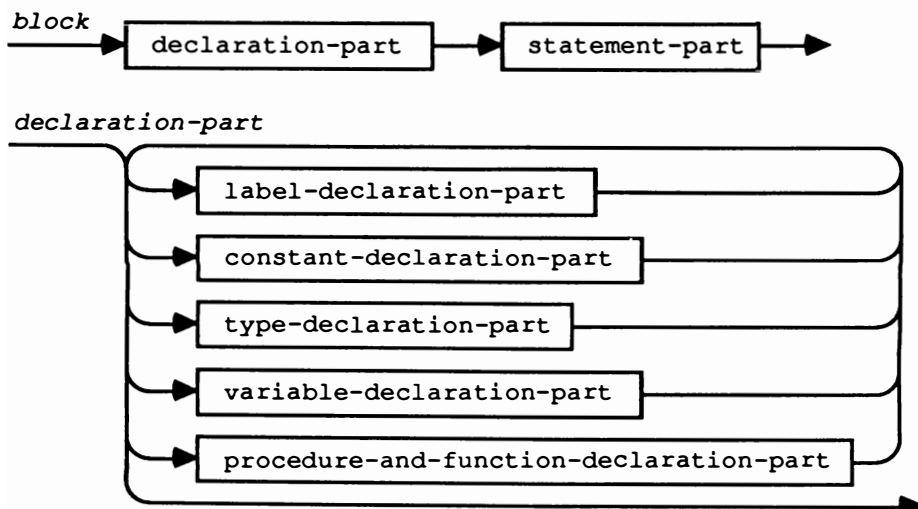
1.9 Compiler Options

A **compiler option** is a comment that contains the \$ character immediately following the initial comment delimiter; for example, { \$ or (* \$. The \$ character is followed immediately by the mnemonic of the compiler option. (see Chapter 15 for available options). A compiler option is always on a line by itself.

2.0 Blocks, Scopes, and Activations

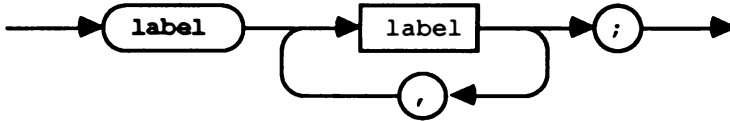
2.1 Definition of a Block

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

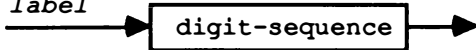


A **label-declaration-part** declares labels (see §1.5) that mark statements in the corresponding statement-part. Each label must mark exactly one statement in the statement-part.

label-declaration-part

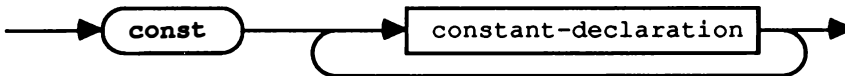


label



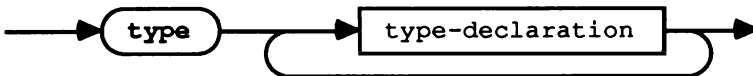
A **constant-declaration-part** contains constant-declarations (see §1.7) local to the block.

constant-declaration-part



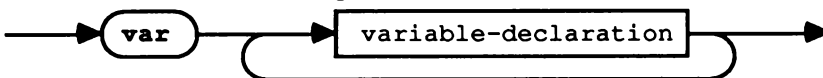
A **type-declaration-part** contains type-declarations (see §3) local to the block.

type-declaration-part



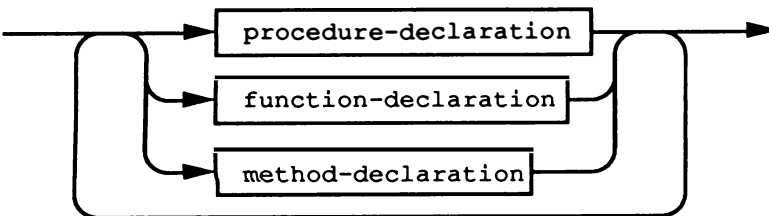
A **variable-declaration-part** contains variable-declarations (see §4) local to the block.

variable-declaration-part



A **procedure-and-function-declaration-part** contains procedure-, function-, and method-declarations (see §7) local to the block.

procedure-and-function-declaration-part



Note: A method declaration can only be declared in the declaration-part of a program or in the implementation part of a unit, not in the declaration-part of a procedure or function.

The statement-part specifies the statements or algorithmic actions (see §6) to be executed upon an **activation** (see §2.3) of the block.



2.2 Rules of Scope

This section describes the rules that THINK Pascal uses to determine what identifiers are visible to a particular block.

2.2.1 Scope of a Declaration

The appearance of an identifier or label in a declaration declares the identifier or label. All other applied occurrences of the identifier or label must be within the **scope** of this declaration.

The scope of a declaration is the block that contains the declaration, and all blocks enclosed by that block except as explained in §2.2.2 and §2.2.4 below. (See also §8.3 for scope rules for Units.)

2.2.2 Redeclaration in an Enclosed Block

Suppose that *outer* is a block, and that *inner* is another block declared within *outer*. If an identifier declared in block *outer* has the same spelling as an identifier declared in block *inner*, then block *inner* and all blocks enclosed by *inner* are excluded from the scope of the declaration in block *outer*.

2.2.3 Position of Declaration within Its Block

The declaration of an identifier or label must precede all applied occurrences of that identifier or label in the program text. In other words, identifiers and labels cannot be used until they are declared.

There are two exceptions to this rule. In a type-declaration-part, the base-type of a pointer-type (see §3.4) can be an identifier that has not yet been declared. In this case, the identifier must be declared somewhere in the same type-declaration-part as the pointer-type. The base-type of an object-type (§3.2.5) can also be an identifier that has not been declared, and the identifier must be declared somewhere in the same type-declaration-part as the object type.

2.2.4 Redeclaration within a Block

An identifier or label cannot be declared more than once within a block, unless it is declared within a contained block, or it is in a record's field-list.

A field-identifier (see §§3.2.2 and 4.3.2) is declared within a record-type. It is meaningful only in combination with a reference to a variable of that record-type. Therefore, a field-identifier can be declared within the same block as another identifier with the same spelling, as long as it has not been declared previously in the same field-list. An identifier that has been declared can be used again as a field-identifier in the same block.

2.2.5 Identifiers of Standard Objects

THINK Pascal provides a set of standard (predeclared) constants, types, procedures, and functions that behave as if they were declared in a block that contains the entire program. In addition, there are two standard file variables, `input` and `output`, that (if used) are "declared" in the program block itself (see §9.4).

2.2.6 Scope of Interface Identifiers

Each interface-part or program with a `uses`-clause (see §8.4) is supplied with the identifiers associated with the unit given in the `uses`-clause. This means that these identifiers behave as though they were declared directly in each interface-part or program with the `uses`-clause.

2.2.7 Scope of fields

In the implementation of a method of an object-type (§3.2.5) all of the identifiers and components of the type and its ancestors are meaningful. The behavior is as if the statement-list of the block were wrapped in `with self do begin ... end`.

2.3 Activations

The execution of a block is referred to as an activation of the block. At any given time, a block normally has either no activations (if it is not currently being executed) or one activation (if it is being executed). It is, however, possible for a block to have multiple activations if it is recursive or if it is mutually recursive with one or more other procedures or functions. A typical example of a recursive function is:

```
function factorial (n: integer): integer;
begin
    if n<=1 then
        factorial := 1
    else
        factorial := n * factorial(n-1)
end;
```

Thus, the execution of `factorial (5)` would lead to 5 activations of `factorial` as follows:

```
factorial (5)    = 5 * factorial (4)
                 = 5 * 4 * factorial (3)
                 = 5 * 4 * 3 * factorial (2)
                 = 5 * 4 * 3 * 2 * factorial (1)
                 = 5 * 4 * 3 * 2 * 1
```

`Factorial` calls itself repeatedly, creating new activations, until the parameter `n` is less than or equal to 1. The last activation then unwinds itself by passing back a result and terminating the activation. The next to last activation then performs the multiplication with the result, passes back its result, and terminates its activation, and so on.

Every activation has, in effect, its own copy of every parameter and variable declared local to the block being activated. Thus, each activation of `factorial` has its own copy of its parameter, which is named `n` in all activations. Because each activation has its own copy of all locally declared

entities, it does not disturb the local entities of any previous activation. §7.3.3 gives a very detailed example of this.

The activation of the program-block also creates the variables local to each used unit's interface-part and implementation-part (see §8.3). This means that there is only one copy of each unit's local variables and that they exist as long as the program is executing.

Note: The value of a variable declared within a particular block is undefined for each new activation of the block. Likewise, the value of every component of a structured-type variable (see §3.2) is initially undefined for each new activation. The value of a structured-type variable remains undefined until it has no components whose values are undefined.

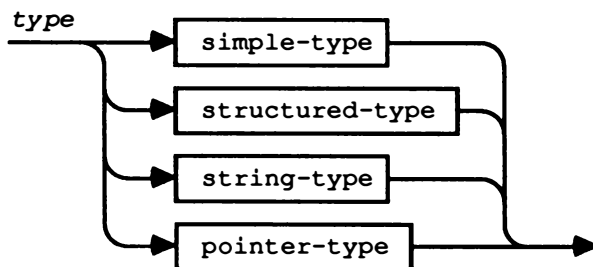
3.0 Types

A **type** is used in declaring variables and in declaring other types. The type of a variable determines the set of values the variable can assume and the operations that can be performed upon it. A **type-declaration** associates an identifier with a type.

type-declaration



type



The occurrence of an identifier on the left-hand side of a type-declaration declares it as a type-identifier for the block in which the type-declaration occurs. A type-identifier may not be included in its own declaration, except for pointer-types (see §2.2.3 and §3.4).

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

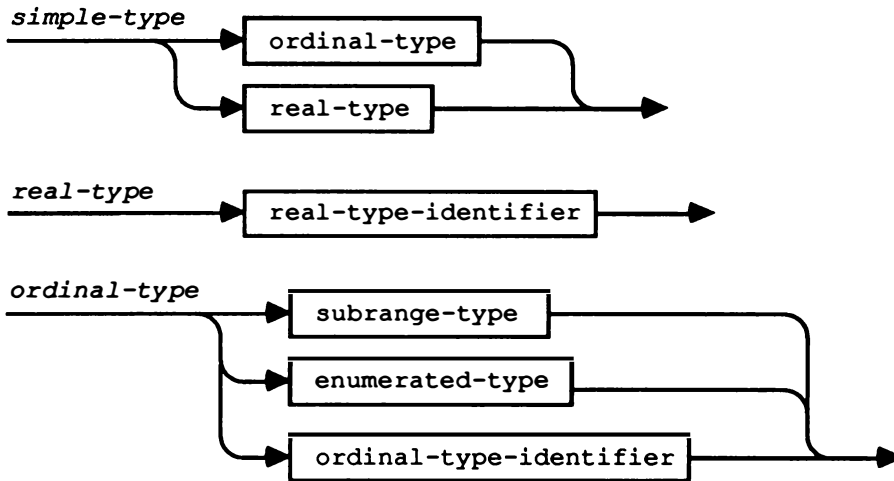
- simple-type-identifier
- structured-type-identifier
- pointer-type-identifier
- ordinal-type-identifier
- integer-type-identifier

- real-type-identifier
- string-type-identifier

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

3.1 Simple-Types

All the **simple-types** define ordered sets of values.



An **integer-type-identifier** is one of the standard identifiers **integer** or **longint**. Constant integer-type values can be denoted as described in §§1.4 and 1.7.

A **real-type-identifier** is one of the standard identifiers `real`, `double`, `extended`, or `computational`. Constant real-type values can be denoted as described in §§1.4 and 1.7.

3.1.1 Ordinal-Types

Ordinal-types are a subset of the simple-types that have the following special characteristics:

- The possible values of an ordinal-type are an ordered set and every value has an **ordinality**, which is an integral value. Except for integer-types, the first value of every ordinal-type has ordinality 0, the next has ordinality 1, etc. For integer-types, the ordinality of a value is the value itself. Every value of an ordinal-type except the first has a **predecessor** based on the ordering of the type, and every value of an ordinal-type except the last has a **successor** based on the ordering of the type.
- The standard function `ord` (see §10.4.1) can be applied to any value of an ordinal-type, and it returns the ordinality of the value.

- The standard function `pred` (see §10.4.4) can be applied to any value of an ordinal-type, and it returns the predecessor of the value.
- The standard function `succ` (see §10.4.3) can be applied to any value of an ordinal-type, and it returns the successor of the value.

The application of `pred` to the first value of an ordinal-type is an error. Likewise, the application of `succ` to the last value of an ordinal-type is an error.

All simple-types except the real-types are ordinal-types.

There are four standard ordinal-types denoted by the standard identifiers:

```
integer
longint
char
boolean
```

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

3.1.1.1 Standard Ordinal-Types

Integer	The integer-type <code>integer</code> has a set of values that are a subset of the whole numbers. The standard <code>integer</code> constant <code>maxint</code> is defined to be $2^{15}-1$, i.e. 32,767. A variable of type <code>integer</code> can have any value in the range <code>-maxint-1..maxint</code> . Values of type <code>integer</code> are 16-bit, signed, 2s-complement numbers.
Longint	The integer-type <code>longint</code> is a type that has a set of values that are also a subset of the whole numbers, a somewhat larger subset than those of <code>integer</code> . The standard <code>longint</code> constant <code>maxlongint</code> is defined to be $2^{31}-1$, i.e. 2,147,483,647. A variable of type <code>longint</code> can have any value in the range <code>-maxlongint-1..maxlongint</code> . Values of type <code>longint</code> are 32-bit, signed, 2s-complement numbers.

There are several **arithmetic operators** that may be used to perform arithmetic with integer-type values. All arithmetic with just integer-type `integer` operands yields results of type `integer`. When one or both operands are of integer-type `longint`, the result is always of type `longint`. A `longint` value may always be used where an `integer` value is required provided that the value falls within the range `-maxint-1..maxint`.

Boolean The ordinal-type `boolean` is an enumerated-type (see §3.1.1.2) defined as:

```
type boolean = (false, true);
```

As a consequence of boolean being an enumerated-type, the following relationships hold:

```
false      < true
ord(false) = 0
ord(true)  = 1
succ(false) = true
pred(true) = false
```

Char The ordinal-type `char` has a set of values that are characters. The ordering of the values is defined by the ordering of the Macintosh character set. The function-call `ord(c)`, where `c` is a `char` value, returns the ordinality of `c` (see §10.4.1).

A character-string of length 1 may be used to denote a constant `char` value, provided that the character is a printable character. Any value of type `char` may be generated via the standard function `chr` (see §10.4.2).

The redeclaration of a standard type-identifier does not affect the operand-types, parameter-types, or result-types of certain standard operators, procedures, and functions declared to be that standard type (see §5.1 and §10). Neither does it affect the type of any literal token (such as a number) declared to be of that type (see §1).

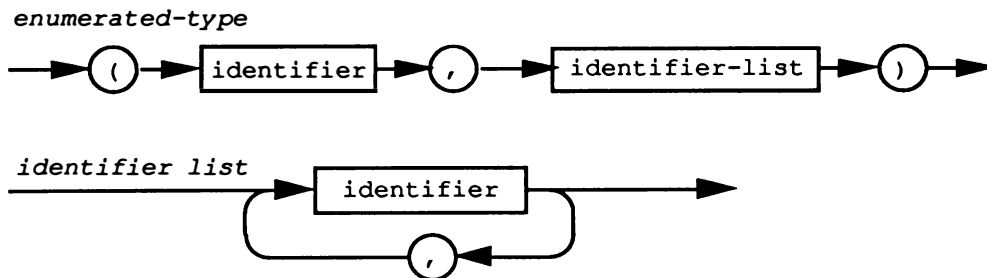
The redeclaration of the standard constant-identifiers `maxint` or `maxlongint` has no effect on the set of possible values for the integer-types.

There are several contexts in which an expression is required to be of the standard type `boolean` (see §6). A redeclaration of the standard type-identifier `boolean` does not alter this requirement.

However, the redeclaration of the standard enumerated-constant identifiers `false` and `true` will affect the value of these identifiers.

3.1.1.2 Enumerated-Types

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



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

The ordinality of an enumerated-constant is its position in the identifier-list in which it is declared, where the ordinality of the first enumerated-constant in the list is always 0. The ordinality of a value of an enumerated-type is the ordinality of the enumerated-constant with the same value.

When the `ord` function (see §10.4.1) is applied to a value `v` of an enumerated-type, it returns an integer-type value that is the ordinality of `v`.

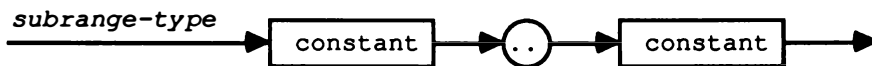
Examples of enumerated-types:

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

Given these declarations, `yellow` is an enumerated-constant of type `color` with ordinality 1, `spade` is an enumerated-constant of type `suit` with ordinality 3, and so forth.

3.1.1.3 Subrange-Types

A **subrange-type** has a subset of the values of some ordinal-type that lie within a certain range. The syntax for a subrange-type is:



Both constants in a subrange-type must be of an ordinal-type and both must be of the same ordinal-type. For all subrange-types of the form `a .. b`, `a` must be less than or equal to `b`. The ordinal-type of `a` and `b` is referred to as the **host-type** of the subrange-type. The values of a subrange-type `a .. b` are those values of its host-type whose ordinalities lie between the ordinalities of `a` and `b` inclusive.

Examples of subrange-types:

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

A variable of subrange-type possesses all the properties of variables of the host-type, with the restriction that its value must always be one of the values in the range defined by the subrange-type.

3.1.2 Real-Types

The **real-types** have sets of values that are subsets of the real numbers; in particular those subsets of the real numbers that may be represented with a floating-point notation using a fixed number of digits. In general, a floating-point notation of a value n is comprised of a set of three values m , b , and e such that

$$m \times b^e \approx n$$

A real-type uses a floating point notation where b is always 2, and where m and e are integral values that lie in a range that depends on the particular real-type. The range of values that m and e can have determine the range and precision of the real-type.

Note: For detailed information about the representation of real-type values, see *Apple Numerics Manual, Second Edition* (Addison-Wesley).

Doing arithmetic with real-type values can lead to results that cannot even be approximated with a floating-point notation. For instance, the division of one by zero is nominally ∞ . Other results make even less sense, such as dividing zero by zero. There are, in fact, special real-type values to represent such results. Normally, however, the generation of such a result by a program is an error and results in the premature termination of the execution of the program.

There are four standard real-types: `real`, `double`, `extended`, and `computational`. No other real-types can be defined.

The approximate range of positive values representable with the real-types `real`, `double`, and `extended` as well as their precision are in Table 3-1:

Table 3-1 The Real-Types

Real-Type	Range	Decimal Digits
<code>real</code>	1.5×10^{-45} to 3.4×10^{38}	7-8
<code>double</code>	5.0×10^{-324} to 1.7×10^{308}	15-16
<code>extended</code>	1.9×10^{-4951} to 1.1×10^{4932}	19-20

Any negative value whose absolute value lies within these ranges is representable with these real-types.

All real-type operands are converted to `extended` before any arithmetic is performed on them, and the results of such arithmetic are always of type `extended`. An `extended` value may be used anywhere a `real` or `double` value is required provided that the value falls within the range of values for `real` or `double`, respectively.

The real-type `computational` is a special real-type where e is always zero, i.e. only integral values may be represented with `computational`. Values of type `computational` must lie in the (exact) range $-2^{63}+1$ to $2^{63}-1$, which is approximately -9.2×10^{18} to 9.2×10^{18} .

All computational operands are converted to extended before any arithmetic is performed on them, and the results of such arithmetic are always of type extended. An extended value may be used anywhere a computational value is required provided that the value, when rounded to an integral value, falls within the range of values for computational.

The type computational is intended for those applications where precise, fixed-point decimal values are required. No explicit decimal point is ever assumed for a computational value, but one can be implicitly assumed by the application. For instance, one can define

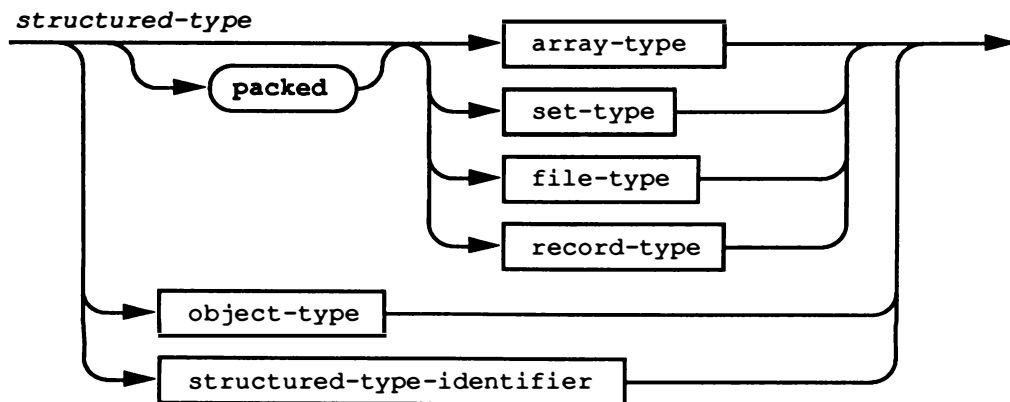
```
type cents = computational;
```

and perform calculations on values of type cents that may also be interpreted as calculations on dollar values with an implied decimal point to the left of the second to last decimal digit. A special form of the standard procedures write and writeln (see §9.4.3.5) may be used to output a computational value with a decimal point inserted between any two decimal digits in the value.

Note: The SANE data types single and comp are equivalent to the Pascal real-types real and computational respectively. All of these names are accepted by THINK Pascal.

3.2 Structured Types

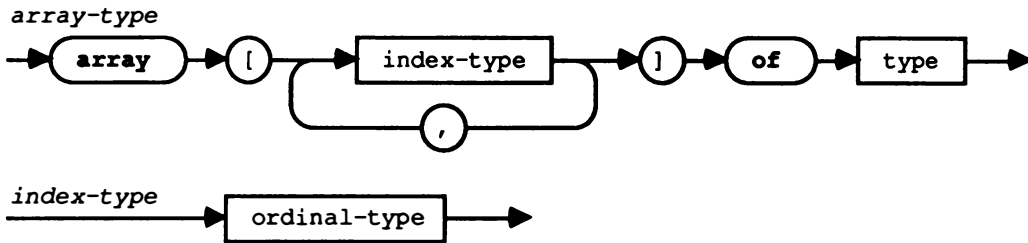
A **structured-type** is characterized by the kind of structuring it embodies and by the types of the components subject to such structuring. The type of a component may itself be structured, in which case the resulting structured-type exhibits more than one level of structuring. There is no inherent limit on the number of levels to which types can be structured other than the amount of memory available.



The use of the word **packed** in the declaration of a structured-type indicates that storage organization of all values of that type should be compressed to economize storage, even if this causes the access of a component of a variable of this type to be less efficient.

3.2.1 Array-Types

An **array-type** is a linear array (or vector) of components that are all of one type, called the **component-type** of the array.



The type that follows the word **of** is the component-type of the array. The number of elements is determined by the **index-type** of the array, which must be an ordinal-type.

If the component-type of an array-type is also an array-type, the result can be regarded as a single multi-dimensional array. An equivalent shorthand notation may be used to declare multi-dimensional arrays that entails declaring a single array with multiple index-types. For example, the type

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

is equivalent to the type

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

where “equivalent” means that they will be interpreted in the same way.

Examples of array-types:

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

A component of an array can be accessed by referencing the array and supplying one or more indices (see §4.3.1).

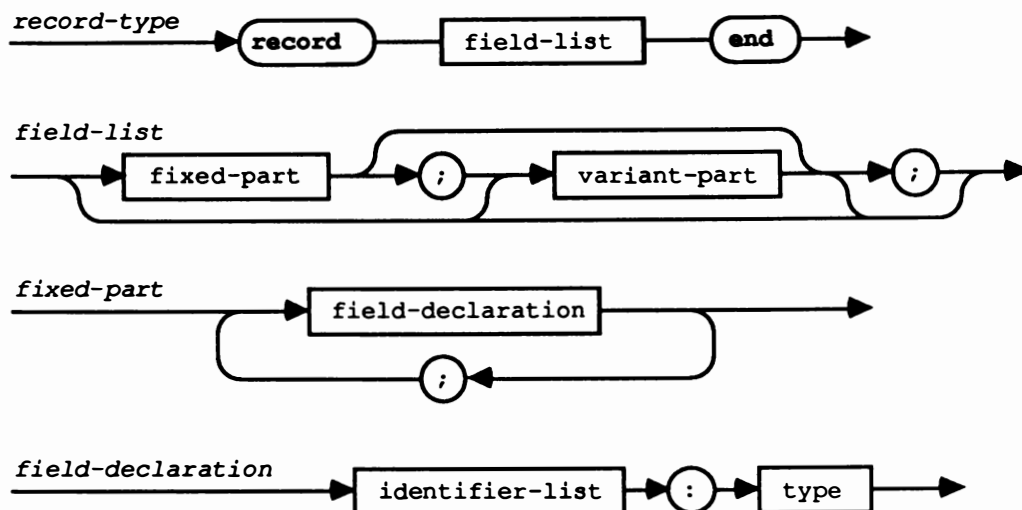
An array-type of the form

```
packed array[1..n] of char
```

is referred to as a **packed-string-type** with *n* components (in American National Standard Pascal these are called **string-types**; they are called packed-string-types here to avoid confusion with THINK Pascal’s string-types (see §3.3). A packed-string-type has certain properties not shared by other array-types or structured-types (see §§3.5 and 5.1.5).

3.2.2 Record-Types

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



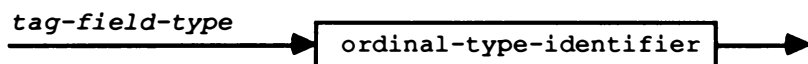
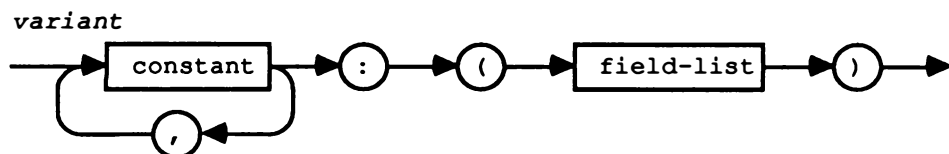
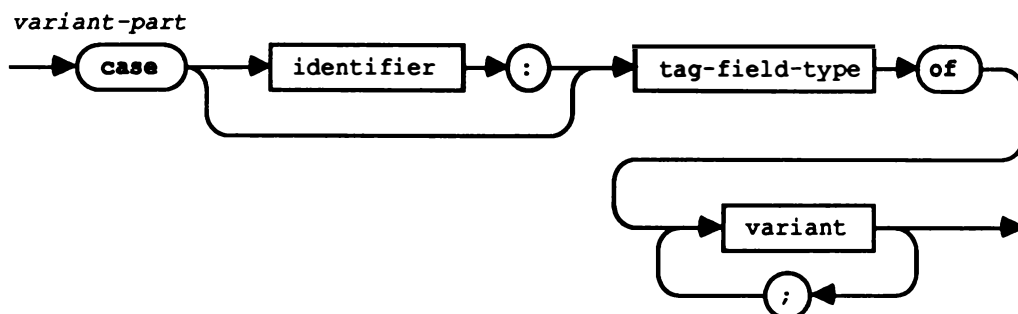
The **fixed-part** of a record-type specifies a field-list that is always accessible in a variable of the record-type, giving an identifier and a type for each field. Each of these fields contains data that is always accessed in the same way (see §4.3.2).

Example of a record-type:

```

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

A **variant-part** consists of alternative field-lists of which only one is accessible at a given time. Each alternative field-list is called a **variant**.



Each variant is preceded by one or more constants. All of the constants must be distinct and must be of an ordinal-type that is compatible with the tag-type (see §3.5).

The variant-part allows for an optional identifier that denotes a **tag-field**. If a tag-field is present, it is considered a field of the fixed-part. The value of the tag-field indicates which variant is active, and thus which variant's fields are accessible at a given time. If there is no tag-field explicitly given, then all fields in all variants are always accessible, i.e. the active variant is the one containing the field most recently referenced.

It is an error to alter the value of a tag-field while a reference to a field of the active variant exists. Whenever the value of the tag-field changes, the values of all of the fields in the newly active variant are undefined. When the value of the tag-field is undefined, the values of all fields in all variants of the corresponding variant-part are undefined.

Note: All of the variants of a variant-part share the same region of memory within a record variable. This is because only one particular variant is ever in use at a time (see also §10.1).

It is not uncommon for Pascal programmers to use a variant-part with no explicit tag-field as a means of converting a value of one type to a value of another type, namely by assigning a value into a field of one variant and referencing the corresponding field in another variant. For example:

```
var
  r: record
    case boolean of
      false: (CharVal: char);
      true:  (IntVal: integer)
    end;
...
r.CharVal := 'W';
...
c := r.IntVal;
...
```

However, this kind of access assumes a knowledge of the underlying representation of variables in memory, and is therefore not recommended. This sort of type coercion should be performed using the predefined functions `chr` and `ord`, or with type casts (see §5.4) which are guaranteed to do what you would expect.

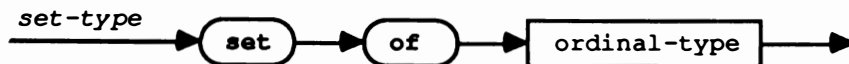
Examples of record-types with variants:

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

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

3.2.3 Set-Types

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



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

The empty set (see §5.3) is a possible value of every set-type.

3.2.4 File-Types

A **file-type** is a structured-type consisting of a linear sequence of components that are all of one type, the **component-type**. The component-type may be any type that is not a file-type or a structured-type that contains a file-type component at any level of structuring. The number of components in a file-type variable is not fixed by the file-type declaration.



The standard file-type `text` denotes a packed file of characters organized into lines. Files of type `text` are supported by the specialized I/O procedures discussed in §9.4.

§4.3.3 and §9 discuss methods of accessing file components.

3.2.5 Object-Types

An **object-type** is similar to a record-type (§3.2.2) except that in addition to fields, object-types can contain components called **methods**. A method is a procedure or function that operates on the object.

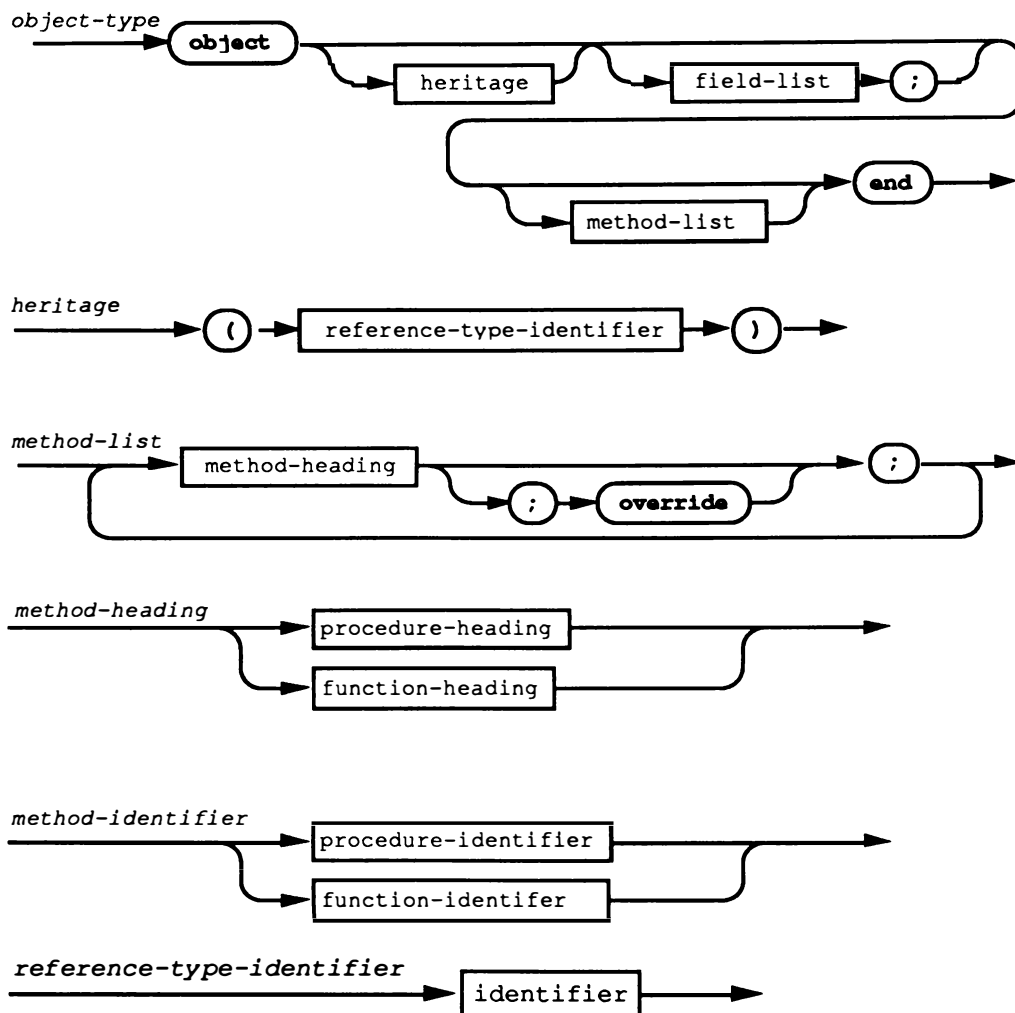
An object-type can be a descendant of any other object-type. It inherits all of the fields and methods of its parent. A descendant object-type can define new fields, new methods, and **override** inherited methods with its own implementation.

Objects are created dynamically during program execution with the `new` and `dispose` procedures (§10.1.1 and §10.1.2). But instead of being identified by pointers, objects are identified by analogous values called **references**.

A type identifier associated with an object type always denotes the reference type whose base type is that object type.

A variable of type object is not the object itself, but rather a reference to the object. So a variable of type object is called a **reference variable**. A reference variable is said to possess **reference-type**. Any number of reference variables can refer to the same object.

Note: This is analogous to how pointers work: a variable of type "pointer to integer" is a pointer variable, not an integer. The pointer variable points to an integer. Several pointer variables can point to the same integer.



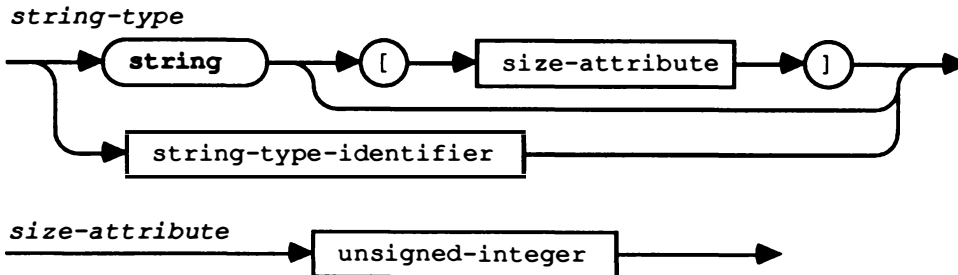
3.3 String-Types

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

A string-type has a static **size** attribute that is an integral value in the range from 1 to 255. The size is a maximum limit on the length of any value of this type. If an explicit size attribute is not given for a string-type, then it is given a size of 255 by default.

The current value of the length attribute of a string-type value is returned by the standard function `length` (see §10.5.1). A **null-string** is a string-type value with a length of zero.

Note: Do not confuse the size with the length.



The ordering relationship between any two string values is determined by lexical comparison based on the ordering relationship between character values in corresponding positions in the two strings. When the two strings are of unequal lengths, each character in the longer string that does not correspond to a character in the shorter one compares “higher”; thus the string value `'attribute'` is greater than the value `'at'`. Two strings must always have the same lengths to be equal; `'X '` (X followed by a space) is always greater than `'X'` (just X).

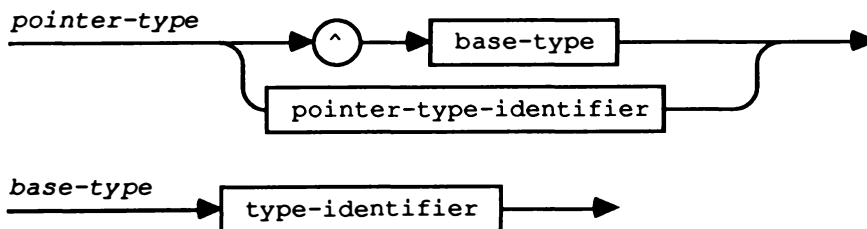
A null-string is only equal to another null-string and is less than every other possible string value.

There are aspects of string-types that make it seem both a simple-type and a structured-type at the same time; it can therefore be considered neither. However, as explained in §4.3.1, individual characters in a string can be accessed as if they were components of an array. Operators applicable to strings are specified in §5.1.5. Standard procedures and functions for manipulating strings are described in §10.5.

3.4 Pointer-Types

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

Pointer values are created by the standard procedure `new` (see §10.1.1), by the `@` operator (see §5.1.6), and by the standard procedure `pointer` (see §10.2.4).



The base-type may be an identifier that has not yet been declared. In this case, it must be declared somewhere in the same type-declaration-part as the pointer-type (see §2.2.3).

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

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

3.5 Identical and Compatible Types

Two types **may** or may not be **identical**, and identity is required in some contexts. Other times, even if not identical, two types need only be **compatible**, and other times **assignment-compatibility** is required.

3.5.1 Type Identity

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

- For variable parameters, the actual and formal parameters must be identical types (see §7.3.2).
- The result types of actual and formal functional parameters must be identical (see §7.3.4).
- Value and variable parameters within parameter-lists of actual and formal procedural or functional parameters must be identical types (see §7.3.5).

Two types, T_1 and T_2 , are **identical** if one of the following is true:

- T_1 and T_2 are the same type-identifier.
- T_1 is declared to be equivalent to a type identical to T_2 .

What the latter, somewhat circular statement implies is that T_1 need not be declared directly to be equivalent to T_2 ; thus the type-declarations

```
T1 = integer;
T2 = T1;
T3 = integer;
T4 = T2;
```

result in T_1 , T_2 , T_3 , T_4 , and **integer** all being identical types.

However, the type-declarations

```
T5 = set of integer;
T6 = set of integer;
```

do not result in T_5 and T_6 being identical, since **set of integer** is not a type-identifier.

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

```
V1, V2: set of integer
```

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

```
V1: set of integer;  
V2: set of integer;  
V3: integer;  
V4: integer;
```

result in V_3 and V_4 being of identical type, but not V_1 and V_2 .

3.5.2 Compatibility of Types

Compatibility between two types is sometimes required. Specific instances where type compatibility is required are noted elsewhere in this manual. Type compatibility is, more importantly, often a precondition of assignment-compatibility.

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

- Both types are identical.
- Both types are real-types.
- Both types are integer-types.
- One type is a subrange of the other.
- Both types are subranges of identical host-types.
- Both types are set-types with compatible base-types.
- Both types are packed-string-types with the same number of components.
- One type is a string-type and the other is a string-type, packed-string-type, or char-type.

3.5.3 Assignment-Compatibility

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

A value V_2 of type T_2 is assignment-compatible with a variable V_1 of type T_1 (i.e. $V_1 := V_2$ is permissible) if any of the following are true:

- T_1 and T_2 are identical types and neither is a file-type or a structured-type that contains a file-type component at any level of structuring.
- T_1 and T_2 are real-types and the value of type T_2 is within the range of possible values of T_1 .
- T_1 is a real-type and T_2 is an integer-type.
- T_1 and T_2 are compatible ordinal-types, and the value of type T_2 is within the range of possible values of T_1 .
- T_1 and T_2 are compatible set-types, and all the members of the value of type T_2 are within the range of possible values of the base-type of T_1 .

- T_1 and T_2 are compatible packed-string-types.
- T_1 is a char-type and the value of type T_2 is a string-type and has a length of 1.
- T_1 is a packed-string-type with n components and the value of type T_2 is a string-type and has a length of n .
- T_1 is a string-type with a size of n and the value of type T_2 is a string-type and has a length less than or equal to n .
- T_1 is a string-type with a size of n and the value of type T_2 is a packed-string-type with a number of components less than or equal to n .
- T_1 is a string-type and T_2 is a char-type.
- T_2 is a reference-type that inherits from T_1 .

It is an error if assignment-compatibility is required and none of the above is true.

3.6 The Type-Declaration Part

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

Example of a type-declaration-part:

```

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

```

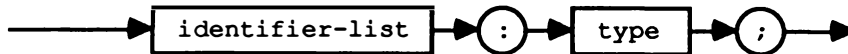
In the above example `count`, `range`, and `integer` denote identical types. The type `year` is compatible and assignment-compatible with, but not identical to, the types `range`, `count`, and `integer`.

4.0 Variables

4.1 Variable-Declarations

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

variable-declaration



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

Examples of variable-declarations:

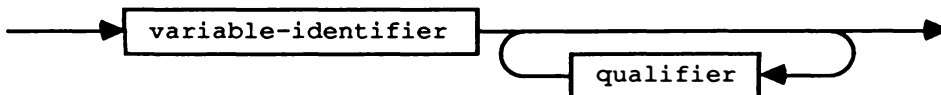
```

x,y,z: real;
i,j: integer;
k: 0..9;
p,q,r: boolean;
operator: (plus, minus, times);
a: array[0..63] of real;
c: color;
f: file of char;
hue1,hue2: set of color;
pl,p2: person;
m,m1,m2: array[1..10,1..10] of real;
coord: polar;
pooltape: array[1..4] of tape;
  
```

4.2 Variable-References

A **variable-reference** denotes either an entire variable, a component of a structured or string-type variable, a dynamic-variable pointed to by a pointer-type variable, or the file-buffer of a file-type variable.

variable-reference

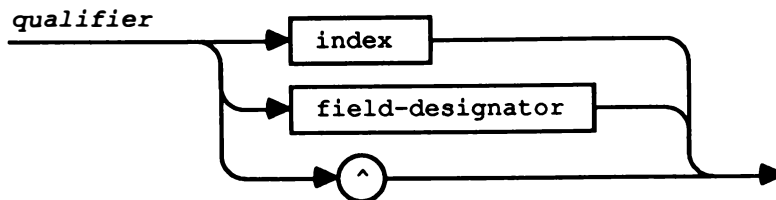


variable-identifier



4.3 Qualifiers

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



As an example, an array identifier with no qualifier is a reference to the entire array-variable:

```
xResults
```

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

```
xResults[current+1]
```

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

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

If the field is a pointer, the field-designator may be followed by the symbol ^ to distinguish between the pointer field and the dynamic-variable being pointed to:

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

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

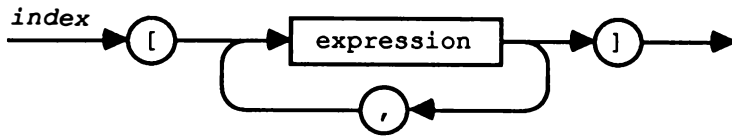
```
xResults[current+1].link^[i]
```

and so on...

4.3.1 Arrays, Strings, and Indices

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

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



Examples of indexed arrays:

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

Each expression in the index selects a component in the corresponding dimension of the array. The number of expressions must not exceed the number of index-types in the array declaration. It is an error if the result of each expression is not assignment-compatible with the corresponding index-type.

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

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

is equivalent to

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

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

Any value assigned to a component-variable of an array or string must be assignment-compatible with the component-type.

When a string value is manipulated by assigning values to individual character positions, the length of the string is not affected. It is an error to access a character position in a string variable with an index less than one or greater than the length of the string variable. For example, suppose that `strval` has been declared as follows:

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

and that the assignment:

```
strval := 'abcde';
```

has been performed. A reference to:

```
strval[0]
```

would be an error since the reference contains an index less than 1. Likewise, a reference to:

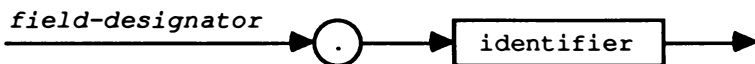
```
strval[6]
```

would be an error since the index is greater than the current length of the string. Note that these references would still be errors even if they occurred on the left-hand side of an assignment statement. To add a character or string to the end of another string, or to manipulate strings in other ways, use the standard procedures described in §10.5.

Note also that, when a program, procedure, or function block is entered, the length of a string variable is initially undefined. Therefore, the manipulation of string variables with indexes before string values have been assigned to these variables can lead to unpredictable results.

4.3.2 Records and Field-Designators

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



Examples of field-designators:

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

It is an error if the field-designator refers to a field in a variant that is not active (see §3.2.2).

Note that a field-designator need not be preceded by a variable-reference to its containing record in a statement within a **with** statement (see §6.2.4).

Fields of an object are accessed with a reference variable (§3.2.5) rather than with a record variable. You can leave out the reference variable and the "." inside a **with** statement. You can also leave out the reference variable in a method block; in this case it is the same as having written **self.** before the field name (see §6.1.2 and §7.3.6).

4.3.3 File-Buffers

Although a **file-variable** may have any number of components, only one component is accessible at any time. The position of the current component in the file is called the **current file position**. Program access to the current component is via a special variable associated with the file, called a **file-buffer**.

The file-buffer is implicitly declared when the file variable is declared. If *f* is a file variable with components of type *t*, then the associated file-buffer is a variable of type *t*.

The file-buffer associated with a file variable is denoted by a variable-reference that refers to the file variable, followed by the ^ symbol. Thus, the file-buffer of file *f* is referenced by *f*^.

§9 describes standard procedures that move the current file position within the file and to transfer data between the file-buffer and the current file component.

4.3.4 Pointers, Reference-Variables, and Dynamic-Variables

The value of a pointer variable is either **nil**, or a value that points to a dynamic-variable.

The dynamic-variable pointed to by a pointer variable is denoted by a variable-reference that refers to the pointer variable followed by the **^** symbol.

Dynamic-variables and pointer values that point to them are created by the standard procedure **new** (see §10.1.1). Additionally, the **@** operator (see §5.1.6) and the standard procedure **pointer** (see §10.2.4) may be used to create pointer values that are not in fact pointers to dynamic-variables, but that will be treated as such.

The constant **nil** (see §3.4) does not point to any object. It is an error if you access a dynamic-variable when the pointer's value is undefined or is **nil**.

Examples of references to dynamic-variables:

```
p1^
p1^.sibling^
```

Note: Although the **^** symbol is used to reference both file-buffers and dynamic-variables, this does not mean that a file variable can be treated like a pointer. Specifically, **ord(f)**, where **f** is a file variable reference, does not yield an address value (see §10.4.1). It is, in fact, an error.

A variable of type object is a **reference variable**. There is no way to reference the object itself, only the fields of an object. That is, you can't use the **^** operator to access the object.

5.0 Expressions

Expressions consist of **operators** and **operands**. Most operators in Pascal are **binary**, i.e. they take two operands; the rest are **unary** and take only one operand. The binary operators and their operands are denoted in the common algebraic fashion: the operands are given with the operator to act upon them in between, e.g., **a+b**. A unary operator is always immediately followed by its operand.

When more complex expressions are written, certain rules must be applied to determine which operands are associated with which operators. For instance, the expression:

```
a+b*c
```

can be interpreted as either **(a+b)*c** or **a+(b*c)**. The **precedence rules** makes the interpretation unambiguous:

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

- When an operand is written between two operators of the same precedence, it is bound to the operator to the left.
- A parenthesized expression is always evaluated before it is applied as an operand.

Table 5-1 gives the precedence of the binary and unary operators:

Table 5-1 Precedence of Operators

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

Thus, $a+b*c$ is interpreted as $a+(b*c)$, since $*$ has a higher precedence than $+$, and $a+b-c$ is interpreted as $(a+b)-c$, since $+$ and $-$ have the same precedence. **precedence rules**

The range of an operator is not infinite. For instance, if a , b , and c are integer-type integer values, and if $a+b$ yields a value greater than `maxint`, then the evaluation $(a+b)-c$ will result in an error whereas $a+(b-c)$ may not. Whenever the order of evaluation of operators is critical, or is otherwise in doubt, parentheses may always be used to force a specific order.

Also be aware that, because the relational operators have the *lowest* precedence, an expression such as

$a < b$ or $c < d$

is wrong because the precedence rules interpret the expression as

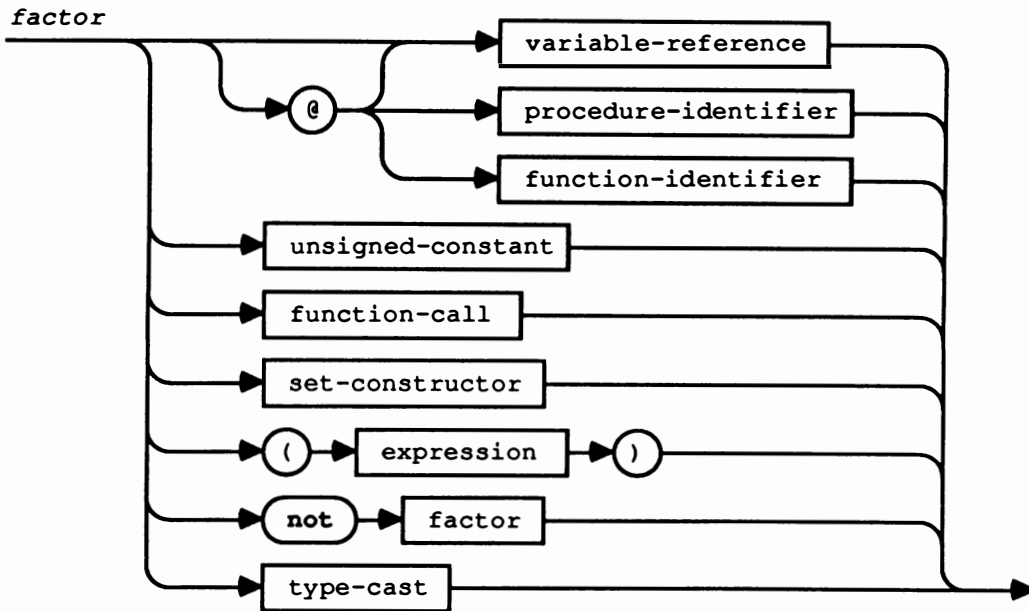
$a < (b \text{ or } c) < d$

which is not a valid expression (see below). Thus, such an expression must be written as

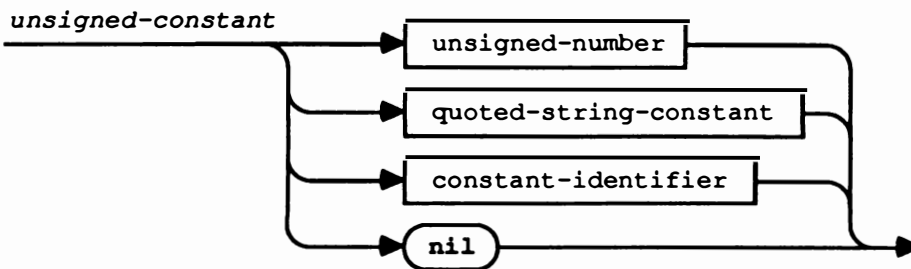
$(a < b) \text{ or } (c < d)$

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

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



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



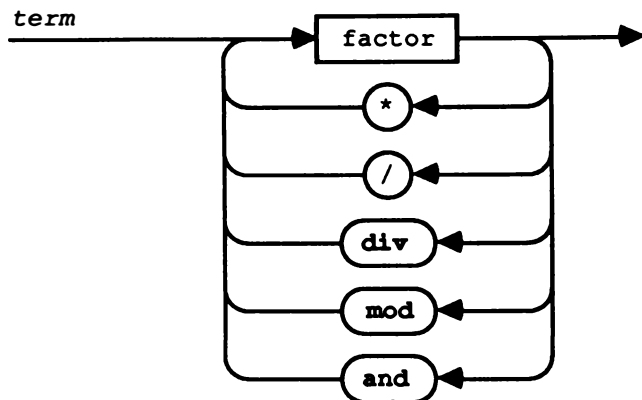
Examples of factors:

```

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

```

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

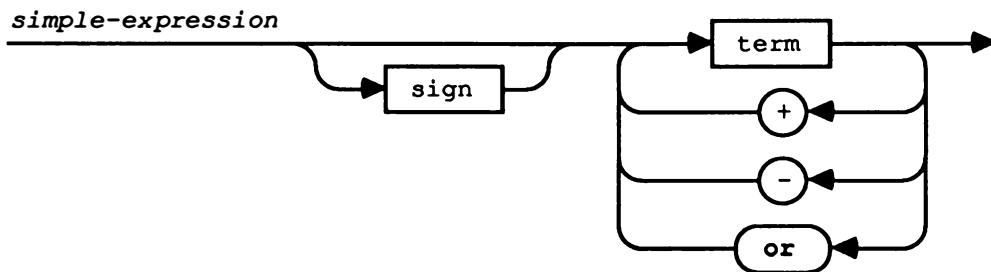


Examples of terms:

```

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

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

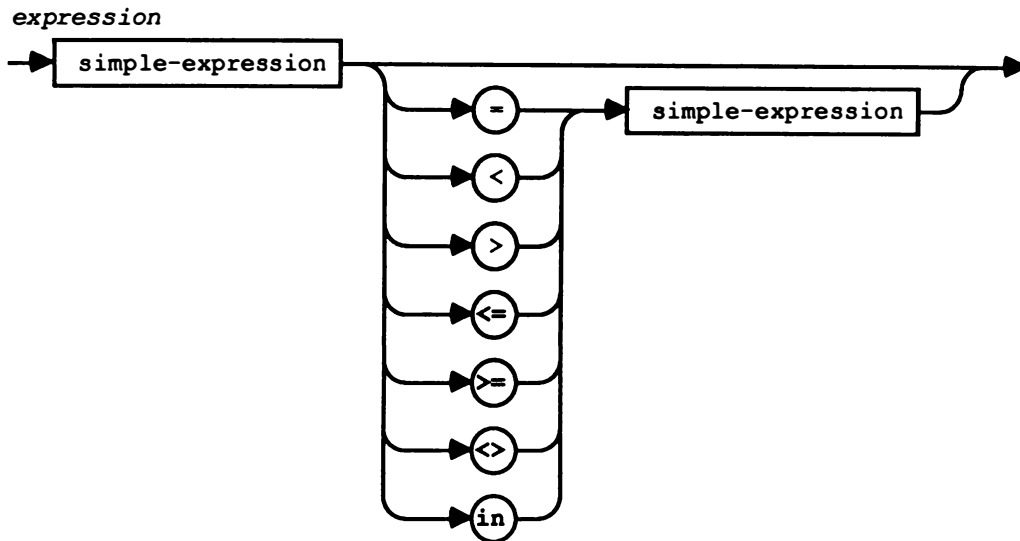


Examples of simple-expressions:

```

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

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



Examples of expressions:

```

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

```

5.1 Operators

This section describes the Pascal arithmetic, relational, and address operators.

5.1.1 Binary Operators: Order of Evaluation of Operands

The order in which the operands of a binary operator are evaluated is unspecified.

A function, in the normal case, simply returns a value. However, a function may also alter the value of variables as a side-effect. Therefore if a function is one operand of a binary operator, and if it modifies the value of the other operand, then the evaluation of the operator may lead to unpredictable results. For instance, if a call to a function $f(x)$ modifies the value of x , then the evaluation of $x + f(x)$ may yield an unexpected result depending on whether or not $f(x)$ is evaluated before the x to the left of the $+$.

5.1.2 Arithmetic Operators

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

Table 5-2 Binary Arithmetic Operations

Operation		Operand Types		Result Type
+	addition	integer	integer	integer
-	subtraction	integer	longint	longint
*	multiplication	longint	longint	longint
		real-type	any type	extended
/	division	any type	any type	extended
div	integer division	integer	integer	integer
		integer	longint	longint
		longint	longint	longint
mod	modulo	integer	integer	integer
		integer	longint	longint
		longint	longint	longint

Note: The symbols +, -, and * are also used as set operators. See Section 5.1.4.

Table 5-3 Unary Arithmetic Operations

Operation		Operand Type	Result Type
+	identity	integer	integer
-	negation	longint	longint
		real-type	extended

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

If both operands of the addition, subtraction, or multiplication operators are of the integer-type `integer`, the result is always of type `integer` and has a value determined by the normal mathematical rules for integer arithmetic. It is an error if the value of the result is outside the range `-maxint-1..maxint`.

If one or both operands of the addition, subtraction, or multiplication operators are of the integer-type `longint`, the result is always of type `longint` and has a value determined by the normal mathematical rules for integer arithmetic. It is an error of the value if the result is outside the range `-maxlongint-1..maxlongint`.

If one of the operands of the addition, subtraction, or multiplication operators is of a real-type, the result is always of type `extended` and has a value that is an approximation of the normal mathematical result. It is an error if the result is outside the range of values representable with the real-type `extended` (see §3.1.2).

Note: See *Apple Numerics Manual, Second Edition* (Addison-Wesley) for more information on all arithmetic operations with operands or results of a real-type.

If the operand of the identity or sign-negation operator is of the integer-type `integer`, the result is always of type `integer` and the absolute value of the result is always identical to the absolute value of the operand.

If the operand of the identity or sign-negation operator is of a real-type, the result is always of type `extended` and the absolute value of the result is always identical to the absolute value of the operand.

If the operand of the identity or sign-negation operator is of the integer-type `longint`, the result is always of type `longint` and the absolute value of the result is always identical to the absolute value of the operand.

The value of `i / j` is always of type `extended` and has a value that is an approximation of the normal mathematical result. It is an error if the result is outside the range of values representable with the real-type `extended` (see §3.1.2). It is an error if `j=0`.

If the operands of the `div` operator are of the integer-type `integer`, the result is always of type `integer`, and the value of `i div j` is the mathematical quotient of `i / j`, rounded toward zero. It is an error if `j=0`.

If one or both of the operands of the `div` operator are of the integer-type `longint`, the result is always of type `longint`, and the value of `i div j` is the mathematical quotients of `i / j`, rounded toward zero. It is an error if `j=0`.

The `mod` operator is defined as:

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

5.1.3 Boolean Operators

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

Table 5-4 Boolean Operations

Operation		Operand Type	Result Type
or	disjunction	boolean	boolean
and	conjunction		
not	negation		

The result of a boolean operation is determined by the normal rules of boolean logic, e.g. a **and** b evaluates to `true` if and only if both a and b are true.

Boolean operators are somewhat unique in that the result of the operation may often be determined by the examination of only one operand. For example, the expression

`(b<>0) and (a/b>10)`

is known to have the value `false` if `b=0` regardless of the value of `a`. However, the evaluation of the subexpression `(a/b>10)` **may or may not** be performed in the evaluation of this expression. It is therefore best to assume that it will and avoid this kind of expression. On the other hand, if an operand of a boolean expression is a function with side-effects (i.e. it modifies a variable as well as returns a value), there is no guarantee that the function will be activated if the result of the operation can be determined solely by the evaluation of the other operand. It is therefore best to also avoid such expressions.

5.1.4 Set Operators

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

Table 5-5 Set Operations

Operation	Operand Type	Result Type
+ union	compatible set types	(see below)
- difference		
* intersection		

The order of evaluation of member-groups and of expressions within member-groups is unspecified.

The results of the set operations are determined by the normal rules of set logic.

- An ordinal value `c` is in the set `a+b` if and only if `c` is in `a` or in `b`.
- An ordinal value `c` is in the set `a-b` if and only if `c` is in `a` and not in `b`.
- An ordinal value `c` is in the set `a*b` if and only if `c` is in `a` and in `b`.

Given the result of a set operation, if the smallest ordinal value that is a member of that result is `a` and if the largest ordinal-value that is a member of the result is `b`, then the type of the result is **set of a..b**.

5.1.5 Relational Operators

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

Table 5-6 Relational Operations

Operation	Operand Type	Result Type
= equal to <> not equal to	One of the following: <ul style="list-style-type: none"> Compatible simple types Compatible string types Compatible packed-string types Compatible pointer types Compatible set types 	boolean
< less than > greater than <= less than or equal to >= greater than or equal to	One of the following: <ul style="list-style-type: none"> Compatible simple types Compatible string types Compatible packed-string types 	boolean
<= subset of >= superset of	Compatible set types	boolean
in member of	(See section 5.1.5.7)	boolean

5.1.5.1 Comparing Ordinals

When the operands of =, <>, <, >, >=, or <= are of an ordinal-type, they must be of compatible types. The result is the mathematical relation of their ordinalities.

5.1.5.2 Comparing Reals

When one operand of =, <>, <, >, >=, or <= is of a real-type, the other must be of a real-type or an integer-type. The result is the mathematical relation of the values represented as extended type values.

Because real-type values are only approximations, the results of these operations may not always be as expected. For instance, if `aReal` is a variable of type `real` and `aDouble` is a variable of type `double`, and if the assignments

```
aReal := 1/3;
aDouble := 1/3;
```

have been performed, then the relation `aReal=aDouble` will return `false`. This is because the value of `aReal` is a representation of 1/3 to only 7-8 decimal digits and the value of `aDouble` is a representation of 1/3 to 15-16 decimal digits. Since the decimal (and even the binary) representation of 1/3 is a repeating sequence of digits, the 8 low-order decimal digits of `aDouble` will differ from the corresponding digits of `aReal` (when converted to extended), which will always be zero.

See *Apple Numerics Manual, Second Edition* (Addison-Wesley) for more information on relational operations with operands of real-type.

5.1.5.3 Comparing Strings

When the relational operators `=`, `<>`, `<`, `>`, `>=`, and `<=` are used to compare strings (see §3.3), they denote lexicographic ordering according to the ordering of the Macintosh character set. Note that any two string values can be compared since all string values are compatible. Additionally, a `char` value is compatible with a string-type value, and when the two are compared, the `char` value is treated as a string-type value with length 1. When a packed-string-type value with `n` components is compared with a string-type value, it is treated as a string-type value with length `n`.

5.1.5.4 Comparing Packed-Strings

The relational operators `=`, `<>`, `<`, `>`, `<=`, and `>=` can also be used to compare two values of a packed-string-type if both have the same number of components. If that number of components is `n`, then the result is the same as if the values were string-type with each having a length of `n`.

5.1.5.5 Comparing Sets

If `a` and `b` are set operands, then

- `a=b` is true if and only if every member of `a` is a member of `b` and every member of `b` is a member of `a`; otherwise, `a<>b`.
- `a<=b` is true if and only if every member of `a` is also a member of `b`.
- `a>=b` is true if and only if every member of `b` is also a member of `a`.

Thus, `a=b` and `a<>b` denote the equivalence and non-equivalence of the sets `a` and `b` respectively, and `a<=b` and `a>=b` denote the inclusion of `a` in `b` and the inclusion of `b` in `a` respectively.

5.1.5.6 Comparing Pointers

The relational operators `=` and `<>` may be applied to compatible pointer-type operands. Two pointers are equal if and only if they point to the same object.

5.1.5.7 Testing Set Membership

The `in` operator yields the value `true` if the value of the ordinal-type operand is a member of the set-type operand; otherwise it yields the value `false`. The type of the left operand must be compatible with the base-type of the right operand.

5.1.6 The @ Operator

A pointer value that points to a variable, procedure, or function can be created with the `@` operator. The operand and result types are shown in Table 5-7.

@ is a unary operator taking a single variable-reference or a procedure or function identifier as its operand and computing the value of its pointer. The type of the value is equivalent to the type of `nil`, i.e. it can be assigned to any pointer variable.

Table 5-7 Pointer Operation

Operation	Operand Type	Result Type
@ pointer formation	One of the following: <ul style="list-style-type: none"> • Variable reference • Procedure identifier • Function identifier 	same as <code>nil</code>

The @ operator is not a standard feature of Pascal, and its indiscriminate use is not recommended. It is intended to be used in conjunction with the Macintosh Toolbox routines.

5.1.6.1 The @ Operator with a Variable

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

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

then the statement

```
twocharptr := @int
```

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

The operand of @ cannot be applied to a component of a **packed** variable.

5.1.6.2 The @ Operator with a Value Parameter

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

```
fooPtr := @foo
```

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

5.1.6.3 The @ Operator with a Variable Parameter

When @ is applied to a formal variable parameter, the result is a pointer to the actual-parameter. Suppose that `fum` is a formal variable parameter of a procedure, `file` is a variable passed to the procedure as the actual-parameter for `fum`, and `fumptr` is a pointer variable.

If the procedure executes the statement

```
fumptr := @fum
```

then `fumptr` is a pointer to `file` and `fumptr^` is a reference to `file` itself.

5.1.6.4 The @ Operator with a Procedure or Function Identifier

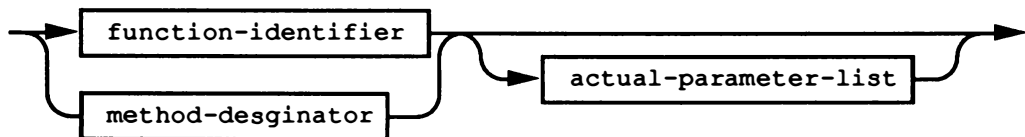
It is possible to apply @ to a procedure or a function, yielding a pointer to the procedure's or function's **entry-point**. This pointer actually points to a **jump table entry** for the routine. See Chapter 13 for more information.

THINK Pascal provides no mechanism for using such a pointer. The typical use for a procedure pointer is to pass it to a Macintosh Toolbox routine. The @ operator can not be applied to predefined, inline, Toolbox, or nested routines.

5.2 Function-Calls

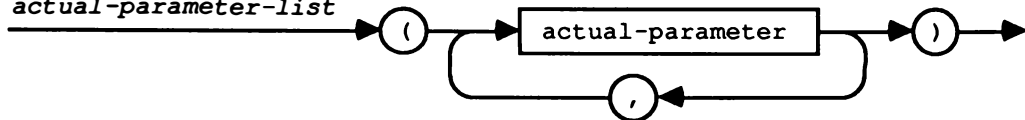
A function-call specifies the activation of the function denoted by the function-identifier. The result returned by the function activation is subsequently used as an expression value. If the corresponding function-declaration contains a list of formal-parameters, then the function-call must contain a corresponding list of actual-parameters. Each actual-parameter is substituted for the corresponding formal-parameter as described in §7.3.

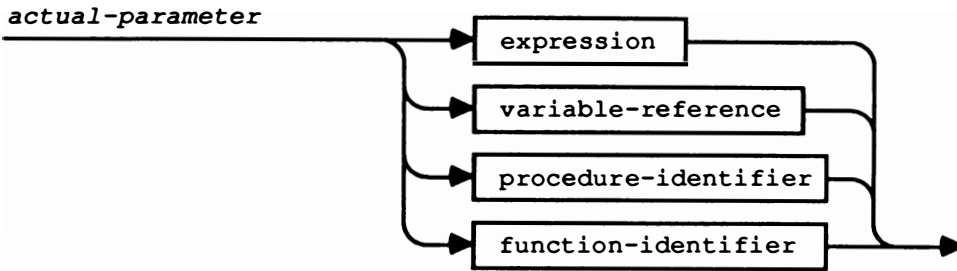
function-call



Note: method-designator is defined in §6.1.2

actual-parameter-list





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

Examples of function-calls:

```

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

5.2.1 Using function calls in l-value contexts

You can use function calls and method calls in l-value contexts. L-value contexts include

- The left-hand side of an assignment statement
- An argument to a **with** statement
- A method call.

For example:

```

const
    WindowList = $09D6;

function GetListHead: ListElementPtr;
...
begin
    nextWindow := WindowPeek(WindowList)^.nextWindow;
    GetListHead^.nextElement := nil;

    with GetListHead^ do
        ...
    end;
```

Inside a function, be careful when you call the function in an l-value context recursively. If the function name is qualified by "^", the function is called recursively and the qualifier is applied to the return value. If the function name isn't qualified or is qualified by "." or "[...]", THINK Pascal assumes you are assigning the return value. For example:

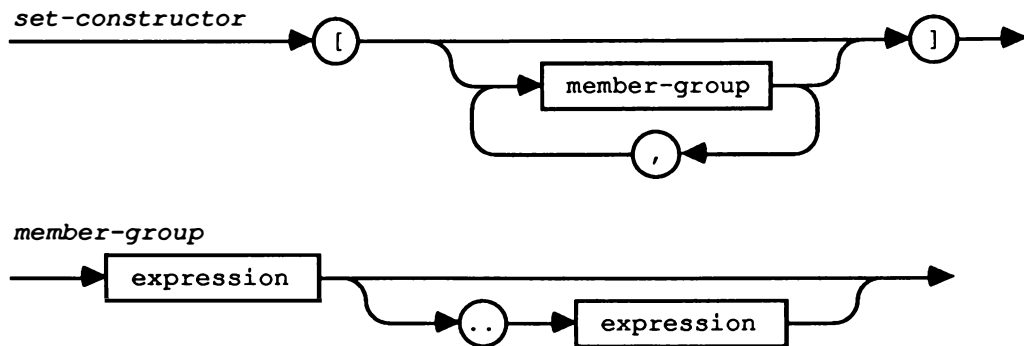
```

function Foo: Ptr;
begin
    Foo := @Bar;      { Makes @Bar the function's return value. }
    Foo^ := 0;        { Calls Foo recursively and assigns 0 to      }
                     { the byte that the result points to.        }
end;

```

5.3 Set-Constructors

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



The notation, [] denotes the empty set, which is assignment-compatible with every set-type. Any member-group $x..y$ denotes as set members all values in the range $x..y$. If the value of x is greater than the value of y , then $x..y$ denotes no members and $[x..y]$ denotes the empty set.

All expression values in the member-groups of a particular set-constructor must be of compatible ordinal-types. If a is the smallest ordinal-value in the resulting set, and if b is the largest ordinal-value in the resulting set, then the base-type of the resulting set is $a..b$.

Examples of set-constructors:

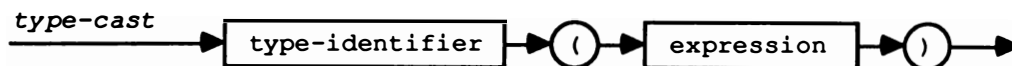
```

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

```

5.4 Type-Casts

Type casting (also called type coercion) provides a way to change the type of an expression to another type.



For ordinal and pointer types, the result of this operation is an expression of type **type-identifier** whose (ordinal) value is obtained by converting the original expression. This conversion may involve truncation or extension of the original value if the storage size of the expression is changed.

For non-ordinal types, the result of this operation is an expression of type **type-identifier** whose internal representation (i.e., the pattern of bits which comprise its value) is the same as the internal representation of the original expression. In particular, the storage size of a (non-ordinal) expression may not be changed by a type cast.

A value of reference-type can be coerced to another reference-type in the same domain.

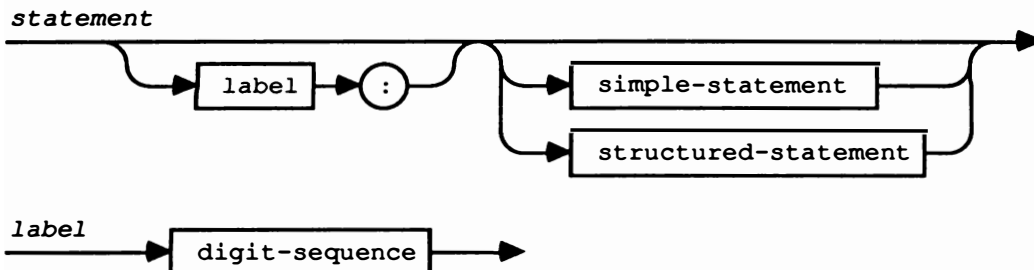
Examples of type-casts:

```

boolean (1)
ptr (longint(p)+1)
ptr (-1)
longint (@proc)
color (x)
point (0)
  
```

6.0 Statements

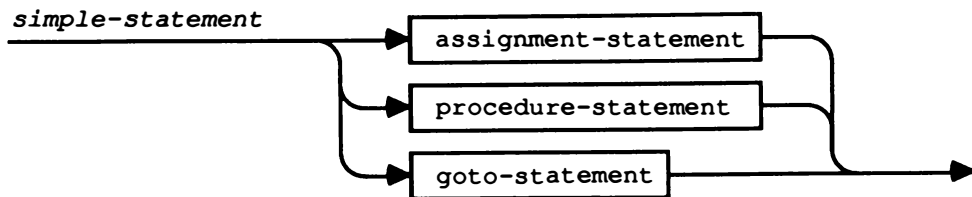
Statements denote algorithmic actions, and are executable. They can be prefixed by labels and a labeled statement can be referenced by a **goto**-statement.



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

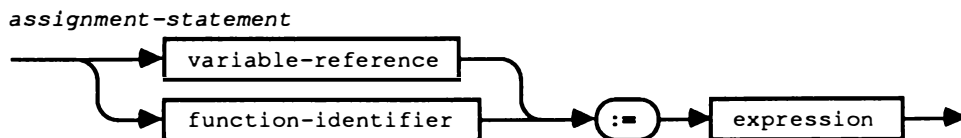
6.1 Simple-Statements

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



6.1.1 Assignment-Statements

The syntax for an **assignment-statement** is as follows:



The assignment-statement can be used in two ways:

- To replace the current value of a variable with a new value specified by an expression.
- To specify an expression whose value is to be returned by a function.

The expression must be assignment-compatible with the type of the variable or the result-type of the function as described in §3.5.3.

It is not specified whether the variable-reference is evaluated before or after the evaluation of the expression. However, once the reference is established, it is not altered by the remaining execution of the assignment-statement. Thus, the outcome of

```
a[x] := f(x)
```

depends on whether f modifies x and, if so, whether $f(x)$ is evaluated before or after $a[x]$.

Examples of assignment-statements:

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

6.1.2 Procedure-Statements

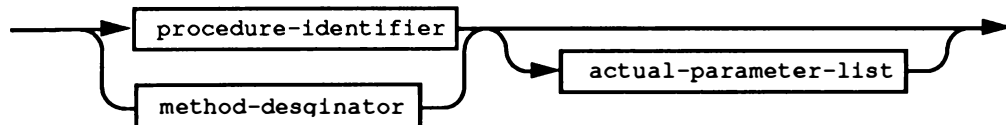
A **procedure-statement** specifies the activation of the procedure denoted by the procedure-identifier. If the corresponding procedure-declaration contains a list of formal-parameters, then the procedure-statement must contain a corresponding list of actual-parameters. Each actual-parameter is substituted for the corresponding formal-parameter as described in §7.3.

A **method-designator** activates the method specified by the method-identifier (§3.2.5) of an object-type referenced by the reference-variable. The actual method activated is specified by the run-time type of the object. When the method is activated, the object is passed as an implicit formal parameter called `self` (see §7.3.6) of the type corresponding to the object whose method was activated.

You can omit the reference variable and the "." within a **with** statement that lists the reference variable (see §4.3.2 and §6.2.4). You can also leave out the reference variable in a method block; in this case it is the same as having written `self.` before the field name.

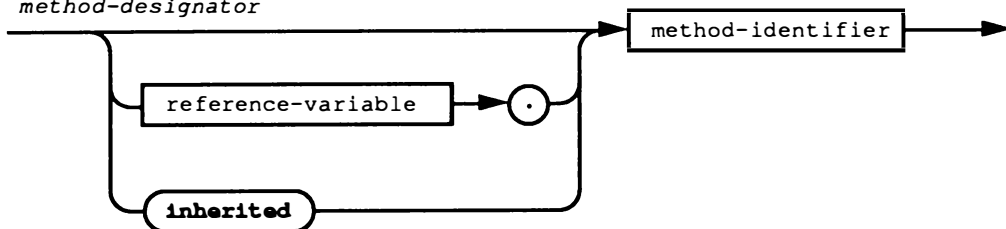
You'll usually use the **inherited** keyword when you override a method (see §3.2.5). It can only be used in a method block. When it appears before a method-identifier, it causes `self` to be the implicit actual parameter to the called procedure. The method activated is always the method inherited from the parent object; any overrides are ignored.

procedure statement



A **method-designator** activates the method specified by the method-identifier (§3.2.5) of an object. The run-time type of the object determines which method is activated.

method-designator



Note: The order in which actual parameters are evaluated and bound to their formal parameters is unspecified.

Examples of procedure-statements:

```
PrintHeading;
Transpose(a,n,m);
Bisect(fct,-1.0,+1.0,x)
```

6.1.3 Goto-Statements

A **goto-statement** causes the statement prefixed by the label that is referenced in the goto-statement to be the next statement executed.



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

A **goto**-statement G can **goto** a labeled statement S if and only if one of the following is true:

- S is a statement that contains G. For example:

```

1:  if ... then
2:      begin

3:          begin
                ...
                goto {1, 2 and 3 are legal, 4 is not}
                ...
            end
        end
    else
4:      begin
                ...
            end
  
```

- S is a statement of a statement-list that contains G. For example:

```

begin
    ...
1:    ...;
    ...
    begin
        ...
2:    ...;
        ...
        begin
            ...
            goto {1, 2, 3, and 4 are legal, 5 is not}
            ...
        end
    end
3:    ...;
    ...
end

4:    ...;
    ...
end;
begin
5:    ...
end
  
```


- S is a statement in a block that contains the block containing G, provided that S is a statement of the outermost statement-list of its block. For example:

```

program a (...);
    procedure b;
        procedure c;
            begin
                goto { 2 and 4 are legal; 1 and 3 are not }
            end;

        procedure d;
            begin
1:         ...
            end;

        begin { b }
2:         ...;
        begin
3:         ...
            end
        end;

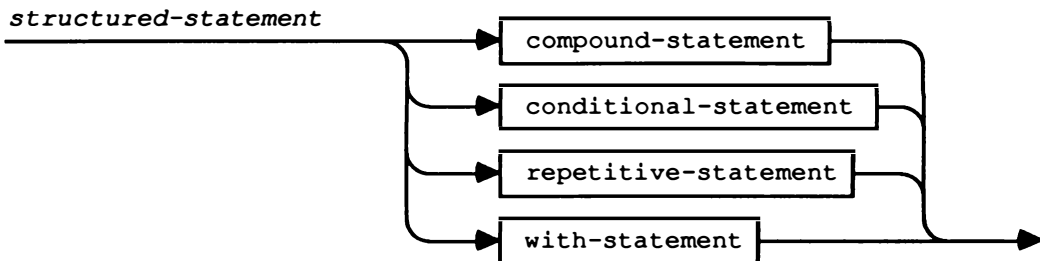
    begin { main }
4:     ...
    end.

```

When the destination of a **goto** is in a block b that does not contain the **goto**, every block activation (see §2.3) that has occurred since the most recent activation of b is terminated.

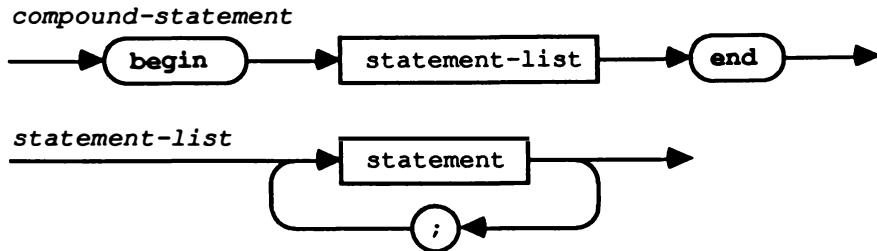
6.2 Structured-Statements

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



6.2.1 Compound-Statements

The **compound-statement** specifies that its component statements are to be executed in the same sequence as they are written. The semicolon is a **statement separator**, not terminator.



Example of compound-statement:

```

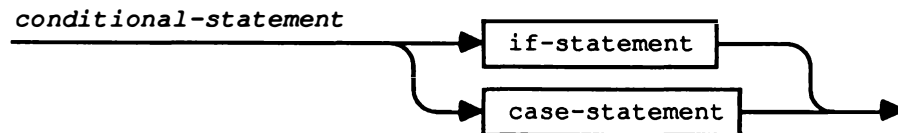
begin
    z := x;
    x := y;
    y := z
end

```

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

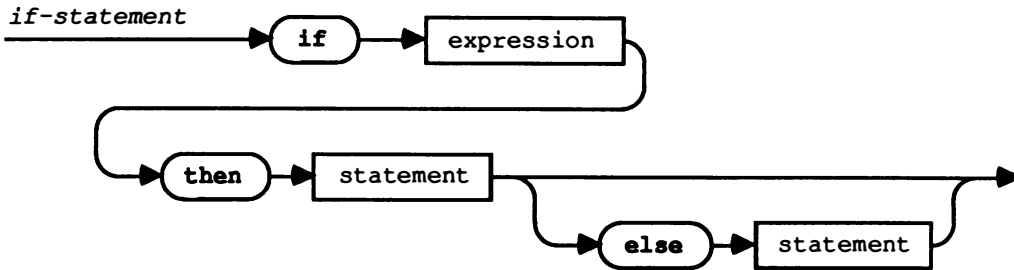
6.2.2 Conditional-Statements

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



6.2.2.1 If-Statements

The syntax for **if-statements** is:



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

If the expression yields `false`, and the **else** part is present, the statement following the **else** is executed; if the **else** part is not present, then execution proceeds with the next statement following the **if** statement.

The syntactic ambiguity arising from something like:

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

is resolved by interpreting it as being equivalent to:

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

rather than:

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

In other words, an **else** is always associated with the closest **if** that is not already associated with an **else**.

Examples of if-statements:

```

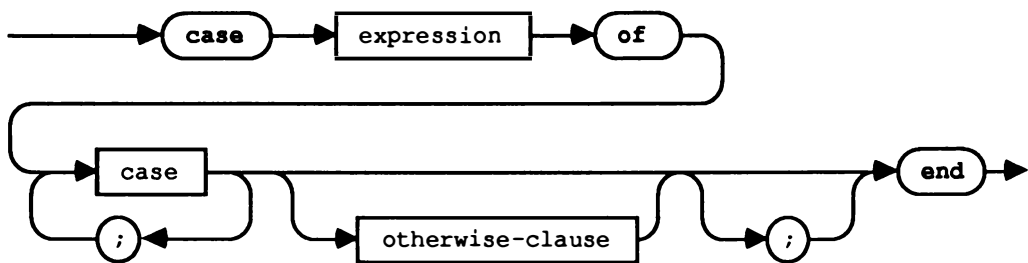
if x < 1.5 then
  z := x+y
else
  z := 1.5

if p1 <> nil then
  p1 := p1^.father
  
```

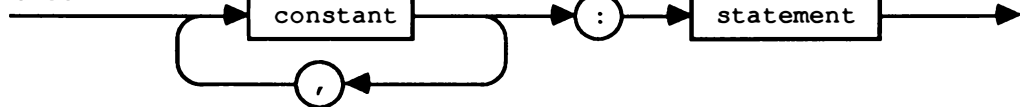
6.2.2.2 Case-Statements

The **case-statement** contains an expression (the **selector**) and a list of statements. Each statement must be prefixed with one or more constants (called **case-constants**), or with the reserved word **otherwise**. All the case-constants must be distinct and must be of an ordinal-type that is compatible with the type of the selector. A case-constant can be of type longint.

case-statement



case



otherwise-clause



The case-statement specifies execution of the statement prefixed by a case-constant equal to the current value of the selector. If no such case-constant exists and an **otherwise** part is present, the statement following the word **otherwise** is executed; if no **otherwise** part is present, it is an error.

Examples of case-statements:

```

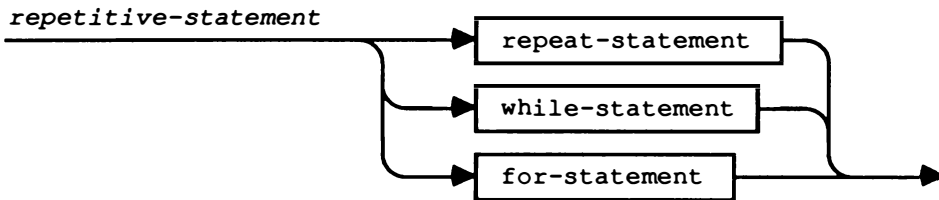
case operator of
  plus:  x := x+y;
  minus: x := x-y;
  times: x := x*y
end

case i of
  1: x := sin(x);
  2,8..12: x := cos(x);
  3..7: x := exp(x);
otherwise
  x := ln(x)
end

```

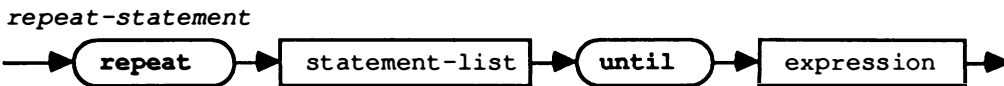
6.2.3 Repetitive-Statements

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



6.2.3.1 Repeat-Statements

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



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

Examples of repeat-statements:

```

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

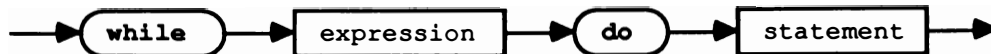
```

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

6.2.3.2 While-Statements

A **while-statement** contains an expression that controls the repeated execution of a statement (which may be a compound-statement).

while-statement



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

The while-statement:

```
while b do
    body
```

is equivalent to:

```
if b then
    repeat
        body
    until not b
```

Examples of while-statements:

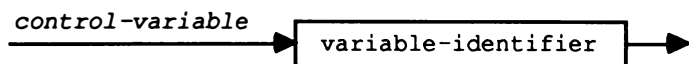
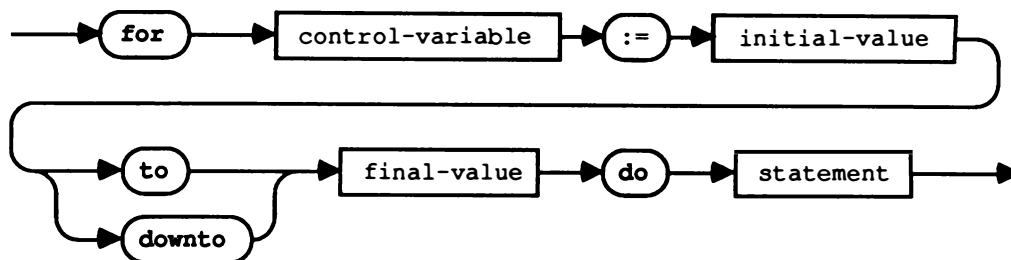
```
while a[i] <> x do
    i := i+1

while i>0 do
    begin
        if odd(i) then
            z := z*x;
        i := i div 2;
        x := sqr(x)
    end

while not eof(f) do
    begin
        process(f^);
        get(f)
    end
```

6.2.3.3 For-Statements

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



The control-variable must be a variable-identifier (without any qualifier) denoting a variable that is declared to be local to the block containing the for-statement. The control-variable must be of an ordinal-type, and the initial and final values must be of a type assignment-compatible with this type.

On entering a for-statement, the initial-value and the final-value are determined once (and only once) for the remainder of the execution of the for-statement.

Loosely speaking, the statement contained by the for-statement is executed once for every value in the range `initial-value..final-value`. With a for-statement using **to**, the control-variable has the value `initial-value` the first time, `succ(initial-value)` the second time, and so on. If the initial-value is greater than the final-value, then the contained statement is not executed. With a for-statement using **downto**, the control-variable has the value `initial-value` the first time, `pred(initial-value)` the second time, and so on. If the initial-value is less than the final-value, then the contained statement is not executed.

It is an error if the value of the control-variable is altered by execution of the contained statement. After a for-statement is executed, the value of the control-variable is undefined, unless the execution of the for-statement was terminated by a **goto** out of the for-statement.

Apart from these restrictions, the for-statement:

```

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

is equivalent to:

```

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

and the for-statement:

```

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

is equivalent to:

```

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

where temp1 and temp2 are auxiliary variables of the host-type of the variable v that do not occur elsewhere in the program; they are used to resolve the expressions e1 and e2 upon entering the statement's body.

Examples of for-statements:

```

for i := 2 to 63 do
  if a[i] > max then
    max := a[i]

for i := 1 to n do
  for j := 1 to n do
    begin
      x := 0;
      for k := 1 to n do
        x := x + m1[i,k]*m2[k,j];
      m[i,j] := x
    end

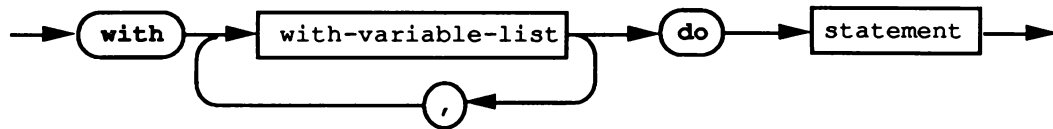
for c := red to blue do
  q(c)

```

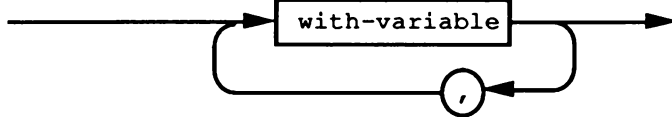
6.2.4 With-Statements

The syntax for a **with-statement** is

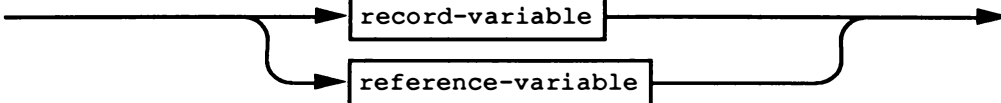
with-statement



with-variable-list



with-variable



The occurrence of a record-variable-reference or a reference-variable in a with-statement affects the way the compiler processes variable-references within the statement following the word **do**. Within a **with-statement**, fields of the record-variable or the reference-variable can be referenced directly by their field-identifiers, without making explicit reference to the record-variable. The same is true for methods of reference-variables.

Example of a with-statement:

```

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

```

This is equivalent to:

```

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

```

Within a with-statement, each variable-reference is checked to see if it can be interpreted as a field of the record. If so, it is always interpreted as such, even if a variable with the same name is accessible also. For instance, suppose that we have the following declarations:

```

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

var
  bar: recTyp;
  foo: integer;

```

The identifier `foo` can refer both to a field of the record variable `bar` and to a variable of type `integer`. In the statement

```

with bar do
  begin
    ...
    foo := 36;
    bar := 2.5;
    ...
  end

```

the `bar` between the `with` and the `do` is a reference to the variable `bar`, but `foo` is a reference to the field `bar.foo`, not the variable `foo`. Likewise, the reference to `bar` within this with-statement refers to `bar.bar`.

The statement:

```
with v1, v2, ... vn do
  s
```

is equivalent to:

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

Thus, if v_n in the above statements is a field of both v_1 and v_2 , it is interpreted to mean $v_2.v_n$ and not $v_1.v_n$.

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

7.0 Procedures and Functions

This section describes how to declare procedures and function in Pascal. It also describes how Pascal interprets parameters passed to these routines.

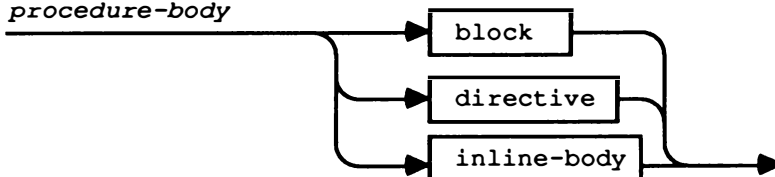
7.1 Procedure-Declarations

A **procedure-declaration** associates an identifier with a block as a procedure so that it can be activated by a procedure-statement (see §6.1.2).

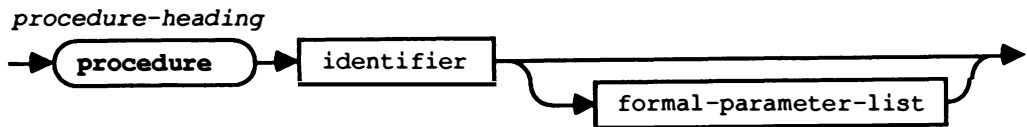
procedure-declaration



procedure-body



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



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

A procedure is activated by a procedure-statement (see §6.1.2), which gives the procedure's identifier and any actual-parameters required by the procedure. The statements to be executed upon activation of the procedure are specified by the statement-part of the procedure's block. If the procedure's identifier is used in a procedure-statement within the procedure's block, the procedure is executed recursively (see §2.3).

Example of a procedure-declaration:

```

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

7.1.1 Forward-Declarations

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

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

Example of a forward declaration:

```

procedure walter(m,n : integer);
forward;

procedure clara(x, y : real);
begin
    ...
    walter(4, 5);
    ...
end;

procedure walter;
begin
    ...
    clara(8.3, 2.4);
    ...
end;

```

7.1.2 External-Declarations

A procedure-declaration that has the directive **external** instead of a block is called an **external declaration**. External procedures and functions are used to declare the Pascal interface to a separately compiled or assembled routine. The external code must be linked with the rest of the program before execution.

Example of an external-declaration:

```

procedure DoInits(num:integer);
external;

```

This means that `DoInits` is an external procedure that will be linked with the rest of the program before execution.

Note: It is the programmer's responsibility to insure that the external procedure is compatible with the external declaration in the Pascal program.

A unit (see §8.3) may declare a procedure in the interface-part and then implement this procedure by an external declaration. For example,

```

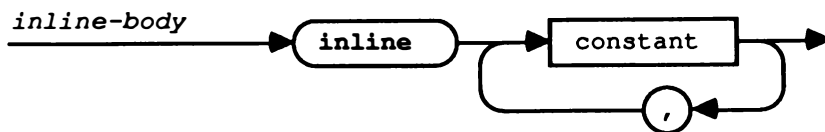
unit a;
interface
    procedure foo (arg:integer);
implementation
    procedure foo;
        external;
end.

```

This description of external procedures also applies to external functions.

7.1.3 Inline-Declarations

A procedure-declaration that has the word-symbol **inline** followed by one or more integer constants instead of a block is called an **inline declaration**. Inline procedures and functions are used to embed machine code in a Pascal program.



When a procedure is normally called, code is generated that reserves one or two words of function result (if a function is being called), and pushes the procedure's arguments (if any). Then a JSR instruction is generated. By declaring a routine as **inline**, the compiler will cause the constants that follow the word-symbol **inline** to be generated in place of the JSR instruction. Each constant represents one word (16 bits) and is generated in the order given.

Example of an inline-declaration:

```

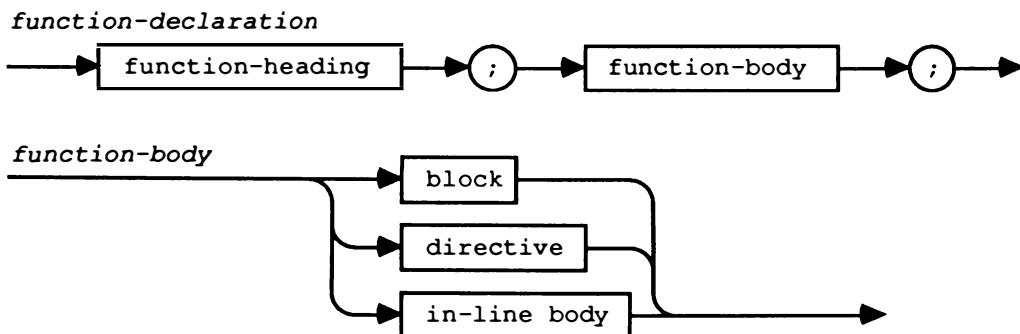
procedure trap (tos : longint);
inline $A9ED;
    
```

- It is the programmer's responsibility to observe the proper rules for adjusting the stack, saving registers, etc.
- An inline procedure declared in a unit's interface section has no corresponding declaration in the implementation section.
- A forward declaration or interface procedure declaration may not be later defined as an inline declaration.

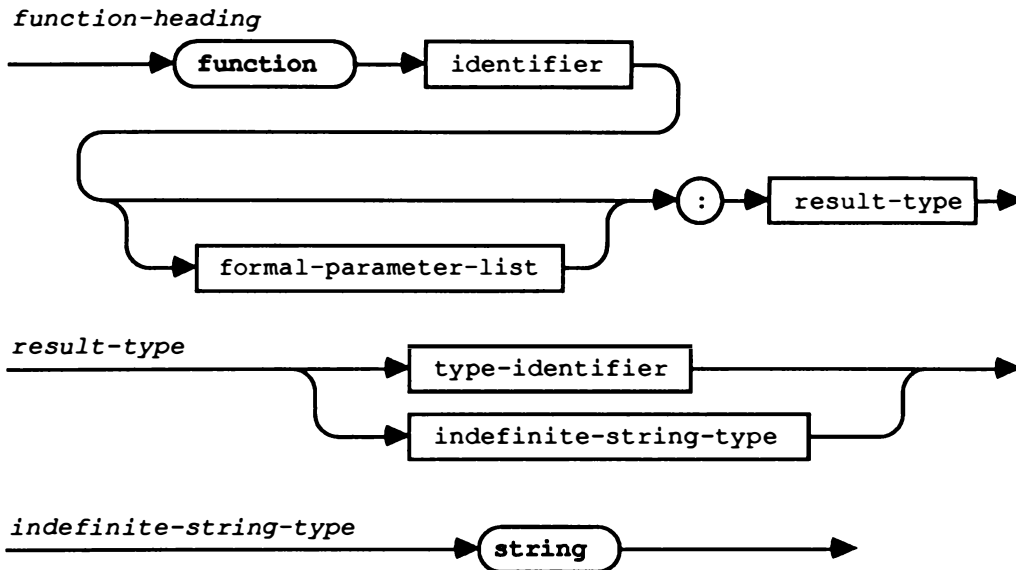
This description of inline procedures also applies to inline functions.

7.2 Function-Declarations

A **function-declaration** serves to declare a part of the program that computes and returns a value of some type.



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



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

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

The statements to be executed upon activation of the function are specified by the statement-part of the function's block. This block should normally contain at least one assignment-statement (see §6.1.1) that assigns a value to the function-identifier. The result of the function is the last value assigned. If no such assignment-statement exists, or if it exists but is not executed, the value returned by the function is *undefined*, which is an error.

If the function's identifier appears as an operand in an expression within the function's block, the function is executed recursively.

Examples of function-declarations:

```

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

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

```

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

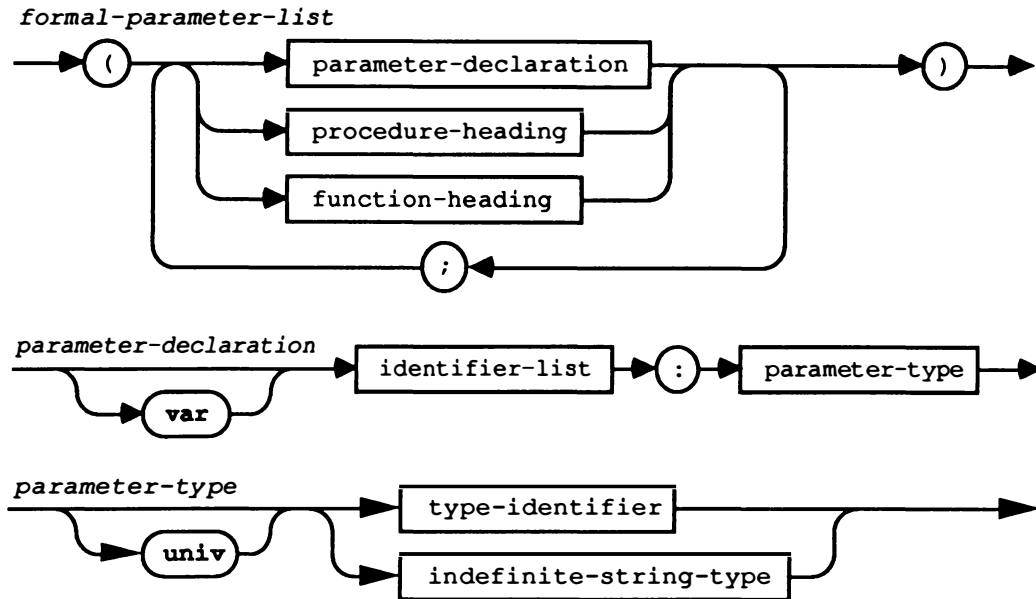
7.3 Parameters

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

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

If it is part of the declaration of a procedural or functional parameter, it declares the formal parameters of the procedural or functional parameter. In this case there is no associated block and the

identifiers of parameters in the formal-parameter-list are not significant (see §7.3.3 and §7.3.4 below).



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

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

The type of a formal-parameter is denoted by either a type-identifier or the word-symbol **string**. Thus, to use a type such as **array**[0..255] **of** **char** as the type of a parameter, you must declare a type-identifier for this type:

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

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

The interpretation of **string** as a parameter-type is described below.

The **univ** qualifier lets you disable type-checking for a routine's parameter. For more information see §7.3.7 below.

7.3.1 Value Parameters

For a **value parameter**, the corresponding actual-parameter in a procedure-statement or function-call (see Sections 5.2 and 6.1.2) must be an expression, and its value must not be of file-type or of any structured-type that contains a file-type. The formal value parameter denotes a variable local to the procedure or function. The current value of the expression is assigned to the formal value parameter upon activation of the procedure or function. The actual-parameter must be assignment-compatible with the type of the formal value parameter (see §3.5.3). If the parameter-type is **string**, then the formal parameter is given a size attribute of 255.

7.3.2 Variable Parameters

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

Within the procedure or function, any reference to the formal variable parameter is a reference to the actual-parameter itself. The type of the actual-parameter must be *identical* to that of the formal variable parameter. However, if the parameter-type is **string**, then any string-type is considered identical to it; the size attribute of the formal parameter is always the size attribute of the actual parameter.

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

Components of variables of any packed structured-type cannot be used as actual variable parameters.

7.3.3 Procedural Parameters

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

Example of procedural parameters:

```

program PassProc;
  var
    i: integer;
  procedure a(procedure x);
    begin
      write('About to call x ');
      x
    end;

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

```

```

function c(procedure x) : integer;
  begin
    x;
    c:=2
  end;

begin {PassProc}
  a(b);
  i:= c(b)
end.

```

If the formal procedure has a formal-parameter-list, then the actual procedure's declaration must also have a formal-parameter-list and both must be compatible (see §7.3.5). However, only the identifier of the actual procedure is written as an actual parameter; no formal or actual parameter-list is given.

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

```

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

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

begin {test}
  CallProc(xAsPar)
end.

```

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

To see what this means, consider the following program taken from an example in the ANSI Pascal Standard (which is in turn taken from an early version of the Pascal Validation Suite):

```

program t6p6p3p4 (output);
  var
    GlobalOne, GlobalTwo : integer;

  procedure dummy;
  begin
    writeln('fail4')
  end;

```

```

procedure p(procedure f(procedure ff; procedure gg);
             procedure g);
var
    LocalToP : integer;

    procedure r;
    begin
        if GlobalOne = 1 then
            begin
                if (GlobalTwo <> 2) or (LocalToP <> 1) then
                    writeln('fail1')
                end
            else if GlobalOne = 2 then
                begin
                    if (GlobalTwo <> 2) or (LocalToP <> 2) then
                        writeln('fail2')
                    else
                        writeln('pass')
                    end
                else
                    writeln('fail3');
                GlobalOne := GlobalOne + 1
            end;

    begin {p}
        GlobalTwo := GlobalTwo + 1;
        LocalToP := GlobalTwo;
        if GlobalTwo = 1 then
            p(f,r)
        else
            f(g,r)
        end;

    procedure q(procedure f; procedure g);
    begin
        f;
        g
    end;

begin {program}
    GlobalOne := 1;
    GlobalTwo := 0;
    p(q, dummy)
end.
  
```

An explanation might make things clearer:

1. At the call to `p` in the main program, `GlobalOne=1` and `GlobalTwo=0`.
2. Within `p`, the formal parameter `f` corresponds to the actual procedure `q` and the formal `g` corresponds to the actual dummy. The values of `GlobalTwo` and `LocalToP` both become 1. Because `GlobalTwo=1`, `p` calls itself recursively.
3. Within this second activation of `p`, the formal `f` corresponds to the formal `f` of the first activation, which corresponds to the actual `q`. The formal `g` corresponds to the actual `r`. The values of `GlobalTwo` and `LocalToP` now become 2. Because `GlobalTwo<>1`, this second activation of `p` now calls its formal parameter `f`, which is the actual procedure `q`.
4. Within `q`, its formal parameter `f` corresponds to the actual procedure `r` and the formal `g` also corresponds to the actual `r`. Procedure `q` now calls its formals `f` and `g`, i.e. `r` and `r`, and the program terminates after all the activations unwind.

It's what happens during the two calls to procedure `r` within procedure `q` that is critical. If this program runs correctly, it will print 'pass'. For this to happen, the first call to `r` will occur while `GlobalOne=1` and will expect `LocalToP` to be 1; the second call to `r` will occur while `GlobalOne=2` and will expect `LocalToP` to be 2. Since there are no assignments to `LocalToP` within `r` or `q`, how can this be?

`LocalToP` is not simply local to the procedure `p`, it is local to each activation of `p`. Since there are two activations of `p`, and since `r` is passed as a parameter in each activation of `p`, each `r` accesses the variable `LocalToP` that is local to the activation in which it is passed. Since, in the first activation of `p`, the value of `LocalToP` is 1, that is the value the first execution of `r` sees when it accesses `LocalToP`. Since, in the second activation of `p`, the value of `LocalToP` is 2, that is the value the second execution of `r` sees.

Predefined, inline, and Toolbox procedures can not be passed as procedural parameters.

7.3.4 Functional Parameters

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

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

7.3.5 Parameter List Compatibility

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

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

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

7.3.6 Implicit parameters

In the declaration of a method for an object-type, there is an implicit parameter called `self`. The type of `self` is the type of the object. `Self`'s scope extends over the method declaration. The value of `self` is assigned when the object variable is created. The value is a reference to the object whose method component was designated to activate the method. The value of `self` is assigned only when the object is created. No subsequent assignment is possible.

7.3.7 Univ parameters

The **univ** qualifier lets you disable type-checking for a routine's parameter. When a formal parameter's type is qualified with **univ**, the actual parameter's type does not have to be compatible with it, but the two types must be the same size.

For example, this function can initialize a variable of type `Point` to zero, since a `Point` is the same size as a `Longint`:

```
procedure ZeroOut (var l: univ Longint);
begin
    l := 0;
end;
```

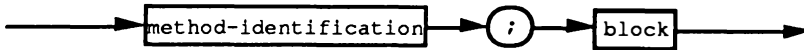
This initializes the point `p` to zero:

```
ZeroOut (p);
```

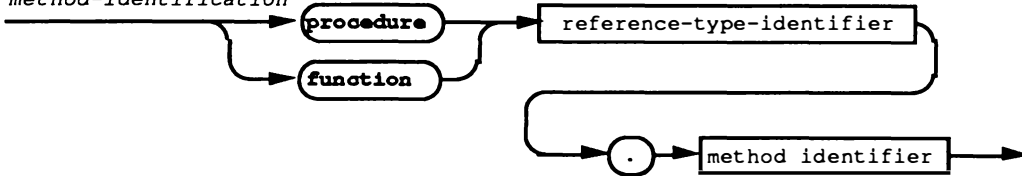
7.4 Method declarations

A method declaration describes the implementation of an object's methods. Methods are declared like forward declarations of procedures and functions. The heading appears in the declaration of the object (see §3.2.5). The body of the method should appear in the same unit as the declaration of the object.

method-declaration



method-identification



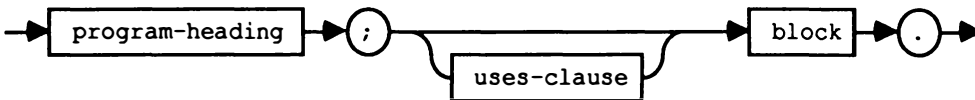
8.0 Programs and Units

This section describes how to write the main program and how to declare units in THINK Pascal.

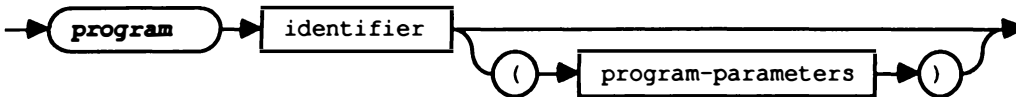
8.1 Program Syntax

A THINK Pascal program has the form of a procedure declaration except for its heading.

program



program-heading



program-parameters



uses-clause



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

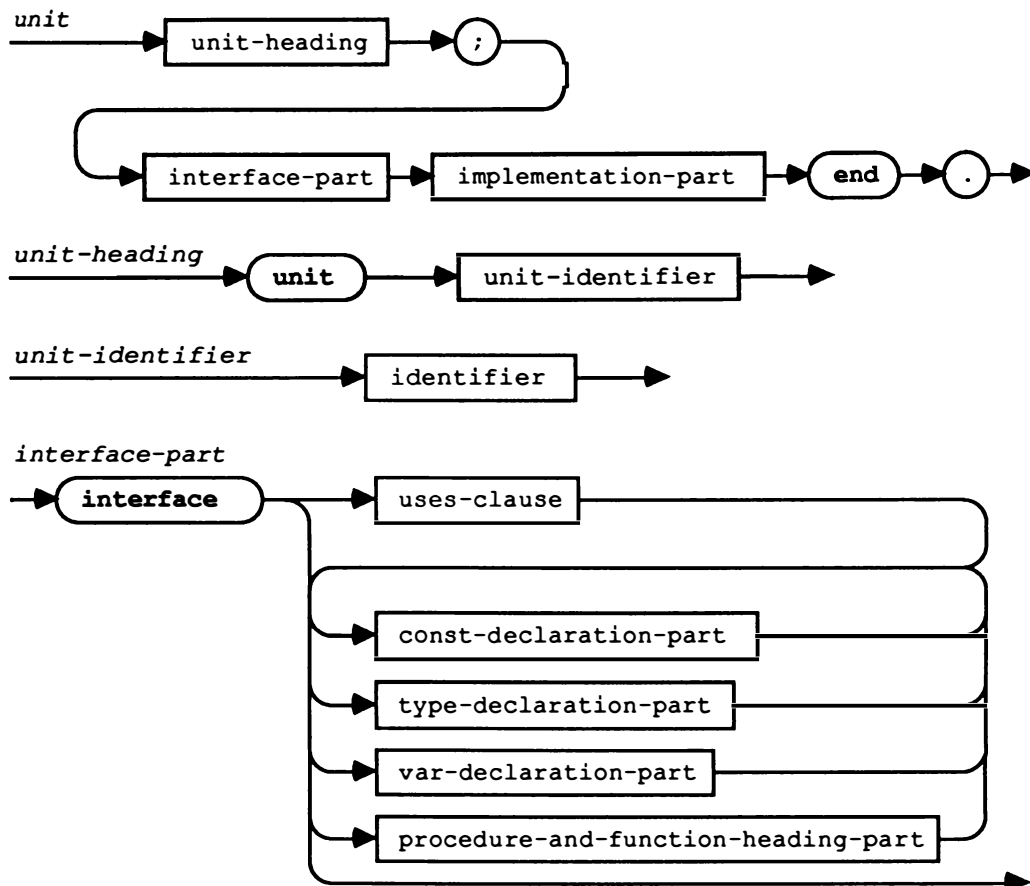
The uses-clause identifies all units required by the program.

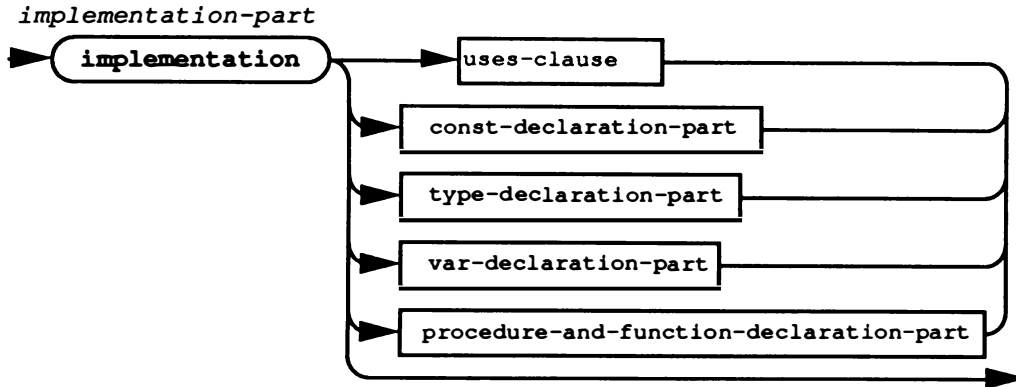
8.2 Program-Parameters

Only the predefined identifiers `input` and `output` are allowed as program-parameters.

8.3 Unit Syntax

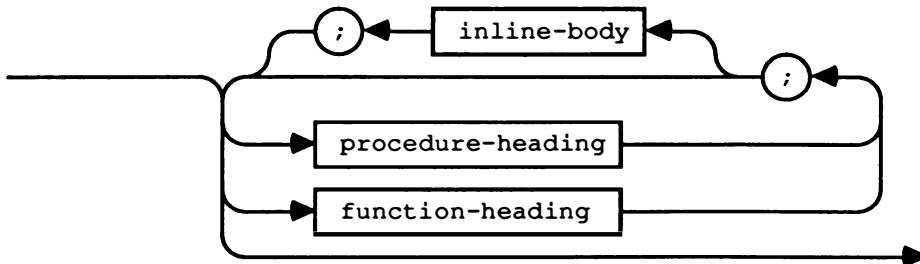
Units provide the means to organize a Pascal program into logically related parts for modular construction of programs and libraries.





The identifier immediately after the word **unit** is the unit's identifier. The **uses-clause** identifies all units required by the unit.

procedure-and-function-heading-part



The procedures and functions declared in the **interface-part** must be redeclared in the **implementation-part**. The parameters and function types of these redeclarations may be omitted, since they were declared in the interface-part. The procedure and function blocks for these routines are included in the implementation-part since they were omitted in the interface-part.

Note: The parameters and functions types can be redeclared in the implementation part only if they are identical to the declarations in the interface part.

The scope of the declaration for an interface-part is also the implementation-part which is associated with that interface part.

8.4 Uses-Clause

The **uses-clause** controls which units are available to the host (other units or the main program). Each identifier in the identifier-list of the uses-clause is the name of a unit to be made available to the host. All declared entities in the used unit appear as though they are declared in the interface part or the main program block which contains the uses-clause.

The uses-clause behaves differently depending on whether the "USES Extensions" option in **Compiler Options...** dialog is on or off.

If the "USES Extensions" option is on, THINK Pascal propagates uses-clauses and lets you put uses-clauses in the implementation-part. If the option is off, THINK Pascal treats uses-clauses as it did in earlier versions.

8.5 Unit Dependencies

In order to satisfy the requirements of §2.2.3, a unit must precede any interface-part or program that it supplies (see §2.2.6). It is therefore not possible to construct a valid program in which two units supply each other.

If the "USES Extensions" option is off, the uses clause in the host must name all units used (directly or indirectly) by the host. Consider the following example:

program Host;	unit UnitA	unit UnitB;
uses	interface	interface
UnitA;	uses	const
begin	UnitB	b = 3;
...	implementation	implementation
end.	const	...
	a = b;	end.
	end.	

The program Host uses UnitA. UnitA uses UnitB. There is an identifier b defined as a constant in the interface of UnitB, but the only reference to b is in the **implementation** part of UnitA. In this case, it is not necessary to name UnitB in the uses-clause of Host.

In the following example, Host needs to use symbols from both UnitA and UnitB, so both names are included in the uses-clause, even though UnitA already includes UnitB.

program Host;	unit UnitA	unit UnitB;
uses	interface	interface
UnitA, UnitB;	uses	const
begin	UnitB	b = 3;
v := b;	const	implementation
w := a;	a = b;	...
end.	implementation	end.
	end.	

Now consider the following example:

```

program Host;
  uses
    UnitB, UnitA;
begin
  ...
end.

unit UnitA;
interface
  uses
    UnitB;
  const
    a = b;
implementation
  ...
end

unit UnitB;
interface
  const
    b = 3;
implementation
  ...
end.

```

This example is like the previous one, except that this time the reference to the identifier `b` is in the **interface** part of `UnitA`. In this case, there is an indirect reference to `UnitB` and it is necessary to name `UnitB` in the **uses**-clause of `Host`. Note that `UnitB` must be named before `UnitA`.

In the first example, if the "USES Extensions" option is turned on, you can place the **uses** `UnitB` statement in the **implementation** part of `UnitA` because the symbol from `UnitB` is only used in the **implementation** of `UnitA`:

```

program Host;
  uses
    UnitA;
begin
  ...
end.

unit UnitA;
interface
implementation
  uses
    UnitB
  const
    a = b;
end.

unit UnitB;
interface
  const
    b = 3;
implementation
  ...
end.

```

If the `Host` needs symbols from `UnitA` and `UnitB`, and the "USES Extensions" option is on, you only need to write **uses** `UnitA` because THINK Pascal propagates the **uses**-clause:

```

program Host;
  uses
    UnitA;
begin
  ...
end.

unit UnitA;
interface
  uses
    UnitB;
  const
    a = b;
implementation
  ...
end

unit UnitB;
interface
  const
    b = 3;
implementation
  ...
end.

```

For more information and examples, see Chapter 10, "Units and Libraries."

9.0 Input/Output

This section describes the standard built-in I/O procedures and functions of THINK Pascal.

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

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

Also, the predeclared file variables `input` and `output` do not act as though they are declared in a block outside the program. See §9.4.

This section and §10 use a modified BNF notation, instead of syntax diagrams, to indicate the syntax of actual-parameter-lists for standard procedures and functions.

Example:

```
write(f, e1 [ , e2, ..., en ])
```

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

- `f`, `e1`, `e2`, and `en` stand for actual-parameters. Notes on the types and interpretations of the parameters accompany the syntax description.
- The notation `e1, e2, ..., en` means that any number of actual-parameters can appear here, separated by commas.
- Square brackets `[]` indicate parts of the syntax that can be omitted. They do not indicate sets.

Thus the syntax shown here means that the `f` parameter is required. Any number of `e` parameters may appear, with separating commas, and there must be at least one `e` parameter.

9.1 Introduction to I/O

A Pascal file variable is any variable whose type is a **file-type**. There are two classes of files: **textfiles** and **non-textfiles**. Any file variable declared to be a type identical (see §3.5.1) to the standard type `text` is a textfile and all others are non-textfiles. The standard type `text` is roughly equivalent to the type **packed file of char** in that a file of type `text` may be treated as though it were a **packed file of char**. However, the semantics of the two differ somewhat and, in particular, there are certain standard procedures and functions that may be applied to textfiles but not to files of type **packed file of char**.

A file variable may (but need not) be associated with an external file. The external file may be a named collection of information stored on a peripheral device or, depending on the device, it may be the peripheral device itself. If a file variable is not associated with an external file or device, it is referred to as an **anonymous file**.

For a file variable to be used it must be **opened**. An existing file may be opened with the `reset` and `open` procedures, and a new file may be created and opened with the `rewrite` and `open` procedures. Files opened with `reset` are **read-only** and files opened with `rewrite` are **write-only**. However, `reset` may be applied to a file opened with `rewrite`, which causes the file to

become read-only. Likewise, `rewrite` may be applied to a file opened with `reset`, which causes the file to become write-only.

Files opened with `open` are **read/write** files, i.e. they allow both reading and writing. `Rewrite` and `reset` may be applied to files opened with `open`. `Rewrite` leaves the files as read/write files. `Reset` causes them to become read-only.

The standard file variables `input` and `output`, if present in the program parameter list, are opened automatically when program execution begins and should not be opened again with `reset` or `rewrite`. `Input` is a read-only file associated with your keyboard and `output` is a write-only file associated with the text window.

A file is a linear sequence of **components**, each of which has the component-type of the file. Each component has a **component-number** that is its position in the file relative to the first component in the file. The first component of a file is considered to be component zero.

At any point in time, there is only one component of a file that may be accessed directly through the **file-buffer** denoted by f^{\wedge} . The **current file position** of f is the component number of the component currently accessible through f^{\wedge} . Whenever a file is opened, the current file position is set to component zero, i.e. to the beginning of the file.

Under certain conditions, such as when the current file position is at the end of the file, the value of f^{\wedge} is said to be undefined. It is an error to attempt to use the value of f^{\wedge} when the value is undefined. Assignment to f^{\wedge} is, however, still possible.

Note: It is an error to cause the current file position of a file f to be altered while a reference to the file-buffer f^{\wedge} exists.

Files are normally accessed **sequentially**. That is, when an I/O operation is completed on a file component, the current file position moves to the numerically next file component. Files opened with `open`, however, may also be accessed **randomly** with the standard procedure `seek`, which may be used to specify that the current file position is to be moved to any component number in the file. The function `filepos(f)` may be applied to any file variable f and returns the component number of the current file position.

9.2 Standard Procedures and Functions for All Files

The procedures and functions described in this section work on files of any file-type.

9.2.1 The Reset Procedure

Opens an existing file for sequential, read-only access or rewinds an open file.

`reset (f [, title])`

f a variable-reference that refers to a variable of file-type. If a **title** is given, the file must not be open. If a **title** is not given, the file must be open.

title an optional expression with a string value.
The string should be a valid name for a file on a file-structured device, or a name for a non-file-structured device.

`Reset(f)` when **f** is already open causes **f** to be "rewound", i.e. the current file position for **f** is reset to the beginning of the file. If **f** was originally opened with `rewrite`, **f** becomes read-only.

`Reset(f, title)` finds an existing external file with the name **title**, and associates **f** with this external file. It is an error if there is no existing external file with that name.

The following conditions always hold after `reset(f, [title])` is executed:

- `Eof(f)` is `true` if the file is empty. Otherwise, `eof(f)` is `false`.
- The current file position is the first component of the file (component zero) and the file buffer variable `f^` contains the value of that component unless `eof(f)` is `true`, in which case the value of `f^` is undefined.

Note: You can use the built-in function `OldFileName` to get the name of an existing text file. See §10.9.6.

9.2.2 The Rewrite Procedure

Creates and opens a new empty file for sequential, write-only access, or rewinds and erases an open file.

`rewrite(f [, title])`

f a variable-reference that refers to a variable of file-type. If a **title** is given, the file must not be already open.

title an optional expression with a string value. The string should be a valid name for a file on a file-structured device, or a name for a non-file-structured device.

`Rewrite(f)` (with no **title**) when **f** is not yet open creates an empty anonymous file for writing to.

`Rewrite(f)` when **f** is already open causes **f** to be "rewound", i.e. the current file position for **f** is reset to the beginning of the file and any prior contents of **f** are deleted. If **f** was originally opened with `reset`, **f** becomes write-only.

`Rewrite(f, title)` creates a new external file with the name `title`, and associates `f` with this external file. If an external file with the name `title` already exists, it is deleted and a new empty file with the same name is created in its place.

The following conditions always hold after `rewrite(f, [title])` is executed:

- `Eof(f)` is true.
- The current file position is component zero, i.e. the first component written to the file will become the first component of the file. The value of `f^` is undefined.

Note: You can use the built-in function `NewFileName` to prompt the user for the name a text file. See §10.9.7.

9.2.3 The Open Procedure

Opens an existing file or creates and opens a new file for random, read/write access.

`open(f, title)`

- | | |
|--------------------|--|
| <code>f</code> | a variable-reference that refers to a variable of file-type. <code>f</code> must not be already open. |
| <code>title</code> | an expression with a string value. The string should be a valid name for a file on a file-structured device, or a name for a non-file-structured device. |

`Open(f, title)` opens an existing external file with the name `title`, and associates `f` with this external file. If an external file with the name `title` does not already exist, a new empty file is created. The file is opened for both reading and writing.

The following conditions always hold after `open(f, title)` is executed:

- `Eof(f)` is true if the file is empty. Otherwise, `eof(f)` is false.
- The current file position is component zero and the file buffer variable `f^` contains the value of that component (unless `eof(f)` is true).

9.2.4 The Close Procedure

Closes a file.

`close(f)`

- | | |
|----------------|---|
| <code>f</code> | a variable-reference that refers to a variable of file-type. <code>f</code> must be open and must not be an anonymous file. |
|----------------|---|

`Close(f)` closes `f`, i.e. the association between `f` and its external file is broken and the file system marks the external file closed. All subsequent references to `f` are invalid (except to open it again). In particular, the value of `f^` becomes undefined.

If a procedure or function block activation that has a file variable `f` local to it is exited and `f` is not already closed, `f` is closed automatically. If a dynamic variable created with `new` is, or contains, a file variable `f` that is still open when the dynamic variable is destroyed with `dispose`, `f` is closed automatically. If a program terminates with any file still open, the file is automatically closed.

9.2.5 The Eof Function

Detects the end of a file.

`eof [(f)] : boolean`

`f` a variable-reference that refers to a variable of file-type. If `f` is omitted, the function is applied to the standard file variable `input`. The file must be open.

Returns `boolean`

`Eof (f)` returns `true` if the current file position is beyond the last component of the file, or if the file contains no components; otherwise, `eof (f)` returns `false`. Specifically, this means the following:

- After a `get`, `eof (f)` returns `true` if the previous file position was the last component of the file.
- After a `put`, `eof (f)` returns `true` if the component written by the `put` is now the last file component.

It is always an error to do a `get (f)` if `eof (f)` is `true`. If `f` is write-only, `eof (f)` will always be `true`.

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

For some devices, `eof` may never be `true`.

9.2.6 The Get Procedure

Advances the current file position and reads the next component of a file.

`get (f)`

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

`Get (f)` advances the current file position to the next file component, and assigns the value of this component to `f^`. If no next component exists, then `eof (f)` becomes `true`, and the value of `f^` becomes undefined.

9.2.7 The Put Procedure

Writes the file buffer to the current file position.

`put (f)`

f a variable-reference that refers to a variable of file-type. The file must be open and the value of `f^` must not be undefined.

`Put (f)` writes the value of `f^` to `f` at the current file position and advances the current file position to the next file component. If the new file position is beyond the end of the file, `eof (f)` becomes `true`, and the value of `f^` becomes undefined.

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

9.2.8 The Seek Procedure

Allows access to an arbitrary file component.

`seek (f, n)`

f a variable-reference that refers to a variable of file-type. The file must be open, and it must have been opened with `open`.

n an expression with an integer-type value that specifies a file component number in the file. Components in files are numbered from zero.

`Seek (f, n)` causes the file component numbered `n` to become the current file position. The value of `f^` becomes the value of that component unless `n` is greater than the number of the last component of the file, in which case `eof (f)` becomes `true` and the value of `f^` is undefined. Thus, `seek (f, maxlongint)` always sets the current file position to the end of file. `Seek` of a device, such as `Printer:` or `Modem:` is not allowed.

9.2.9 The Filepos Function

Returns the component number of the current file position.

`filepos (f)`

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

Returns `longint`

`Filepos (f)` returns a `longint` value that is the file component number of the current file position.

9.3 Standard Procedures for Non-Textfiles

The standard procedures in this section may, in fact, be applied to textfiles. However, their interpretation when applied to textfiles is somewhat different and is elaborated in §9.4.

9.3.1 The Read Procedure for Non-Textfiles

Reads a file component into a variable.

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

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

v₁, ..., v_n each **v** is a variable-reference with a type that the component type of **f** must be assignment-compatible with.

If we consider **ff** to be the variable referenced by **f**, then this form of **read** is considered to be equivalent to:

```
begin
  read (ff, v1);
  read (ff, v2);
  ...
  read (ff, vn)
end
```

where **read (f, v)** is in turn equivalent to:

```
begin
  v := ff^;
  get(ff)
end
```

Note: There is normally a restriction against passing components of packed variables as actual variable parameters (§7.3.2). This interpretation of **read** means that each **v** is *not* considered an actual variable parameter and *may* be a component of a packed variable.

To understand why the distinction has to be made between **f** and **ff** above, consider the following example:

```
var
  a: array[ 1..10 ] of file of integer;
  i, j : integer;
...
i := 1;
read(a[i], i, j);
```

If, say, the value of **i** that is read is 2, and **a[i]** is reevaluated for each **v**, then the value read for **i** will be read from **a[1]** and the value of **j** from **a[2]**. In fact, **a[i]** is evaluated only once before anything is read, and thus all values are read from **a[1]**. **ff** is the result of this one-time evaluation. This **ff** notation will be used again in subsequent sections.

9.3.2 The Write Procedure for Non-Textfiles

Writes a file component from a variable.

```
write(f, e1 [, e2, ..., en ])
```

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

e₁, ..., e_n each **e** is an expression with a type that must be assignment-compatible with the component type of **f**.

If we consider **ff** to be the variable referenced by **f**, then this form of **write** is considered to be equivalent to:

```
begin
  write(ff, e1);
  write(ff, e2);
  ...
  write(ff, en)
end
```

where **write(f, e)** is in turn equivalent to:

```
begin
  ff^ := e;
  put(ff)
end
```

9.4 Standard Procedures and Functions for Textfiles

This section describes input and output using file variables of the standard type **text**. As previously noted, in Pascal the type **text** is distinct from **packed file of char**. A textfile is still considered to be a sequence of character components (i.e. is it still a **packed file of char**). However, it is additionally considered to be a sequence of lines, where each line is terminated by an **end-of-line** character.

All of the standard procedures and functions in §9.2 may still be applied to a textfile as though it were a **packed file of char**. However, there are additional procedures and functions you can use with textfiles but not other files.

Note: When the value of the file component at the current file position of a file **f** is an end-of-line character, it appears in the file buffer **f^** as a space character.

In particular, there are special forms of **read** and **write** that allow you to read and write values that are not of type **char** and will translate them to and from their character representation. For example, **read(f, i)** where **i** is an integer variable will read a sequence of digits (a digit being one of the characters '0' through '9'), interpret that sequence as an integer-type value, and store it in **i**.

As noted previously there are two standard textfile variables, `input` and `output`. The standard file variable `input` is a read-only file associated with your keyboard. If `input` appears in the program parameter list, then the `input` file is opened automatically when program execution begins as though a `reset` were performed for it. The standard file variable `output` is a write-only file associated with the Text window. If `output` appears in the program parameter list, then the `output` file is opened automatically when program execution begins as though a `rewrite` were performed for it.

All of the standard procedures and functions in this section need not have a file variable explicitly given as a parameter (in addition to `eof`, as described in §9.2.5). In these cases, `input` or `output` will be assumed by default, depending on whether the procedure or function is input-oriented or output-oriented.

9.4.1 The Read Procedure for Textfiles

Reads one or more values from a textfile into one or more program variables.

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

f an optional variable-reference that refers to a variable of type `text`. The file must be open. If **f** is omitted, it is assumed to be the standard text file `input`.

v₁, ..., v_n each **v** is a variable-reference that refers to a variable of one of the following types:

- `Char` or a subrange of `char`.
- An integer-type: `integer` (or a subrange) or `longint`.
- A real-type: `real`, `double`, `extended`, or `computational`.
- An enumerated-type (including `boolean`) or a subrange.
- A string-type.

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

```
begin
  read(ff, v1);
  read(ff, v2);
  ...
  read(ff, vn)
end
```

9.4.1.1 Read with a Char-Type Variable

This is considered equivalent to:

```
begin
  v := ff^;
  get(ff)
end
```

Remember that if the current file position is over an end-of-line character, `ff^` contains a space character.

9.4.1.2 Read with an Integer-Type Variable

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

In reading the sequence of characters, blanks and end-of-line characters preceding the first digit or the sign are skipped. Reading ceases as soon as a character is reached that, together with the characters already read, does not form part of a signed whole number, or as soon as `eof(f)` becomes true.

It is an error if a signed whole number is not found after skipping any preceding blanks and end-of-line characters.

The following things are true immediately after `read(f, v)` when `v` is of an integer-type:

- The current file position will be over the character following the last character in the numeric string, unless the last character in the string was the last character in the file.
- `Eof(f)` will return `true` if the last character in the numeric string was the last character in the file.
- `Eoln(f)` will return `true` if the last character in the numeric string was the last character on the line.

9.4.1.3 Read with a Real-Type Variable

If `f` is of type `text` and `v` is of a real-type, then `read(f, v)` implies the reading from `f` of a sequence of characters that represents a signed-number according to the syntax of §1.4 (again, except for hexadecimal notation). If the value read is assignment-compatible with the type of `v`, then the value is assigned to the variable `v`; otherwise, it is an error.

In reading the sequence of characters, blanks and end-of-line characters preceding the first digit or the sign are skipped. Reading ceases as soon as a character is reached that, together with the characters already read, does not form a valid signed-number.

It is an error if a valid signed-number is not found after skipping any preceding blanks and end-of-line characters.

Immediately after `read(f, v)`, where `v` is a real-type variable, the following conditions are true:

- The current file position will be over the character following the last character in the numeric string, unless the last character in the string was the last character in the file.
- `Eof(f)` will return `true` if the last character in the numeric string was the last character in the file.
- `Eoln(f)` will return `true` if the last character in the numeric string was the last character on the line.

9.4.1.4 Read with a String-Type Variable

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

Note: Read with a string variable does not skip to the next line after reading, and the end-of-line character is left waiting in the file buffer. For this reason, you cannot use successive `read` calls to read a sequence of strings, as they will never get past the first line — after the first read, each subsequent read will see the end-of-line and will read a zero-length string. Instead, use `readln` to read string values (see §9.4.2). `Readln` skips to the beginning of the next line after reading.

The following things are true immediately after `read(f, v)` when v is of a string-type:

- The current file position will be over the character following the last character in the string, unless the last character in the string was the last character in the file.
- `Eof(f)` will return `true` if the last character in the string was the last character in the file.
- `Eoln(f)` will return `true` unless `eof(f)` is `true`.

9.4.1.5 Read with an Enumerated-Type Variable

If f is of type `text` and v is of an enumerated type, then `read(f, v)` implies the reading from f of a sequence of characters that form an identifier according to the syntax of §1.2. If the identifier read is identical (ignoring the case of letters) to an enumerated constant of the enumerated type of v , the value of the enumerated constant is assigned to v ; otherwise, it is an error.

In reading the sequence of characters, blanks and end-of-line characters preceding the first letter of the identifier are skipped. Reading ceases as soon as a character is reached that, together with the characters already read, does not form part of an identifier, or as soon as `eof(f)` becomes `true`.

It is an error if an identifier is not found after skipping any preceding blanks and end-of-line characters.

If f is of type `text`, the following things are true immediately after `read(f, v)` when v is an enumerated-type variable:

- The current file position will be over the character following the last character in the identifier, unless the last character in the string was the last character in the file.
- `Eof(f)` will return `true` if the last character in the identifier was the last character in the file.
- `Eoln(f)` will return `true` if the last character in the identifier was the last character on the line.

9.4.2 The Readln Procedure

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

`readln` [([`f`,] [`v`₁, `v`₂, ..., `v`_{*n*}])]

`f` same as in `read`.

`v`₁, ..., `v`_{*n*} same as in `read`, except they are optional. In fact, the whole parameter list can be omitted.

If the first parameter does not specify a file variable, or if the parameter-list is omitted, the procedure reads from the standard file input.

`Readln(f)`, with no input-variables, causes the current file position to advance to the beginning of the next line (if there is one, else to the end of the file), i.e.:

```
begin
  while not eof(ff) and not eoln(ff) do
    get(ff);
  if not eof(ff) then
    get(ff)
end
```

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

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

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

- `Eof(f)` will return `true` if the line read was the last line in the file.
- `Eoln(f)` will return `false` unless the line following the line read is empty.

9.4.3 The Write Procedure for Textfiles

Writes one or more values to a text file.

```
write([ f, ] P1 [, P2, ..., Pn ])
```

f an optional variable-reference that refers to a variable of type **text**. The file must be open. If **f** is omitted, the procedure writes to the standard file output..

P₁, ..., P_n the **write-parameters**. Each write-parameter includes an **output expression**, whose value is to be written to the file. As explained below, a write-parameter may also contain the specifications of a field-width and a number of decimal places. Each output expression must have a result of **char-type**, an **integer-type**, a **real-type**, a **string-type**, a **packed-string-type**, or an **enumerated-type**. At least one write-parameter must be present.

Write(f, P₁, ..., P_n) is equivalent to:

```
begin
    write(ff, P1);
    ...
    write(ff, Pn)
end
```

9.4.3.1 Write-Parameters

Each write-parameter has the form

```
OutExpr [ : MinWidth [ : DecPlaces ] ]
```

where **OutExpr** is an output expression. **MinWidth** and **DecPlaces** are expressions with integer-type values.

MinWidth specifies the minimum field width. **MinWidth** must be greater than zero. Exactly **MinWidth** characters are written (using leading spaces if necessary), except when **OutExpr** has a value that must be represented in more than **MinWidth** characters; in this case, enough characters are written to represent the value of **OutExpr**. Likewise, if **MinWidth** is omitted, then enough characters as necessary are written to represent the value of **OutExpr**.

DecPlaces specifies the number of decimal places in a fixed-point representation of a real value. It can be specified only if **OutExpr** has a real-type value, and if **MinWidth** is also specified. If specified, it must be greater than zero. If **DecPlaces** is not specified, a floating-point representation is written.

9.4.3.2 Write with a Char-Type Value

If **MinWidth** is omitted, the character value of **OutExpr** is written on the file **f**. Otherwise, **MinWidth-1** spaces followed by the character value of **OutExpr** is written.

9.4.3.3 Write with a String-Type Value

Assuming the string value of `OutExpr` has a length `L`, if `L < MinWidth`, the string value is written on the file `f` preceded by `MinWidth - L` spaces. If `L > MinWidth`, the first `MinWidth` characters of the string are written. If `L = MinWidth`, or if `MinWidth` is omitted, the entire string value is written on the file.

9.4.3.4 Write with an Integer-Type Value

If `OutExpr` has an integer-type value, its decimal (base 10) representation is written on the file `f`. Assume that `OutDigits` is a string-type value that contains the decimal representation of `abs(OutExpr)` with no leading zeros unless the value of `OutExpr = 0`, in which case `OutDigits` contains the single character '0'. If `MinWidth` is omitted from the write-parameter, then it is assumed to be zero. Thus, the representation of `OutExpr` is written to `f` as if by the algorithm:

```
begin
  if MinWidth >= length(OutDigits) + 1 then
    write(ff, ' ' : MinWidth - length(OutDigits) - 2);
  if OutExpr < 0 then
    write(ff, '-')
  else if MinWidth >= length(OutDigits) + 1 then
    write(ff, ' ');
  write(ff, OutDigits)
end
```

9.4.3.5 Write with a Real-Type Value

If `OutExpr` has a real-type value, its decimal representation is written on the file `f`. This representation depends on the presence or absence of `and`, if present, the value of `DecPlaces`.

If `DecPlaces` is present, a **fixed-point** representation is written. Assume that `IntDigits` is a string-type value that contains the decimal representation of `trunc(abs(OutExpr))` with no leading zeroes unless the value of `OutExpr = 0`, in which case `IntDigits` contains the single character '0'. Assume that `FracDigits` is a string-type value that contains the decimal representation of

$$\text{round}((\text{abs}(\text{OutExpr}) - \text{trunc}(\text{abs}(\text{OutExpr}))) * 10^{\text{DecPlaces}})$$

with enough leading zeroes to make `length(FracDigits) = DecPlaces`. Thus, the fixed-point representation is written to `f` as if by the algorithm:

```
begin
  if MinWidth >= length(IntDigits) + length(FracDigits) + 2 then
    write(ff, ' ' : MinWidth - TotalDigits - 3);
  if OutExpr < 0 then
    write(ff, '-')
  else if MinWidth >= length(IntDigits) + length(FracDigits) + 2 then
    write(ff, ' ');
  write(ff, IntDigits, '.', FracDigits)
end
```

If `DecPlaces` is not specified, a **floating-point** representation is written. If `MinWidth` is omitted from the write-parameter, then it is assumed to be 10. Assume that `abs (OutExpr)` has a representation in the floating-point notation of the form:

$$m.n \times 10^e$$

where $0 < m \leq 9$ unless `OutExpr=0`, in which case $m=n=e=0$. Assume that `IntDigit` is a string-type value that contains the decimal representation of m (a single digit). Assume that `FracDigits` is a string-type value that contains the first `MinWidth-9` digits of the decimal representation of n rounded, and with leading zeros retained and trailing zeros added if necessary. Assume that `ExpDigits` is a string-type value that contains the decimal representation of e with enough leading zeros to make `length (ExpDigits)=4`. Also assume that `NegExp` has the value `true` if $e < 0$, and otherwise the value `false`.

Thus, the floating-point representation is written to `f` as if by the algorithm:

```
begin
  if OutExpr<0 then
    write(ff, '-')
  else
    write(ff, ' ');
  write(ff, IntDigit, '.', FracDigits, 'E');
  if NegExp then
    write(ff, '-')
  else
    write(ff, '+');
  write(ff, ExpDigits)
end
```

9.4.3.6 Write with a Packed-String-Type Value

If `OutExpr` is of a packed-string-type, the effect is the same as writing a string whose length is the number of components in the type.

9.4.3.7 Write with an Enumerated-Type Value

If the value of `OutExpr` is of an enumerated type, the string representation of the enumerated constant identifier corresponding to the value is written on the file `f`. If the length of this string representation is `L` and $L < \text{MinWidth}$, then $\text{MinWidth}-L$ spaces are written out before the string. In any case the entire string is always written, even if $L > \text{MinWidth}$.

9.4.4 The Writeln Procedure

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

```
writeln [ ([ f, ] [ p1, p2, ..., pn ] ) ]
```

f same as for `write`.

p₁, ..., p_n same as for `write`, except they are optional. In fact, the whole parameter list may be omitted.

If the first parameter does not specify a file variable, or if the parameter-list is omitted, the procedure writes to the standard file output.

`Writeln(f)` writes an end-of-line character to the file `f`.

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

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

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

- `Eof(f)` will return `true` if the last character written became the last character in the file. If `f` is write-only, then `eof(f)` will necessarily be `true`.
- `Eoln(f)` will return `false` unless the character following the last character written is an end-of-line character.

9.4.5 The Eoln Function

```
eoln [ (f) ]
```

f a variable-reference that refers to a variable of type `text`. The file must be open. If `f` is omitted, the function is applied to the standard file input.

Returns `boolean`

`Eoln(f)` returns `true` if the character at the current file position is an end-of-line character. It is an error to call `eoln(f)` if `f` is a non-textfile, if `f` is write-only, or if `eof(f)` is `true`.

Note: Every line in a file is expected to be terminated by an end-of-line character. This may not actually be the case. The last character in a file may not be an end-of-line character as it should. If a file `f` is read-only (opened with `reset`) then, upon reaching the end of the file, if the last character was not an end-of-line character, `f^` becomes a space character, `eoln(f)` becomes `true`, and `eof(f)` remains `false`. The *next* attempt to read a character will then cause `eof(f)` to become `true`. This will *only* happen if the file is read-only and not if it is read/write.

9.4.6 The Page Procedure

page [(f)]

f a variable-reference that refers to a variable of type `text`. The file must be open. If **f** is omitted, the standard textfile output is assumed.

Page (f) causes a skip to the top of a new page when **f** is printed or displayed. If **f** is write-only, and if the last character in **f** is not an end-of-line character, then one is inserted before the page is done.

9.4.7 Lazy I/O

Consider this small program:

```

program count(input, output);
  var
    s: string;
    ch: char;
begin
  write('Type a line of characters -- ');
  readln(s);
  writeln('You typed ', length(s), ' characters');
end.

```

If you take all the parts of this section literally, there are two problems with this program:

- Because `reset` (which is done implicitly for input when the program starts) causes the first character of input to appear in `input^`, the program will hang waiting for input from the keyboard before the `write` statement is executed. This means the prompt the program is supposed to give you for input will not appear until after you type a character.
- Having typed a line of characters followed by the Return key (the “end-of-line key” so to speak), `readln` causes the first character of the line following the one just read to appear in `input^`. This means that the `writeln` following the `readln` will not be executed until you type another character following the return.

This behavior has been the bane of Pascal programmers since the language was created. This idiosyncrasy exists partly because Pascal was originally designed to run on batch systems — back in the days when interactive systems were rare.

On batch systems, input was expected to be associated with a previously prepared input file (typically a deck of punched cards) and output was expected to be associated with a file where the results of your program would appear for your inspection after its execution was complete (typically a line printer listing).

Although, when the ANSI Pascal Standard was being drafted, many subtle changes were made to the Pascal language, this problem was never completely resolved. So many programs had already been written that depended on this behavior when doing I/O. To change the language in any sig-

nificant way would have made these programs invalid. Instead, the standard is worded in such a way that it is possible to get around this problem.

For instance, when the standard specifies that, after `reset (f)`, `f^` contains the first component of the file, it does so in a way that allows `f^` to remain undefined *until its value is needed*. Thus, in the above program, it's not strictly necessary to read the first character from the keyboard as soon as execution begins; it suffices to do so when the `readln` needs that character. Likewise, after `readln` processes the end-of-line character (the Return key), it is not necessary to then read another character from the keyboard. It is in fact never necessary because the program does not reference input again.

Interpreting the standard's semantics in this way is popularly known as **Lazy I/O**, and is the only technique that allows interactive I/O in Pascal while preserving the standard's I/O semantics. Other techniques exist, but they cause I/O operations to behave differently depending on whether you have specified that the I/O is to be done interactively or not. These other techniques make it difficult to write a program that runs the same when its input comes from a file as it does when input comes from a keyboard.

The Lazy I/O technique involves separating the operations of advancing the current file position and doing input from the file by deferring the actual input of data from a file until absolutely necessary. Conditions that make input from a file `f` necessary include:

- A reference to `f^` other than to assign it a value or to pass it as an actual variable parameter.
- A call to `eof (f)`. In this case, input may be necessary because it is not (necessarily) possible to know whether the end of the file has been reached without trying to read beyond it.
- A call to `eofln (f)`. In this case, input may be necessary to get another character and see if it is an end-of-line character.
- A call to `get (f)`. In this case, input will only be necessary if prior to the call to `get` there was deferred input that was never actually done. If so, the original deferred input will be performed, but the input implied by the `get` will in turn be deferred.
- A call to `read` or `readln` from `f`. Here, the input necessary to do the `read` or `readln` will be performed, but the input of the component following the last component read will be deferred.

Although Lazy I/O makes it possible to write interactive programs without much difficulty, you should nevertheless be aware of the conditions described above to avoid peculiar situations that may cause your program to hang waiting for input unexpectedly. For instance:

```

program process_lines(input, output);
  var
    s : string;
begin
  repeat
    write('>');
    readln(s);
    ...
  until eoln
end.

```

This program might be intended to read and process lines, prompting each line with the '>' character, until an empty line is entered. An empty line is an end-of-line character immediately following the end of the previous line. The program reads a line and tests to see if an end-of-line immediately follows. The problem is that the program, after prompting for and reading the first line, will stop as a result of the eoln and wait for another character to be typed before issuing another prompt.

A better way to write this program, one that avoids this problem, might be:

```

program process_lines(input, output);
  var
    s: string;
    empty : boolean;
begin
  empty := false;
  repeat
    write('>');
    if eoln then
      empty := true
    else
      begin
        readln(s);
        ...
      end
    until empty
end.

```

or even better:

```

program process_lines(input, output);
  var
    s : string;
begin
  repeat
    write('>');
    readln(s);
    ...
  until length(s) = 0
end.

```

9.5 Devices on the Macintosh

On the Macintosh, there are basically three devices to be concerned about: disk drives, a printer, and a modem.

The disk drives are never addressed directly. Rather, you address the disks themselves by name. Each disk is referred to as a **volume** and the disk's name is its **volume name**. For disk files, the title parameter for reset, rewrite, and open (see §§9.2.1 through 9.2.3) consists of a file name optionally preceded by a volume name and a colon, e.g.:

```
MyVolume:MyFile
```

To learn more about volumes and files, see *Inside Macintosh II*, Chapter 4, "The File Manager," *Inside Macintosh IV*, Chapter 19, "The File Manager," and *Inside Macintosh VI*, Chapter 25, "The File Manager."

For the printer and modem, the title parameter for reset, rewrite, and open consists of the device name followed by a colon:

```
Printer:
Modem:
```

The device name for the printer is simply `printer:`. Likewise the device name for the modem is `modem:`. Since the printer is a write-only device, you can only use `'printer:'` as the title parameter for `rewrite`. It is an error to give `'printer:'` as the title parameter for `reset` or `open`.

The device names `printer:` and `modem:` are provided for convenience only. In most cases, using these device names is not sufficient for general applications, and completely inadequate for Macintosh application. Please see *Inside Macintosh* to learn more about the Serial Manager and the Print Manager.

- A file variable opened with `'modem:'` as the title parameter reads from and writes to the modem at 300 baud.

- It is an error to open a file variable with 'printer:' or 'modem:' as the title parameter if it is not of type text.
- For the printer and the modem to work properly, they must be connected to their proper sockets in the back of the Macintosh. See the Macintosh user's guides for details.

9.6 Error Handling Routines

Most of the time, THINK Pascal reports I/O errors with a dialog box. When the dialog box appears, it doesn't give you an opportunity to handle the error in your program. The routines described in this section, let you turn off THINK Pascal's I/O error checking so you can do it yourself.

9.6.1 IOCheck

Enables and disables runtime checking of I/O errors.

IOCheck (bool)

Bool a boolean value. If bool is true, THINK Pascal reports dynamic errors. If bool is false, you can check the results of input/output operations with the IOResult function.

9.6.2 IOResult

Returns result of last of I/O routine.

IOResult

Returns integer

IOResult returns the result of the last I/O operation. If the value is negative, it is a Macintosh error. The meanings of positive values are:

- 0 no error
- 15 can't close an anonymous file
- 16 file is not opened for writing
- 17 file is not opened for reading
- 18 file is not opened for random access
- 19 end of file during read
- 20 file is not open
- 21 file is not a disk file
- 22 component in seek is < 0
- 23 file is already open
- 24 bad device type for open, rewrite, or reset
- 25 port is in use by AppleTalk
- 26 illegal signed number in read
- 27 string too long in read or write
- 28 illegal enum value in read
- 29 illegal floating point value in read

10.0 Standard Procedures and Functions

This section describes all the standard (built-in) procedures and functions in THINK Pascal, except for the I/O procedures and functions described in §9, and the Macintosh Toolbox procedures and functions described in *Inside Macintosh*.

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

Note: You cannot use standard procedures and functions as actual procedural and functional parameters.

This section uses a modified BNF notation, instead of syntax diagrams, to indicate the syntax of actual-parameter-lists for standard procedures and functions. The notation is explained at the beginning of §9.

10.1 Dynamic Allocation Procedures

The procedure `new` creates dynamic variables that your program uses. Dynamic variables are variables that can be accessed only through pointer variables, and they are created by allocating a region of memory from a portion of free memory called the **heap**. The address of the allocated region is the pointer value that is used to access the dynamic variable. The `dispose` procedure is used to destroy dynamic variables created with `new`, in the process returning that variable's region of memory to the heap for reuse.

10.1.1 The New Procedure

Creates a new dynamic variable and sets a pointer variable to point to it, or creates a new object and sets a reference variable to reference it.

`new (p [, c1, c2, ..., cn])`

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

c₁, ..., c_n each *c* (if given) is a constant. If the base-type of *p* is a record-type with variants, then each *c* may be a constant corresponding to a case-constant of a variant of a variant-part (see below). Variants are not allowed for reference variables. THINK Pascal *always* allocates enough space for the *largest* variant regardless of the case-constants supplied.

Or

p a variable-reference that refers to a variable of reference-type.

`New (p)` creates a new variable of the base-type of *p*, and makes *p* point to it. The variable can be referenced as *p*[^]. It is an error if the heap does not contain enough free space to create the new variable.

If the base-type of *p* is a record-type with a variant-part, *new* (*p*) allocates enough memory to the variable to accommodate the largest variant. If *p* is a reference variable, *new* creates an object whose type corresponds to *p*'s reference type and sets *p* to reference it.

Keep in mind that THINK Pascal *always* allocates enough memory for the *largest* case variant. For compatibility with other versions of Pascal, THINK Pascal lets you supply a constant *c* that is a case-constant of one of the variants of the variant-part. If the record variant itself contains a variant-part, then a case-constant *c* may be given to select a particular variant of that variant-part, and so on. Each *c* in the parameter list must be given in the order of the nesting of the variants, one for each level of nesting. In some versions of Pascal, only the amount of memory necessary to accommodate that particular variant is allocated to the variable.

10.1.2 The Dispose Procedure

Destroys a dynamic variable or reference variable.

`dispose (p [, c1, c2, ..., cn])`

p a variable-reference that refers to a pointer-variable. It must be a pointer that was previously assigned by the *new* procedure or was assigned a meaningful value by an assignment statement. It is an error to attempt to *dispose* of a pointer variable that is currently being accessed, or whose value is undefined or is **nil**.

c₁, ..., c_n Each *c* (if given) is a constant. If the base-type of *p* is a record-type with variants, then each *c* may be a constant corresponding to a case-constant of a variant of a variant-part (see 10.1.1 above).

Or:

p a variable reference that refers to a variable of reference-type.

Dispose (*p*) destroys the variable referenced by *p* and returns its memory region to the heap. The value of *p* then becomes undefined and it is an error to subsequently make reference to *p*[^].

If the dynamic variable pointed to by *p* was created by *new* with a list of case-constants, then the same list of case-constants (in the same order) *must* be given to *dispose*.

10.1.3 HeapCheck Procedure

Enables and disables runtime checking of dynamic memory errors.

HeapCheck (bool)

Bool a boolean value. If bool is true, THINK Pascal reports dynamic errors. If bool is false, you can check the results of dynamic memory allocations with the HeapResult function.

10.1.4 HeapResult Function

Returns result of the last of dynamic memory allocation routine.

HeapResult

Returns integer

HeapResult returns the result of the last heap operation. If the value is negative, it is a Macintosh error. The meanings of positive values are:

0 no error
11 attempt to DISPOSE a nil pointer

10.2 Transfer Procedures and Functions**10.2.1 The Trunc Function**

Converts a real-type value to a longint value.

trunc (x)

x an expression with a value of a real-type.

Returns longint

Trunc (x) returns a longint result that is the value of x rounded to the nearest whole number that is between 0 and x inclusive. It is an error if the result of this rounding is outside the range -maxlongint-1..maxlongint.

10.2.2 The Round Function

Converts a real-type value to a longint value.

round (x)

x an expression with a value of a real-type.

Returns longint

Round (x) returns a longint result that is the value of x rounded to the nearest whole number. If x is exactly halfway between two whole numbers, the result is the whole number with the greatest absolute magnitude. It is an error if the result of this rounding is outside the range -maxlongint..maxlongint.

10.2.3 The Ord4 Function

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

`ord4 (x)`

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

Returns longint

`Ord4 (x)` returns the ordinal value of `x`.

If `x` is of a pointer-type, the result is the address of the dynamic variable pointed to by `x`.

If `x` is of an ordinal-type, the result is the ordinality of `x` (see §3.1.1), represented as a longint .

10.2.4 The Pointer Function

Converts an integer-type value to a generic pointer-type value.

`pointer (x)`

x an expression with a value of integer-type.

Returns a generic pointer which matches *any* pointer

`Pointer (x)` returns a pointer value that points to whatever is at the address `x` as though it were a dynamic variable created at that address. This pointer is of the same type as `nil` in that it is assignment-compatible with any pointer-type.

As a convenience, `pointer` may be also applied to an expression of any pointer-type, effectively making that expression assignment-compatible with any (other) pointer-type.

10.3 Arithmetic Functions

In general, any extended real-type result returned by an arithmetic function is an approximation. There is one exception to this: the result of the `abs` function is exact.

10.3.1 The Odd Function

Tests whether an integer-type value is odd.

`odd (x)`

x an expression with a value of integer-type.

Returns boolean

`Odd (x)` returns `true` if `x` is odd, i.e. not divisible by 2 without a remainder. If `x` is even it returns `false`.

10.3.2 The Abs Function

Returns the absolute value of a numeric value.

`abs (x)`

x an expression with a value of an integer-type or a real-type.

Returns integer, longint, or extended.

`Abs (x)` returns the absolute value of `x`; i.e. if `x` is negative, `-x` is returned; otherwise `x` is returned. If `x` is of a real-type, the result type is extended. If `x` is of type `longint`, the result type is `longint`. Otherwise, the result type is `integer`.

10.3.3 The Sqr Function

Returns the square of a numeric value.

`sqr (x)`

x an expression with a value of an integer-type or a real-type.

Returns integer, longint, or extended.

`Sqr (x)` returns the square of `x`, i.e. `x*x`.

It is an error if the result is not within the range of values representable by the result-type (see §3.1).

10.3.4 The Sqrt Function

Returns the square root of a numeric value.

`sqrt (x)`

x an expression with a value of an integer-type or real-type. It is an error if `x < 0`.

Returns extended.

`Sqrt (x)` returns the positive square root of `x`, i.e. the positive value `y` such that `y*y=x`. It is an error if the result is a value too small to be represented by the real-type extended (see §3.1.2).

10.3.5 The Sin Function

Returns the sine of a numeric value.

`sin (x)`

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

Returns extended.

`Sin (x)` returns the trigonometric sine of `x`.

10.3.6 The Cos Function

Returns the cosine of a numeric value.

`cos (x)`

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

Returns extended.

`Cos (x)` returns the trigonometric cosine of `x`.

10.3.7 The Exp Function

Returns the exponential of a numeric value.

`exp (x)`

x an expression with a value of an integer-type or real-type.

Returns extended.

`Exp (x)` returns the value of e^x , where e is the base of the natural logarithms. It is an error if the result cannot be represented with the real-type extended (see §3.1.2).

10.3.8 The Ln Function

Returns the natural logarithm of a numeric value.

`ln (x)`

x an expression with a value of an integer-type or real-type. It is an error if $x \leq 0$.

Returns extended.

`Ln (x)` returns the natural logarithm (\log_e) of `x`.

10.3.9 The Arctan Function

Returns the arctangent of a numeric value.

`arctan(x)`

x an expression with a value of an integer-type or real-type. It is an error if $x < 0$.

Returns extended.

`Arctan(x)` returns the principal value, in radians, of the arctangent of `x`.

10.4 Ordinal Functions

10.4.1 The Ord Function

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

`ord(x)`

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

Returns integer or `longint`

If `x` is of pointer-type, the result is the `longint` address of the dynamic variable pointed to by `x`.

If `x` is of an ordinal-type, the result type is the ordinality of `x` (see §3.1.1). If `x` is of type `longint`, the result type is `longint`. Otherwise, the result type is `integer`.

10.4.2 The Chr Function

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

`chr(x)`

x an expression with an integer-type value that must be in the range 0..255.

Returns `char`

`Chr(x)` returns the `char` value whose ordinal number is `x`.

For any `char` value `ch`, the following is always true:

`chr(ord(ch)) = ch`

10.4.3 The Succ Function

Returns the successor of a value of ordinal-type.

`succ (x)`

x an expression with a value of ordinal-type.

Returns same as parameter.

`Succ (x)` returns the successor of `x`.

It is an error if `x` is the last value in the type of `x`, i.e. it has no successor. Otherwise

`ord (succ (x)) = ord (x) + 1`

10.4.4 The Pred Function

Returns the predecessor of a value of ordinal-type.

`pred (x)`

x an expression with a value of ordinal-type.

Returns same as parameter.

`Pred (x)` returns the predecessor of `x`.

It is an error if `x` is the first value in the type of `x`, i.e. it has no predecessor. Otherwise,

`ord (pred (x)) = ord (x) - 1`

10.5 String Procedures and Functions

10.5.1 The Length Function

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

`length (str)`

str an expression with a value of a string-type.

Returns integer

`Length (str)` returns the current length attribute of `str` (see §3.3).

10.5.2 The Pos Function

Searches a string for the first occurrence of a specified substring.

`pos(substr, str)`

`substr` an expression with a value of a string-type.

`str` an expression with a value of a string-type.

Returns integer

`Pos(substr, str)` searches for `substr` within `str`, and returns an integer value that is the index of the first character of `substr` within `str`.

If `substr` is not found, `pos(substr, str)` returns zero.

10.5.3 The Concat Function

Takes a sequence of strings and concatenates them.

`concat(s1 [, s2, ... sn])`

`s1, ..., sn` each is an expression with a value of string-type. Any practical number of parameters may be passed.

Returns string-type

`Concat(s1, ..., sn)` concatenates all the parameters in the order in which they are written, and returns the concatenated string. Note that the number of characters in the result cannot exceed 255.

10.5.4 The Copy Function

Returns a substring of specified length, taken from a specified position within a string.

`copy(source, index, count)`

`source` an expression with a value of a string-type.

`index` an expression with an integer-type value.

`count` an expression with an integer-type value.

Returns string-type

`Copy(source, index, count)` returns a string containing `count` characters from `source`, beginning at `source[index]`. If `count ≤ 0`, then a null string is returned. If `index < 1` or `index + count > length(source)`, i.e. if character positions outside the range `1..length(source)` are implicitly referenced, it is *not* an error. However, only the characters that lie within that range are copied.

10.5.5 The Delete Procedure

Deletes a substring of specified length from a specified position within the value of a string variable.

`delete(dest, index, count)`

dest a variable-reference that refers to a variable of a string-type. This is a variable parameter.

index an expression with an integer-type value.

count an expression with an integer-type value.

`Delete(dest, index, count)` removes `count` characters from the value of `dest`, beginning at `dest[index]`. If `index < 1` or `index + count > length(source)`, i.e. if character positions outside the range are implicitly referenced, it is *not* an error. However, only the characters that lie within that range are deleted.

10.5.6 The Omit Function

Deletes a substring of specified length from a specified position within a string value and returns the result.

`omit(str, index, count)`

str a value of string-type.

index an expression with an integer-type value.

count an expression with an integer-type value.

Returns string-type

`Omit(str, index, count)` removes `count` characters from the value of `str`, beginning at `dest[index]`, and returns the resulting string value. This is similar to `delete` except that `str` is not affected; the resulting string value is returned as the value of the function instead.

10.5.7 The Insert Procedure

Inserts a substring into the value of a string variable, at a specified position.

`insert (source, dest, index)`

source an expression with a value of string-type.

dest a variable-reference that refers to a variable of string-type. This is a variable parameter.

index an expression with an integer-type value.

`Insert (source, dest, index)` inserts `source` into `dest`. The first character of `source` becomes `dest [index]`. If `index < 1` or `index > length (dest)`, it is *not* an error. If `index < 1` then `source` is appended to the left of `dest`. If `index > length (dest)` then `source` is appended to the right of `dest`. It *is* an error, however, if the length of the resulting string is greater than 255.

10.5.8 The Include Function

Inserts a substring into a string value, at a specified position, and returns the result.

`include (source, str, index)`

source an expression with a value of string-type.

str a value of string-type.

index an expression with an integer-type value.

Returns string-type

`Include (source, str, index)` inserts `source` into the value of `str` at `str [index]` and returns the result. This is similar to `insert` except that `str` is not affected; the resulting string value is returned as the value of the function instead.

10.6 THINK Pascal Window Manipulation Procedures

10.6.1 The HideAll Procedure

`HideAll`

`HideAll` causes all of the windows on the THINK Pascal desktop to be hidden. All of these windows may be revealed again with the **Windows** menu. In addition, the Text and Drawing windows may be revealed by calling the procedures described below.

10.6.2 The ShowText Procedure

`ShowText`

`ShowText` causes the Text window to be revealed and to become the active window. The size and position of the window is unchanged.

10.6.3 The ShowDrawing Procedure

ShowDrawing

ShowDrawing causes the Drawing window to be revealed and to become the active window. The size and position of the window is unchanged.

10.6.4 The SetTextRect Procedure

SetTextRect (WRect)

WRect a value of type Rect.

WRect is a rectangle in QuickDraw's global coordinate system that determines the position and size of the Text window on the Macintosh screen.

10.6.5 The SetDrawingRect Procedure

SetDrawingRect (WRect)

WRect a value of type Rect.

WRect is a rectangle in QuickDraw's global coordinate system that determines the position and size of the Drawing window on the Macintosh screen.

10.6.6 The GetTextRect Procedure

GetTextRect (WRect)

WRect a value of type Rect.

GetTextRect returns a rectangle in WRect with coordinates in QuickDraw's global coordinate system. This rectangle indicates the current size and position of the Text window.

10.6.7 The GetDrawingRect Procedure

GetDrawingRect (WRect)

WRect a value of type Rect.

GetDrawingRect returns a rectangle in WindowRect with coordinates in QuickDraw's global coordinate system. This rectangle indicates the current size and position of the Drawing window.

10.6.8 The SaveDrawing Procedure

SaveDrawing (title)

title a string-type value that must contain a valid file name for a file-structured device (see §9.5).

SaveDrawing saves the contents of the Drawing window as a file that may be read by MacPaint. The title string contains the name of the picture file to be created. If a file by that name already exists, it is overwritten.

Note: SaveDrawing actually saves the contents of the current QuickDraw GrafPort. However, unless you specifically change the current port to be another

port, the current port will always be the Drawing window's GrafPort when your program is running.

10.7 Bit operations

These procedures and functions in this section operate on the bits of values. These functions and procedures actually generate inline code.

10.7.1 BitAnd

BitAnd(N1, N2)

N1, N2 either integer or longint values.

Returns integer or longint

BitAnd returns N1 AND N2. If both of the arguments to BitAnd are integer, the result is integer. If one or both of the arguments is a longint, the result is a longint.

10.7.2 BitOr

BitOr(N1, N2)

N1, N2 either integer or longint values.

Returns integer or longint

BitOr returns N1 OR N2. If both of the arguments to BitOr are integer, the result is integer. If one or both of the arguments is a longint, the result is a longint.

10.7.3 BitXor

BitXor(N1, N2)

N1, N2 either integer or longint values.

Returns integer or longint

BitXor returns N1 XOR N2. If both of the arguments to BitXor are integer, the result is integer. If one or both of the arguments is a longint, the result is a longint.

10.7.4 BitNot

BitNot(aNum)

aNum either an integer or longint value.

Returns integer or longint

BitAnd returns the one's complement of aNum, that is, all the bits are inverted. If the argument to BitNot is integer, the result is integer. If the argument is a longint, the result is a longint.

10.7.5 BAND

BAND(*x*, *y*)

x, *y* scalar values of any size. Values smaller than 32 bits are zero extended.

Returns *longint*

BAND returns *x* AND *y*.

10.7.6 BOR

BOR(*x*, *y*)

x, *y* scalar values of any size. Values smaller than 32 bits are zero extended.

Returns *longint*

BOR returns *x* OR *y*.

10.7.7 BXOR

BXOR(*x*, *y*)

x, *y* scalar values of any size. Values smaller than 32 bits are zero extended.

Returns *longint*

BXOR returns *x* XOR *y*.

10.7.8 BNOT

BNOT(*x*)

x a scalar value of any size. Values smaller than 32 bits are zero extended.

Returns *longint*

BNOT returns NOT *x*, that is the one's complement of *x*.

10.7.9 BSL

BSL(*x*, *y*)

x, *y* scalar values of any size. Values smaller than 32 bits are zero extended.

Returns *longint*

BSL returns *x* left-shifted by *y* bits.

10.7.10 BSR**BSR**(*x*, *y*)*x*, *y* scalar values of any size. Values smaller than 32 bits are zero extended.**Returns** longintBSR returns *x* right-shifted by *y* bits.**10.7.11 BROTL****BROTL**(*x*, *y*)*x*, *y* scalar values of any size. Values smaller than 32 bits are zero extended.**Returns** longintBROTL returns *x* left-rotated by *y* bits. The high order bits rotate the low order bits.**10.7.12 BROTR****BROTR**(*x*, *y*)*x*, *y* scalar values of any size. Values smaller than 32 bits are zero extended.**Returns** longintBROTR returns *x* right-rotated by *y* bits. The low order bits rotate to the high order bits.**10.7.13 BTST****BTST**(*x*, *y*)*x*, *y* scalar values of any size. Values smaller than 32 bits are zero extended.**Returns** booleanBTST returns `true` if bit *y* in *x* is set. The high order bit is bit 31; the low order bit is bit 0.**10.7.14 BCLR****BCLR**(*x*, *y*)*x* a var longint variable reference*y* an integer reduced modulo 32.BCLR clears bit *y* of *x*.

10.7.15 BSET

BSET(x,y)

x	a var longint variable reference
y	an integer reduced modulo 32.

BSET sets bit y of x.

10.7.16 HiWord

HiWord(long)

long	a longint value.
------	------------------

Returns integer

HiWord returns the upper 16-bits of a longint value as an integer value.

10.7.17 HiWrd

HiWrd(long)

long	a longint value.
------	------------------

Returns integer

HiWrd returns the upper 16-bits of a longint value as an integer value.

10.7.18 LoWord

LoWord(long)

long	a longint value.
------	------------------

Returns integer

LoWord returns the lower 16-bits of a longint value as an integer value.

10.7.19 LoWrd

LoWrd(long)

long	a longint value.
------	------------------

Returns integer

LoWrd returns the lower 16-bits of a longint value as an integer value.

10.8 Control Procedures

These procedures let you exit loops and routines.

10.8.1 Cycle

Cycle to next repetition of enclosing loop statements.

Cycle

Cycle goes to the next repetition of the enclosing **while**, **repeat**, or **for** statement. It can only be used within one of these statements,

10.8.2 Leave

Leave enclosing loop statements.

Leave

Leave goes to the statement following the enclosing **while**, **repeat**, or **for** statement. It can only be used within one of these statements,

10.8.3 Exit

Exit enclosing procedure.

Exit (procName)

procName the name of an enclosing procedure.

Exit returns from the procedure procName. ProcName must be the name of the procedure in which the statement appears, or it must nest the procedure in which it appears.

10.8.4 Halt

Exit the program

Halt

Halt exits the program.

10.9 Miscellaneous Procedures and Functions**10.9.1 The Sizeof Function**

Returns the number of bytes occupied by a specified variable, or by any variable of a specified type.

sizeof (id)

id either a variable-identifier or a type-identifier

Returns integer or longint

Sizeof (id) returns the number of bytes of memory occupied by id. If id is a variable-identifier; if id is a type-identifier, it returns the number of bytes occupied by any variable of type id. If the size fits in an integer, sizeof returns an integer; otherwise, it returns a longint.

10.9.3 The WriteDraw Procedure

WriteDraw is similar to write, except that the text output goes to the current GrafPort at the current PenLoc instead of to a text file or to the Text window.

WriteDraw (p₁ [, p₂, ..., p_n])

p₁, ..., p_n the **write-parameters**. Each write-parameter includes an **output expression**, whose value is to be written to the file. As explained in §9.4.3.1, a write-parameter may also contain the specifications of a field-width and a number of decimal places. Each output expression must have a result of char-type, an integer-type, a real-type, a string-type, a packed-string-type, or an enumerated-type. At least one write-parameter must be present.

WriteDraw takes the same parameter list as write (see §9.4.3), except that no file parameter is ever given. The text that results from the evaluation of each write-parameter is written in the current GrafPort starting at the current pen position.

Do not forget to set current PenLoc before using WriteDraw.

10.9.4 The StringOf Function

StringOf is also similar to write, except that the text output is returned as a string-type value instead of being written to a textfile or to the Text window.

StringOf (p₁ [, p₂, ..., p_n])

p₁, ..., p_n the **write-parameters**. Each write-parameter includes an **output expression**, whose value is to be written to the file. As explained in §9.4.3.1, a write-parameter may also contain the specifications of a field-width and a number of decimal places. Each output expression must have a result of char-type, an integer-type, a real-type, a string-type, a packed-string-type, or an enumerated-type. At least one write-parameter must be present.

Returns string-type

StringOf takes the same parameter list as write (see §9.4.3), except that no file parameter is ever given. The text that results from the evaluation of each write-parameter is accumulated as a string-type value that is the result of the function call.

10.9.5 The ReadString Procedure

ReadString is similar to read, except that the text is read from a string parameter instead of a textfile.

ReadString(s, v₁ [, v₂, ..., v_n])

s a string-type value

v₁, ..., v_n Each v is a variable-reference that refers to a variable of one of the following types:

- Char or a subrange of char.
- An integer-type: integer (or a subrange) or longint.
- A real-type: real, double, extended, or computational.
- An enumerated-type (including boolean) or a subrange.
- A string-type.

ReadString reads text from the string value s just as if it were doing a read (see §9.4.1) from a textfile. The values read are placed in the v parameters in the given order. Just as it is an error to attempt to read beyond the end of a file, it is an error to attempt to read characters beyond the end of the string value s.

10.9.6 The OldFileName Function

Returns the title of an existing disk file selected by the user.

OldFileName (Prompt)

prompt a string-type value.

Returns string-type

OldFileName causes a dialog box to appear on the Macintosh screen. The Prompt string is *not* displayed. It is there for historical reasons, but must be supplied. With the dialog box, the user can peruse the existing files on any number of disks and select one of them. OldFileName returns a string value that is the title of the file the user selected, which can in turn be given to reset, rewrite, or open.

10.9.7 The NewFileName Function

Returns the title of a new disk file selected by the user.

NewFileName (Prompt [, Name])

Prompt a string-type value

Name an optional string-type value.

Returns string-type

NewFileName causes a dialog box to appear on the Macintosh screen. The **Prompt** string is displayed within this box to give the user some indication of what the box is asking for. With the dialog box, the user can select any disk and enter the name (or choose the default name) of a file to be created on that disk. **NewFileName** returns a string value that is the title of the file the user selected, which can in turn be given to **reset**, **rewrite**, or **open**.

The **Name** string is displayed in the dialog box as selected text, and will be the string returned unless the user enters another name.

10.9.8 The Synch Procedure

Synch

The **Synch** procedure is used to synchronize your program's actions with the Macintosh screen's drawing cycle, which occurs every 60th of a second. This may, in particular, be used to synchronize calls to **QuickDraw** with the screen drawing cycle to avoid unnecessary flicker and scanning bar phenomena when moving things around quickly in the **Drawing** window.

When you call the **synch** procedure, the procedure will not return until the screen has reached the point in its cycle where the electron beam has returned to the top of the screen and is about to redraw the entire picture. Depending on where your **QuickDraw** calls are going to be drawing on the screen, additional delays may be necessary to synchronize the drawing with the timing of the electron beam's scanning of the screen.

10.9.9 The Note Procedure

Note (Freq, Ampl, Duration)

Freq The frequency, an **longint** value in the range 12..783360

Ampl The amplitude, an integer-type value in the range 0..255

Duration The duration, an integer-type value in the range 0..255.

Note causes a single square-wave tone to be generated, of the given amplitude, duration, and frequency (the frequency is specified in hertz).

10.9.10 The InLine Procedures

```

InlineP (Trap [, P1, P2, ..., Pn ])
BInlineF(Trap [, P1, P2, ..., Pn ])
WInlineF(Trap [, P1, P2, ..., Pn ])
LInlineF(Trap [, P1, P2, ..., Pn ])

```

Trap an expression with an integer value that indicates the number of the trap to be called.

P₁, ..., P_n each p (if given) is an expression with any type value.

The InLine procedures provides the ability to call the stack-based Macintosh Toolbox routines. The InLine procedures work by disabling all type and parameter checking normally used in THINK Pascal. This makes it possible for just four predefined routines to invoke any stack-based Toolbox trap.

Toolbox routines can be either functions or procedures. Functions can return either a byte (8-bits), a word (16-bits), or a longword 32-bits as results.

Invoke Toolbox routines with one of the three following routines:

This routine Invokes these Toolbox routines

BinlineF Toolbox functions which return a byte (boolean)

WInlineF Toolbox functions which return a word (integer)

LInlineF Toolbox functions which return a longword (longint)

InlineP Toolbox procedures

All procedures and functions use the same basic parameter-passing mechanism. Parameters are passed either by reference (variable parameters) or by value. To force a parameter to an InLine routine to be passed by reference, the @ operator may be applied to the name of the variable that is to be passed. For example,

```
@WindPtr
```

will force a reference to WindPtr to be passed to a Toolbox routine. Simply using a variable, constant, or literal value will pass a parameter by value.

The first parameter to any InLine call is the value which specifies the trap number. The trap number indicates which Toolbox routine is to be called. All subsequent parameters must exactly match the number and type of the parameters for the particular Toolbox routine being called.

Warning: Because there is no type checking, none of THINK Pascal's usual implicit type-coercion will be performed (e.g., integer to longint). All parameters *must* match exactly the parameters for the particular Toolbox routine being called.

Note: The `InLine` procedures are provided only for compatibility with Macintosh Pascal. THINK Pascal provides direct access to the Macintosh toolbox and therefore the only need for `InLine` is for porting Macintosh Pascal programs which used these routines.

10.9.11 The Generic Procedure

`Generic(Instr, Regs)`

Instr an expression with an integer value that indicates the instruction to be executed.

Regs a variable-reference of type `RegisterRecord` (see below) that indicates the values to be written to the MC68000 registers.

`Generic` lets you call register-based Macintosh ROM routines. It can also be used to execute any machine-language code that you have stored in a Pascal data structure.

`RegisterRecord` denotes a data structure consisting of 13 32-bit values - five address register values (A0 . . A4), followed by eight data register values (D0 . . D7). The exact type of this structure is immaterial. For example, you could declare:

```
var regs:   record
            a: array[0..4] of longint;
            d: array[0..7] of longint
        end;
```

The register values passed to `Generic` are written to the MC68000 registers. Then the one-word instruction denoted by the `Instr` argument is executed. Finally, the `Regs` structure is updated with the (possibly) new values of the MC68000 registers before `Generic` returns to the program.

Usually, `Generic` will be used to execute a register-based Toolbox trap. In such cases, the value you pass to `Generic` via the `Instr` argument is the trap value.

The `Instr` argument to `Generic` does not have to be a trap value, it can be any 16-bit MC68000 instruction.

Note: The `Generic` procedure is provided only for compatibility with Macintosh Pascal. THINK Pascal provides direct access to the Macintosh toolbox as well as the ability to call assembly language routines. Therefore, the only need for `Generic` is for porting Macintosh Pascal programs which used this routine.

10.9.12 The Macsbug Support Procedures`Debugger``DebugStr (message)`

`Message` a string-type value.

`Debugger` and `DebugStr` cause a User Break which interrupts the execution of the program and transfers control to the Macsbug low-level debugger. If `DebugStr` is used, the message argument is displayed.

If Macsbug is not installed, then calls to `Debugger` and `DebugStr` give the run time error message "Macsbug/TMON not installed."

10.9.13 Member Function`Member (r, t)`

`r` an object-reference expression.

`t` a type-identifier of a reference type.

`Member` returns `true` if `r` references an object of type `t`.

10.9.14 The Time-of-Compilation Functions`comdate``comptime`

Returns string-type

These return the day and time that the enclosing file was compiled. `Comdate` returns the date in the form `mm/dd/yy` (e.g., `8/24/90`). `Comptime` returns the time in the form `hh:mm:ss XM` (e.g., `11:05:35 AM`).

THINK PascalTM

P A R T F I V E

Utilities

- 18 Project Utilities
- 19 The Profiler
- 20 The Pascal Source Converter
- 21 Resource Description Files
- 22 Using SAREz
- 23 Using SAdRez
- 24 Using SAPostRez

Project Utilities

18

Introduction

This chapter describes Project Utilities, a utility program that backs up and prints files from THINK Pascal projects faster than you could from the Finder or THINK Pascal. You can back up all the files in your project, including libraries and resource files, to another disk to guard against crashes. You can print out the files in your project in succession, one after the other. Project Utilities lets you select which files to back up or print, according to file type, creation date, or some other criteria.

Topics covered in this chapter

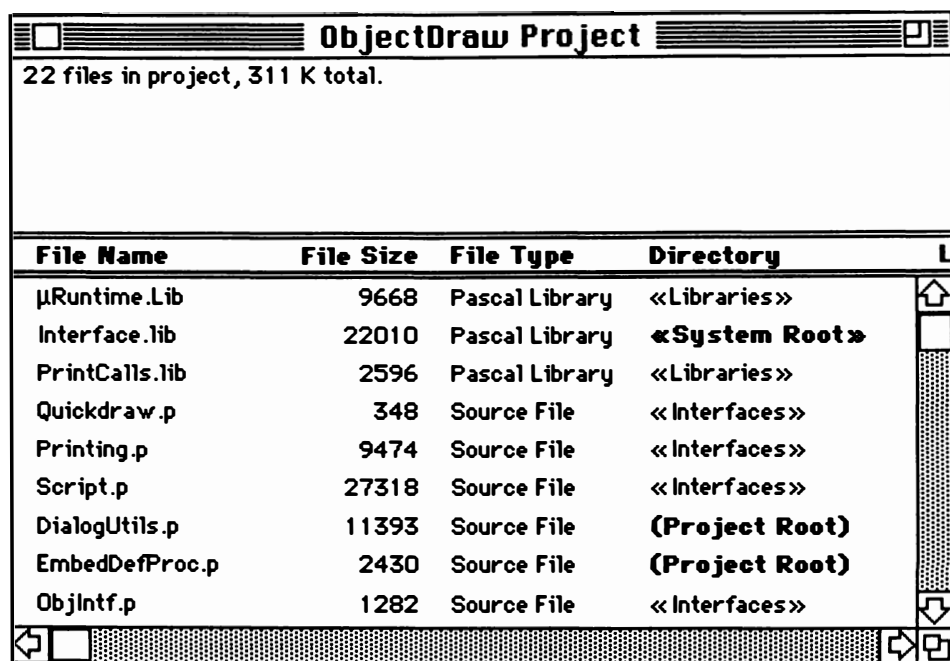
- Using Project Utilities
- Selecting files
- Printing files
- Backing up files

Using Project Utilities

If you installed THINK Pascal according to the instructions in Chapter 2, "Installing THINK Pascal," Project Utilities is in the THINK Pascal 4.0 Folder.

Note: Project Utilities must be in the same folder as THINK Pascal. Otherwise, it will not be able to find all the files in a project.

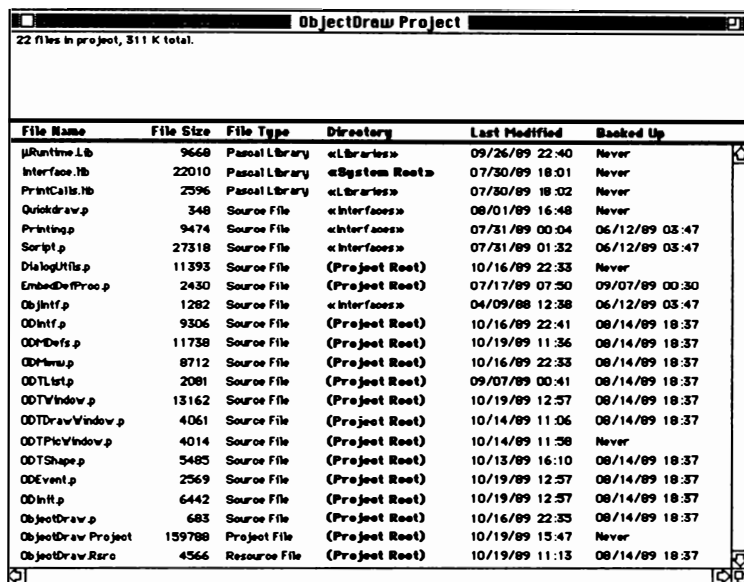
To start Project Utilities, double-click on its icon. Choose a project from the standard file dialog that Project Utilities displays. You'll see a Project Utilities project window like this:



The screenshot shows a window titled "ObjectDraw Project". Below the title bar, it says "22 files in project, 311 K total." Below this is a table with four columns: "File Name", "File Size", "File Type", and "Directory". The table lists several files, including libraries and source files. On the right side of the table, there are navigation icons: a home icon, a square icon, a scroll bar, and a list icon.

File Name	File Size	File Type	Directory
μRuntime.Lib	9668	Pascal Library	«Libraries»
Interface.lib	22010	Pascal Library	«System Root»
PrintCalls.lib	2596	Pascal Library	«Libraries»
Quickdraw.p	348	Source File	«Interfaces»
Printing.p	9474	Source File	«Interfaces»
Script.p	27318	Source File	«Interfaces»
DialogUtils.p	11393	Source File	(Project Root)
EmbedDefProc.p	2430	Source File	(Project Root)
ObjIntf.p	1282	Source File	«Interfaces»

The Project Utilities window displays the name, size, type, directory, last date modified, and last date backed up for every file in your project. If Project Utilities can't find a file, it displays its name in italics and writes "File not found" in the Directory field. To see the fields beyond the right edge of the window, use the scroll bar at the bottom of the window. If you have a large screen, you can resize the window so it looks like this:



22 files in project, 311 K total.

File Name	File Size	File Type	Directory	Last Modified	Backed Up
µRuntime.Lib	9668	Pascal Library	<Libraries>	09/26/89 22:40	Never
Interface.Lib	22010	Pascal Library	<System Roots>	07/30/89 18:01	Never
PrintCalls.Lib	2596	Pascal Library	<Libraries>	07/30/89 18:02	Never
QuickDraw.p	348	Source File	<Interfaces>	08/01/89 16:48	Never
Printing.p	9474	Source File	<Interfaces>	07/31/89 00:04	06/12/89 03:47
Script.p	27318	Source File	<Interfaces>	07/31/89 01:32	06/12/89 03:47
DialogUtils.p	11393	Source File	(Project Root)	10/16/89 22:33	Never
EmbedDefProc.p	2430	Source File	(Project Root)	07/17/89 07:30	09/07/89 00:30
ObjInit.p	1282	Source File	<Interfaces>	04/09/88 12:38	06/12/89 03:47
ObjInit.p	9306	Source File	(Project Root)	10/16/89 22:41	08/14/89 18:37
ObjDef.p	11738	Source File	(Project Root)	10/19/89 11:36	08/14/89 18:37
ObjMenu.p	8712	Source File	(Project Root)	10/16/89 22:33	08/14/89 18:37
ObjTLst.p	2081	Source File	(Project Root)	09/07/89 00:41	08/14/89 18:37
ObjVWindow.p	13162	Source File	(Project Root)	10/19/89 12:37	08/14/89 18:37
ObjDrawWindow.p	4061	Source File	(Project Root)	10/14/89 11:06	08/14/89 18:37
ObjPictWindow.p	4014	Source File	(Project Root)	10/14/89 11:06	Never
ObjShape.p	5485	Source File	(Project Root)	10/13/89 16:10	08/14/89 18:37
ObjEvent.p	2569	Source File	(Project Root)	10/19/89 12:37	08/14/89 18:37
ObjInit.p	6442	Source File	(Project Root)	10/19/89 12:37	08/14/89 18:37
ObjectDraw.p	683	Source File	(Project Root)	10/16/89 22:33	08/14/89 18:37
ObjectDraw Project	159788	Project File	(Project Root)	10/19/89 15:47	Never
ObjectDraw.Rsrc	4566	Resource File	(Project Root)	10/19/89 11:13	08/14/89 18:37

You can open several projects at once. To open another project, choose **Open...** from the **File** menu and select the project. To close the project in the front window, choose **Close** from the **File** menu.

Reading the display

Most of the fields in the Project Utilities project window are self-explanatory. The File Size field displays the size of the file in bytes. The Last Modified field displays the date when the file was last modified. The Backed Up field displays the date the file was last backed up or "Never" if it was never backed up.

The other fields need a little more explanation. The File Type field displays the type of the file. It can be one of these:

File Type	Description
Source File	Any text file or source file <i>except</i> a THINK Pascal source file saved as Entire Document
Pascal Document	A THINK Pascal source file saved as Entire Document
Pascal Library	A library created with THINK Pascal
C Library	A library created with THINK C

File Type	Description
MPW Object File	An object file (.o file) created with MPW
Project File	A THINK Pascal project file
Resource File	A resource file created with ResEdit or Rez

The Directory field displays the folder that contains the file. If the file is in the THINK Pascal tree, the folder name is in angle quotes («»). If it's in the project tree, the folder name is in parentheses (()). If it's in neither tree, Project Utilities lists the full pathname .

Note: If you're not sure what the THINK Pascal and project trees are, see "Organizing Your Files" in Chapter 7.

This really isn't as confusing as it may sound. This chart explains what the entries mean:

If it looks like...	the file is in...	For example:	means the file is in
«file-name»	the THINK Pascal tree	«Libraries»	the Libraries folder in the THINK Pascal folder
(file-name)	the project tree	(MyLibs)	the MyLibs folder in your project's folder
folder:file-name	neither tree	HD20:Math	the Math folder on the disk HD20
«System Root»	the same folder as THINK Pascal		
(Project Root)	the same folder as your project		

Printing the project window

To print out the contents of the Project Utilities project window, choose **Print Project...** from the **File** menu. This command displays the standard Print dialog and prints the window.

Saving a list of files

To save a list of the full pathnames of the files in your project, choose **Write Project Listing...** from the **Project** menu. This displays a standard save dialog that lets you choose where to save the listing.

Selecting Files

Project Utilities lets you select any number of files to back up or print. You can choose them yourself or use the commands in the **Project** menu. You can select files in Project Utilities project window the same way you do in the Font/DA Mover:

To select...	Do this...
A single file	Click on the file name
A range of files	Hold down the Shift key and click on a file name
Many individual files	Hold down the Command key as you click on each file

Selecting files by type

To select all the files of a certain type, use the **Select By File Type** command. It shows a hierarchical menu that lets you select a type. See "Reading the display" above for an descriptions of the file types.

Selecting files by folder

To select all the files from a certain folder or tree, use the **Select Files From** command. It shows a hierarchical menu that lets you select one of these:

•THINK Pascal Tree•	Files in the THINK Pascal tree.
(Project Tree)	Files in the project tree.

The hierarchical menu also has entries for each folder that contains a file in the project.

Selecting files by ending

To select all the files that end in a certain suffix, use the **Select Files Ending In** command. It shows a hierarchical menu that lets you select one of these:

.p	THINK Pascal source files (including files saved as Entire Document)
.c	THINK C source files
.h	THINK C header files
.o	MPW object files
.Lib	THINK C or THINK Pascal libraries

Selecting files by date

To select all the files that were modified after a certain date, use the **Select By Date...** command. It displays this dialog:

Select files modified after:

October 31 1990

12 00 00

OK Cancel

Project Utilities initially displays the current date and time. To change the time, click on the arrows to the left of each item. The longer you hold down the mouse, the faster the item changes. The time selected here is 12 noon on October 31, 1990.

Selecting files for backup

To select all the files that need to be backed up, use the **Select If Backup Needed** command. It selects all the files that have never been backed up or that have been modified since they were last backed up.

Printing Files

Project Utilities can print several text files in succession, one after the other.

Note: Project Utilities cannot print THINK Pascal documents saved as Entire Document. You can print these from THINK Pascal or you can open them in THINK Pascal and save them again with the Entire Document option unchecked. Also, Project Utilities does not "pretty-print" files like THINK Pascal does. It does not print Pascal keywords in bold face and may indent some lines differently.

To print all the selected text files, choose **Print Selected Files...** from the **File** menu. You'll see the standard Print dialog with a few extra options:

ImageWriter v2.7 OK Cancel

Quality: ☐ Best ☒ **Faster** ☐ Draft

Page Range: ☒ **All** ☐ From: To:

Copies:

Paper Feed: ☒ **Automatic** ☐ Hand Feed

☒ **Print Pages In Reverse Order %R**

☐ **Frame Printing Area %F**

☒ **Print Page Headers %H**

Printing Font: Geneva-9

Time Stamp:

☒ **File's Modification Date %M** ☐ **Date of Printing %P**

Here's what the extra options mean:

Print Pages in Reverse Order	Prints the pages from last to first, instead of first to last.
Frame Printing Area	Prints a border around each page.
Print Page Headers	Prints a header on each page, containing the page number, the file name, and a time stamp. You can choose what time to print with the Time Stamp option described below.

Printing Font

Displays a dialog that lets you select the font, type size, and style used for printing. The default is the font and size specified in the THINK Pascal **Source Options...** dialog. To change it, select a font and size from the pop-up menus. To use a size that's not in the menu, type it into the box. Change the font style with the check boxes. The box outlined in grey contains a sample of what a line of your code will look like. For example, this dialog is set up for nine point Geneva, plain:

Font:

Size:

☒ Plain %P ☐ Bold %B ☐ Outline %O
☐ Condensed %N ☐ Italic %I ☐ Shadow %S
☐ Extended %E ☐ Underline %U

`hPE''.tabWidth := 4 * CharWidth(CHR($OD));`

OK Cancel %.

Time Stamp

Lets you choose which time is printed in the page headers (if you have Print Page Headers selected). These are the choices:

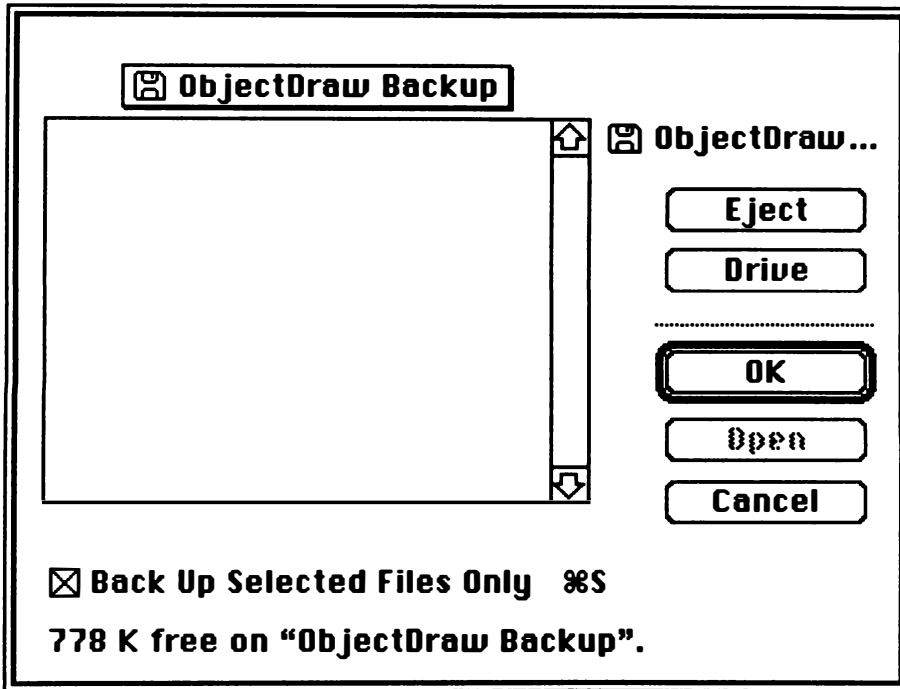
- **File's Modification Date.** The date the file was last modified
- **Date of Printing.** The date you print the file.

Backing Up Files

Project Utilities lets you back up your project's files to another folder, disk, or a set of floppy disks.

Note: Project Utilities works in the background under MultiFinder. While backing up, you can switch to another application or move and resize the project window. However, you cannot close the project window or bring another Project Utilities window to the front.

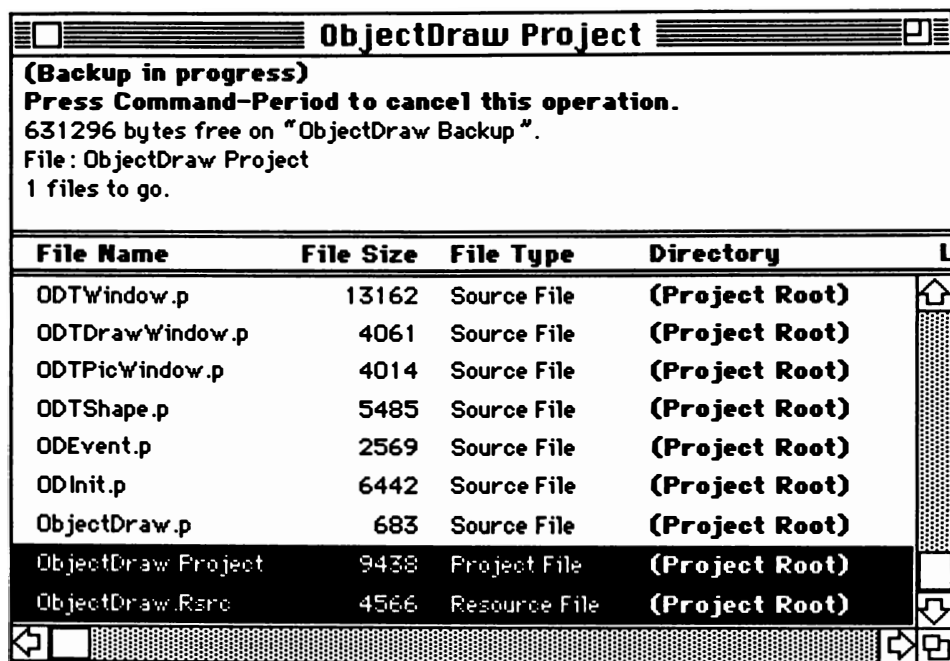
To start, select all the files you want to back up. Usually, you'll want to choose **Select If Backup Needed** to select only the files that have never been backed up or that were modified since you last backed them up. Then choose **Back Up...** from the **Project** menu. You'll see this dialog:



Project Utilities lets you save a back up to any device you can use in the Finder, such as a hard drive or floppy disks. The **Back Up...** dialog works pretty much like a standard file dialog box. The message at the bottom of the dialog tells you how much space is free on the device selected. If **Back Up Selected Files Only** is checked, Project Utilities will back up only the files you checked. Otherwise, it will back up all the files in the project. The **OK** button (or pressing Command-Enter) tells Project Utilities to go ahead and do the back up. The **Open** button lets you open a folder. If you select a folder and click on the **OK** button, Project Utilities will place your files in that folder.

Note: You cannot back up your files to the same folder that your project is in. And to be certain your backup is secure in case of a disk crash, you should back up your files to a different disk.

As Project Utilities backs up your project, it displays its progress in the top of the project window, like this:



As Project Utilities copies a file, it deselects it and notes the back up date and time in the file. You can cancel the back up at any time by pressing Command-. (Command-Period).

If you're backing up to a floppy, Project Utilities displays a dialog box asking you to insert another disk when the first one is full.

If the destination disk or folder already has a file with the same name as one of the files in your back up, Project Utilities asks you what to do:

“DialogUtils.p” already exists on the destination. Do you want to cancel the backup, replace the existing file, or skip this file?

Cancel %.

Skip %S

Replace %R

☐ **Don't Ask Again %D**

You can choose to skip the file, replace the file, or cancel the back up. If you check the box labeled “Don’t Ask Again,” Project Utilities will do the same thing whenever it comes across a file that’s already in the destination. For example, if you check “Don’t Ask Again” and click on the Replace File button, Project Utilities replaces files in the destination whenever it finds duplicate names.

The Profiler

19

Introduction

This chapter shows you how to use the THINK Pascal code profiler. The profiler collects statistics about your program, including the time spent in each routine.

Topics covered in this chapter:

- Using the profiler
- Reading the report
- Summary

Using the Profiler

To use the profiler in your project, follow these steps:

- Turn on the Profile option in the **Compile Options...** dialog.
- Turn on the Debug and Names options (using the project window or the Debug and Names directives) for every routine you want measured. If the Debug option is off for a routine, the profiler will not collect statistics for it. If the Names option is off for a routine, the profiler won't list its name but ???????? instead.

The profiler records information for every function and procedure call. It collects statistics for up to two hundred routines. If there are any more, it lumps their statistics together under the name `catchall`. It can handle a calling chain of up to two hundred routines. If the calling chain becomes longer (which is possible in a heavily recursive program), the profiler will stop collecting statistics. When your program exits, the profiler writes a report to the Text Window. To see a sample report, see "Reading the Report," below.

You can control how the profiler profiles your code. For example, you can increase the number of routines it collects statistics for or print out a report before your program exits. To call the routines that control the profiler, follow these steps:

- Include the file `Profile.p` in your project. If you installed THINK Pascal according to the instructions in Chapter 2, "Installing THINK Pascal," it should be in your THINK Pascal 4.0 Folder
- Add `Profiler` to the **uses** clause of every unit and program that calls profiler routines.

These are the routines control the profiler. For detailed descriptions, see "Summary," below.

- To collect statistics for more than two hundred routines, call `InitProfiler`. It takes an argument specifying how many routines to collect statistics for, up to 32,768.
- To stop collecting statistics temporarily, set `%_PTrace` to `FALSE`. To start collecting statistics again, set `%_PTrace` to `TRUE`.
- To print out the statistics collected so far, call `DumpProfile` or `DumpProfileToFile`. `DumpProfile` prints the statistics in the Text Window. `DumpProfileToFile` takes an argument specifying a file to print to.
- To reinitialize the profiler without stopping it, call `ResetProfile`.
- To stop the profiler, call `TerminateProfile`.

Reading the Report

The profiler logs the time spent and the number of statements executed for each routine and prints the results to the Text Window. This report was produced with the Long Names on:

THINK Pascal Procedure Profile

All time is measured in milliseconds.
rounded down to the nearest millisecond.

An asterisk (*) next to a routine name indicates that that routine was left via a nonlocal GOTO, EXIT to an uplevel routine, or "longjmp"-style context restoration.

```
Elapsed Time   =    289945
Measured Time  =   1027888
Total Calls    =     5052
```

Routine Name	Min Time	Max Time	Avg Time	Total Time	% Time	Times Called
SORTING	312619	312619	312619	312619	30.41	1
QUICKSORT	1	39373	129	436725	42.49	3367
PARTITION	4	159	6	11282	1.10	1683
BUBBLESORT	267262	267262	267262	267262	26.00	1

Note: For more information on the Long Names option, see "Names { \$N± }," Chapter 15.

This describes the report's headings:

Heading	Description
Min Time	The minimum time spent in
Max Time	The maximum time spent in the routine.
Avg Time	The average time spent in the routine.
Total Time	The total time spent in the routine.
% Time	The percentage of program's time spent in the routine.
Times Called	The number of times the routine was called.

Time is measured in milliseconds (thousandths of a second). If a function did not exit normally and aborted in one of the following ways, the profiler puts an asterisk by its name:

- Called `exit`
- Used a non-local `goto`.

Summary

These procedures control the profiler:

function `InitProfiler (nEntries: integer): Boolean;`

This lets you specify how many routines you want statistics for, up to 32,768. It must be the first statement in your program. You need to call this routine only if you want to collect statistics for more than 200 routines. When you turn on the Profile option, THINK Pascal automatically initializes the profiler and reserves space for 200 routines.

This routine returns `true` if it can initialize the profiler and `false` if it can't. If this function can't initialize the profiler, you're probably asking it to collect statistics for too many routines. Each function or procedure you collect statistics for takes up 94 bytes, and each function or procedure call takes up 14 bytes.

procedure `DumpProfile;`

This dumps the current statistics about your program to the Text Window.

procedure `DumpProfileToFile (fileName: Str255);`

This dumps the current statistics to the file `fileName` in the default volume. If the file already exists, the profiler will delete and overwrite it. The default volume is usually the folder your project is in, unless you've changed it with `SetVol`.

procedure `ResetProfile;`

This reinitializes the profiler and its statistics. Use this to collect two different sets of statistics in the same run of your program.

procedure `TerminateProfile;`

This stops the profiler. After you call it, the profiler collects no more statistics.

```
var %_PTrace: Boolean;
```

This lets you turn off the profiler over certain parts of your program. To stop collecting statistics, set %_PTrace to FALSE. To start collecting statistics again, set %_PTrace to TRUE.

The Pascal Source Converter

20

Introduction

The Pascal Source Converter is an application that helps you translate programs written for Apple's MPW Pascal for use with THINK Pascal. The Pascal Source Converter:

- converts MPW Pascal directives to THINK Pascal directives,
- processes MPW Pascal \$I include directives,
- processes \$IFC directives,
- converts nested comments into unnested comments,
- resolves **uses** clauses into a form compatible with THINK Pascal, and
- converts source code according to a 'differences' file.

The Pascal Source Converter was initially designed to convert the MacApp source files so they can be used with THINK Pascal. Since the converter is script driven, you can use it to convert programs that you've already written for MPW Pascal so you can use them in THINK Pascal. The Pascal Source Converter is in the MacApp 2.0 for THINK Pascal 4.0 folder. For information about installing the Pascal Source Converter on your disk, see Chapter 2, "Installing THINK Pascal."

What you should know

You should be familiar with THINK Pascal and with MPW Pascal. Particularly, you should know how the MPW Pascal compiler directives work.

Topics covered in this chapter

- Scripts
- Input and output directories
- Specifying directories and file names
- Converting Files
- Pascal Source Converter directives
- A sample script

Scripts

The Pascal Source Converter uses a **script** to specify which files need to be converted and to control how a file should be converted. Scripts are files of type TEXT. By convention, scripts end with the suffix **.Script**.

Note: If you want the Pascal Source Converter to open scripts automatically when you double-click on them, set the creator to **MACv**.

A Pascal Source Converter script consists of several directives. Pascal Source Converter directives look just like regular Pascal compiler directives. The converter processes these directives in scripts:

\$BEEP	\$ENDC	\$OUTPUT
\$CREATOR	\$IFC	\$PORTABLE
\$DEPEND	\$INCLUDES	\$SETC
\$DIFFS	\$INPUT	\$SKIP
\$DIR	\$LINE	\$USES
\$ELSEC	\$NOUSES	

To learn how to write a script, you might want to look at the file `MacApp.Script` in the folder `MacApp 2.0` for THINK Pascal 4.0. This is the script that the Pascal Source Converter uses to convert the Apple MacApp source files into a form that THINK Pascal can use.

Input and Output Directories

The Pascal Source Converter processes either one file or every file in a specified directory and places the converted file in an output directory. Use the `$OUTPUT` directive to specify the directory that will receive the converted file. Use the `$INPUT` directive to specify the file or directory that you want to convert. All of your scripts will have lines that look like this:

```
{ $OUTPUT directory }
{ $INPUT directory or filename }
```

The `$OUTPUT` directive must come before the `$INPUT` directive. Both directories must already exist. The next section tells you how you can have a user create a directory from a script.

Note: The input directory must not be the same as the output directory.

Specifying Directories and File Names

In Pascal Source Converter scripts, directory names and file names are strings that contain the full path to the file or directory. You can set a compile-time variable to a string that contains a path name and use it to modify a directory string.

```
{ $SETC RootDir = 'MyDisk:Master Folder:' }
{ $SETC InputFolder = (RootDir) 'SourceFolder:' }
{ $SETC OutputFolder = (RootDir) 'DestinationFolder:' }

{ $OUTPUT OutputFolder }
{ $INPUT InputFolder }
```

To use the value of a compile-time variable to modify a file name or directory, enclose the variable name in parentheses, and place it before the quoted part of the string. Note that the folder names end with a colon.

The Pascal Source Converter can prompt the user to supply a directory name or a file name. If the string begins with a special character, the Pascal Source Converter displays a dialog that lets the user choose a file or folder. The converter uses the rest of the string as a prompt.

Character	Prompt the user for...
\$	an existing directory
#	a new directory name
?	an existing file name

For instance, these directives let the user choose an existing input directory and create a new output directory:

```
{ $SETC InputDir = '$Please find the input directory' }
{ $SETC OutputDir = '#Please create a new directory' }
{ $OUTPUT OutputDir }
{ $INPUT InputDir }
```

Converting Files

Once you've specified the input and output directories, the Pascal Source Converter converts every file in that directory and places a converted copy of the file in the output directory. The Pascal Source Converter is a non-destructive converter. It won't alter the original file. However, you should not set the input directory to be the same as the output directory.

\$IFC Conditional compilation directives

The Pascal Source Converter processes conditional compilation (\$IFC) directives. If it encounters a compiler variable that is defined either in the converter script or in the source file, the effect is the same as conditional compilation. For instance the converter turns these lines:

```
{ $SETC WeAreTesting = TRUE }

procedure MyProc (a: Integer);
begin
    someGlobal := a;
    { $IFC WeAreTesting }
    writeln('The value of a is: ', a:1);
    { $ENDC }
end;
```

into this:

```
procedure MyProc (a: Integer);
begin
    someGlobal := a;
    writeln('The value of a is: ', a:1);
end;
```

The `$SETC` directive can appear either in the converter script or in the source file. The Pascal Source Converter never processes `$SETC` directives that appear in the implementation part of a unit.

If a compiler variable is undefined at the time that the Pascal Source Converter encounters it, the converter leaves the lines alone. The Pascal Source Converter never converts any `$IFC` directives of the form `{ $IFC NOT UNDEFINED THINK_Pascal }` or `{ $IFC UNDEFINED THINK_Pascal }`. All of the code to the matching `$ENDC` is sent directly to the output file.

Note: THINK Pascal defines `THINK_Pascal` as `TRUE` when you're in the THINK Pascal environment. Your script and your program should **never** define `THINK_Pascal`.

\$I Include file directives

When the Pascal Source Converter encounters an MPW Pascal `$I` (include file) directive, it either includes the contents of the named file to the file that contains the directive, or it adds the file to the **uses** clause.

If the `$I` directive contains the name of a file qualified by an MPW Shell variable, the `$I` directive is replaced by the contents of the file. For instance, if the Pascal Source Converter encounters a directive like this:

```
{ $I $$SHELL(MAPInterfaces) UMacApp.p }
```

the contents of `UMacApp.p` are inserted where the directive appears.

Note: You can use the `$SETC` directive to set the value of an MPW Shell variable.

If the `$I` directive isn't qualified by an MPW Shell variable and the `$I` directive is in the interface part of a unit, the file name is converted to a unit name and it's appended to the end of the **uses** clause. For instance, the Pascal Source Converter converts this fragment:

```
unit UMacApp;
interface

    uses SysEqu, UMacAppUtilities;

    { $I UAssociation.p }
    { $I UList.p }

implementation

    { more code here }

end.
```

to this:

```

unit UMacApp;
interface

    uses SysEqu, UMacAppUtilities, UAssociation, UList;

implementation

    { more code here }

end.

```

If the \$I directive is not qualified but does not appear in the interface part of a unit, the converter treats it the same way as it does qualified \$I directives.

Compiler directives

The Pascal Source Converter converts some MPW Pascal compiler directives into equivalent THINK Pascal compiler directives. The Pascal Source Converter also removes MPW Pascal compiler directives that THINK Pascal does not support. The Pascal Source Converter can convert equivalent compiler directives in either a portable or non-portable way depending on the setting of the \$PORTABLE converter directive. For example, if your script includes the directive \$PORTABLE+, this directive:

```
{ $D+ }
```

gets converted to:

```

{$IFC UNDEFINED THINK_Pascal}
{$D+}    MPW Pascal directive to turn on Macsbug names
{$ELSEC}
{$N+}    THINK Pascal directive to turn on Macsbug names
{$ENDC}

```

If the Pascal Source Converter is converting a main program instead of a unit, it automatically supplies the THINK Pascal \$I- directive to suppress Macintosh Toolbox initialization at the beginning of the file. If the \$PORTABLE+ directive is on, the converter adds these lines to the beginning of the file:

```

{$IFC NOT UNDEFINED THINK_Pascal}
{$I-}
{$ENDC}

```

Comments

The Pascal Source Converter converts nested comments (which MPW Pascal allows) into unnested comments. For instance:

```
{ This comment {contains a comment} inside it }
```

becomes:

```
{ This comment ( * contains a comment * ) inside it }
```

Note: The converter places a space between the parentheses and the asterisks.

Differences

Sometimes, it's just not possible to convert a program from MPW Pascal to THINK Pascal without altering the source directly. If you're working with someone else's MPW Pascal code, you may not be able to simply distribute translated code. The Pascal Source Converter contains a mechanism that applies a list of specific differences to the original MPW Pascal file to create a new THINK Pascal file.

Note: If you're using the Pascal Source Converter to convert your own files, you won't need to use this feature. It's primarily designed for those who need to convert MPW Pascal source code without actually distributing modified code.

When the Pascal Source Converter opens a file either through an \$INPUT directive or in the course of processing a \$I directive, it looks in the differences folder (specified with a \$DIFFS directive) for a folder with the same name as the folder that contains the file. It then looks within that folder for a file with the same name as the file it's processing, but with .Diff appended to it. If it finds one, it merges the changes into the resulting file.

For example, assume that you used the \$DIFF directive to tell the converter that the differences folder is the folder AllDiffs. As the Pascal Source Converter processes a file called MySource.p (which resides in a folder called MyFolder), it looks in the folder AllDiffs for a folder called MyFolder. Then it looks in MyFolder for a file called MySource.p.Diffs. If it finds it, it applies the changes to the source file.

To create a .Diff file, you need to use the MPW Compare tool and give it this command:

```
Compare newfile oldfile > newfile.Diffs
```

Note: The order of the files is **extremely** important!

Pascal Source Converter Directives

This section lists the directives you can use in Pascal Source Converter scripts. Don't forget that you can use these directives in source files as well.

```
{ $BEEP }
```

Beep the speaker. This directive is useful when you're debugging your script.

```
{ $CREATOR 'xxxx' }
```

Set the creator of converted files to 'xxxx'. The default is 'PJMM'.


```
{ $DEPEND unit-name unit1 unit2 ... }
```

Specify dependencies among units. As the converter processes MPW Pascal \$I directives, it may add unit names to the **uses** clause of units. These units may in turn require other units.

For instance, the MPW file ULoMem.p uses \$I directives to include SysEqu.p and Devices.p which the Pascal Source Converter converts to unit names in a **uses** clause. Because ULoMem uses symbols from SysEqu and Devices in its interface part, any unit that uses ULoMem also needs to use SysEqu and Devices. To make sure that the converter knows this you need a line like this in your script:

```
{ $DEPEND ULoMem SysEqu Devices }
```

You can read this directive as "ULoMem uses SysEqu and Devices, so any unit that uses ULoMem should also use SysEqu and Devices."

So the MPW Pascal line:

```
uses ULoMem;
```

is converted to the THINK Pascal line

```
uses SysEqu, Devices, ULoMem;
```

\$DEPEND applies to the next \$INPUT directive.

```
{ $DIFFS directory }
```

Specify the directory that contains the folders that contain .Diff files.

```
{ $DIR directory }
```

Set the default directory for unqualified file names to be *directory*. An unqualified file name doesn't include any directory information. For instance when the Pascal Source Converter processes these lines,

```
{ $DIR 'MyDisk:PascalFile:' }
{ $INPUT 'MyFile.p' }
```

it will look for the file MyFile.p in MyDisk:PascalFile:.

```
{ $IFC condition }
```

```
{ $ELSEC }
```

```
{ $ENDC }
```

These directives work the same way as the Pascal compiler directives. If a compiler variable is undefined, the Pascal Source Converter leaves the entire conditional (up until the closing \$ENDC) alone. The Pascal Source Converter also leaves alone directives of the form \$IFC [NOT] UNDEFINED THINK_PASCAL.

```
{ $INCLUDES directory }
```

This directive tells the Pascal Source Converter where to find include files in unqualified \$I directives. An unqualified \$I directive looks like this:

```
{ $I AnIncludeFile.p }
```

Qualified \$I directives use an MPW Shell variable to specify their directory.

The \$INCLUDES directive applies to the next \$INPUT directive. For instance, in this script fragment:

```
{ $OUTPUT SomeDirectory }
{ $INPUT SourceDirectory }
{ $INCLUDE TheIncludes }
{ $INPUT AnotherSource }
```

the Pascal Source Converter will look for include files in the directory `TheIncludes` only for the files in `AnotherSource`.

```
{ $INPUT directory-or-filename }
```

Process all the files in the specified directory or the specified file. After processing the file, the Pascal Source Converter places the converted version of the file in the output directory. Your script must have at least one \$INPUT directive after an \$OUTPUT directive. For instance, this script:

```
{ $OUTPUT OutputDir }
{ $INPUT 'MyDisk:SomeFolder:MyFile.p' }
{ $INPUT 'MyDisk:AnotherFolder:' }
{ $INPUT SomeFolder }
```

will convert the file `MyFile.p`, all the files in the folder `AnotherFolder`, and all the files in the directory specified in the script variable `SomeFolder` into the folder specified by `OutputDir`.

```
{ $LINE 'string' }
```

Insert the string into the output file. This directive is useful when you're debugging scripts.

```
{ $NOUSES unit-name unit1 unit2 ... }
```

When the Pascal Source Converter encounters the unit *unit-name*, it removes the specified units from the **uses** clause. If you supply an asterisk (*) as the *unit-name*, the Pascal Source Converter removes the specified units from every **uses** clause. \$NOUSES applies to the next \$INPUT directive.

```
{ $OUTPUT directory}
```

Set the output directory. The output directory is where the Pascal Source Converter places the converted files. The \$OUTPUT directive specifies where to place the files processed by the next \$INPUT directive. This means that an \$OUTPUT directive must precede an \$INPUT directive. If you specify another \$OUTPUT directive, it will apply to the files processed by the next \$INPUT directive. For instance, in this script:

```
{ $OUTPUT FirstDir}
{ $INPUT Source1Dir}
{ $INPUT Source2Dir}
{ $OUTPUT SecondDir}
{ $INPUT Source3Dir}
```

the Pascal Source Converter places the converted files from Source1Dir and Source2Dir in FirstDir and the converted files from Source3Dir in SecondDir.

```
{ $PORTABLE±}
```

Determines how the converter process the MPW Pascal compiler \$D (Macsbug names) directive. If your script contains the directive \$PORTABLE+ the converter changes the directive so you can use the source file with either THINK Pascal or MPW Pascal. For example, if your script includes the directive \$PORTABLE+, this directive

```
{ $D+}
```

gets converted to

```
{ $IFC UNDEFINED THINK_Pascal}
{ $D+} { MPW Pascal directive to turn on Macsbug names }
{ $ELSEC}
{ $N+} { THINK Pascal directive to turn on Macsbug names }
{ $ENDC}
```

If the Pascal Source Converter is converting a main program instead of a unit, it automatically supplies the THINK Pascal \$I- directive (suppress Macintosh Toolbox initialization) at the beginning of the file. If the \$PORTABLE+ directive is on, the converter adds these lines to the beginning of the file:

```
{ $IFC NOT UNDEFINED THINK_Pascal}
{ $I-}
{ $ENDC}
```

The \$PORTABLE directive applies to the next \$INPUT directive. For example:

```
{ $PORTABLE-}
{ $INPUT NonPortableSourceDir}
...
{ $PORTABLE+}
{ $INPUT PortableSourceDir}
```

The files in `NonPortableSourceDir` will have all MPW Pascal `$D` directives converted to THINK Pascal `$N` directives, but the files in `PortableSourceDir` will have the conditional compilation wrapper so you can use the source file with either compiler.

The default is `{ $PORTABLE- }`.

```
{ $SETC compiler-variable = value }
```

This directive works just like the regular Pascal `$SETC` directive. It sets the specified compiler variable to the specified value. In Pascal Source Converter scripts you can set a compiler variable to a string.

To prompt the user for a directory name or a file name, use one of these three special characters as the first character of a string. The converter uses the rest of the string as a prompt.

Character	Prompt the user for...
\$	an existing directory
#	a new directory name
?	an existing file name

For example, you can use this line to prompt for a folder:

```
{ $SETC IncludesDir = '$Which folder contains the include files?' }
```

```
{ $SKIP filename }
```

Skip the specified file when processing the current directory. The file is not converted to the output directory. The `$SKIP` directive applies to the previous `$INPUT` directive.

```
{ $USES unit name unit1 unit2 ... }
```

When the Pascal Source Converter encounters the unit `<unit name>`, it adds the specified units to the `uses` clause. The Pascal Source Converter takes care to avoid placing duplicate names in `uses` clauses. `$USES` applies to the next `$INPUT` directive.

A Sample Script

Here is a small script that converts all the files in one directory. For a more complex scripts, see the file `Generic.Script` and `MacApp.Script` in the MacApp 2.0 for THINK Pascal 4.0 Folder.

```
{ Convert all the files in one directory }
{ $SETC InputDir = '$Which folder contains the original sources?' }
{ $SETC OutputDir = '#Name a folder for the converted sources.' }
{ $SETC IncludeDir = '$Which folder contains the include files?' }

{ $PORTABLE+ }
{ $INCLUDES IncludeDir }
{ $OUTPUT OutputDir }
{ $INPUT InputDir }
```

Resource Description Files

21

Introduction

This chapter describes resource description files, the files compiled by SAREz and produced by SADeRez. See Chapters 22, "SAREz," and 23, "SADeRez," for more information on these utilities. Complete background information on Macintosh resource files is given in the *Inside Macintosh I*, Chapter 5, "The Resource Manager," and *Inside Macintosh VI*, Chapter 13, "The Resource Manager."

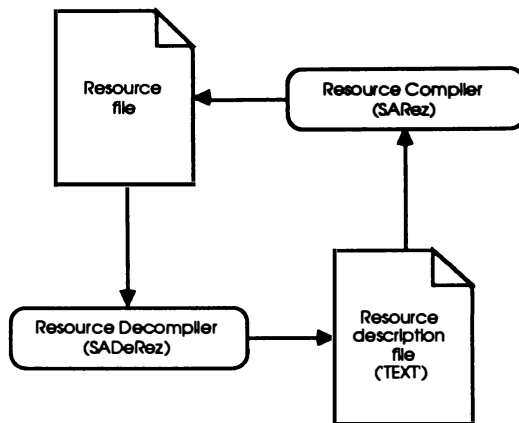
Both SAREz and SADeRez started out life as tools in Apple's Macintosh Programmer's Workshop (MPW), where they were known as Rez and DeRez. MPW has a command-line interface, much like UNIX. At times it will seem these utilities work a bit oddly or have options you can't use. This is because of their heritage. But, despite their upbringing, they are still powerful and useful utilities.

Topics covered in this chapter

- The resource compiler and decompiler
- Structure of a resource description file
- Resource description statements
- Labels
- Preprocessor directives
- Resource description syntax

The Resource Compiler and Decompiler

The resource compiler, SAREz, compiles a text file (or files) called a **resource description file** and produces a resource file as output. The resource decompiler, SAdRez, decompiles an existing resource, producing a new resource description file that can be understood by SAREz. This picture illustrates the complementary relationship between SAREz and SAdRez.



SAREz can combine resources or resource descriptions from a number of files into a single resource file. SAREz can also delete resources and change resource attributes. SAREz has preprocessor directives that let you substitute macros, include other files, and use if-then-else constructs. (These directives are described under the heading "Preprocessor Directives" later in this chapter.)

Resource decompiler

SAdRez creates a textual representation of a resource file based on resource type declarations identical to those used by SAREz. (If you don't specify any type declarations, the output of SAdRez takes the form of raw data statements.) The output of SAdRez is a resource description file that may be used as input to SAREz. This file can be edited in any text editor, so you can add comments, translate resource data to a foreign language, or specify conditional resource compilation with the if-then-else structures of the preprocessor.

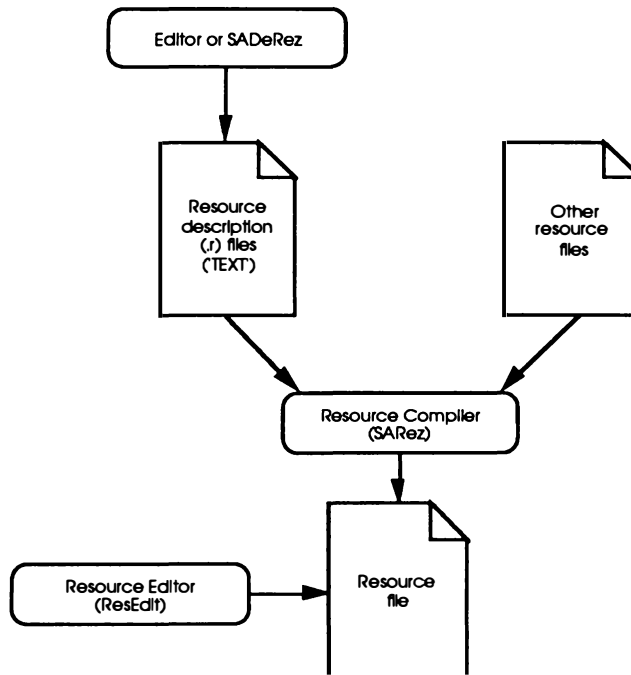
Standard type declaration files

Three text files, `Types.r`, `SysTypes.r`, and `Pict.r`, contain resource declarations for standard resource types. These files are located in the `{RIncludes}` folder.

File	Contains...
<code>Types.r</code>	Type declarations for the most common Macintosh resource types ('ALRT', 'DITL', 'MENU', and so on)
<code>SysTypes.r</code>	Type declarations for 'DRVr', 'FOND', 'FONT', 'FWID', 'INTL', 'NFMt', and many others
<code>Pict.r</code>	Type declaration for PICT resources for debugging PICTs

Using SAREz and SAdRez

SAREz and SAdRez are primarily used to create and modify resource files. This picture illustrates the process of creating a resource file.



Structure of a Resource Description File

The resource description file consists of resource type declarations (which can be included from another file) followed by resource data for the declared types. Note that the resource compiler and

resource decompiler have no built-in resource types. You need to define your own types or include the appropriate .r files.

A resource description file may contain any number of these statements:

Statement	Description
include	Include resources from another file.
read	Read data fork of a file and include it as a resource.
data	Specify raw data
type	Type declaration — declare resource type descriptions for subsequent resource statements.
resource	Data specification — specify data for a resource type declared in a previous type statement.
change	Change the type, ID, name, or attributes of existing resources.
delete	Delete existing resources.

Each of these statements is described in the sections that follow.

A **type declaration** describes how the declaration of a resource of that type will look like. You must declare a type (with a type statement) before you make a resource of that type (with a resource statement). Otherwise, you can freely mix type and resource statements in a file. You can redefine a type any number of times, even a type defined in Types.r, SysTypes.r, or Pict.r.

A resource description file can also include comments and preprocessor directives:

- **Comments** can be included any place white space is allowed in a resource description file, by putting them within the comment delimiters /* and */. Note that comments do not nest. For example, this is *one* comment:

```
/* Hello /* there */
```

SARez also supports C++ style comments:

```
type 'tost' {    // the rest of this line is ignored
```

- **Preprocessor directives** substitute macro definitions and include files, and provide if-then-else processing before other SARez processing takes place. The syntax of the preprocessor is very similar to that of the C-language preprocessor.

Sample resource description file

An easy way to learn about the resource description format is to decompile some existing resources. For example, using SAdRez to decompile an application's resources might generate this:

```
resource 'WIND' (128, "Sample Window") {
    {64, 60, 314, 460},
    documentProc,
    visible,
    noGoAway,
    0x0,
    "Sample Window"
};
```

This resource data corresponds to the following type declaration, contained in Types.r:

```
type 'WIND' {
    rect;                                /* boundsRect */
    integer documentProc, dBoxProc, plainDBox, /* procID */
        altDBoxProc, noGrowDocProc,
        zoomProc=8, rDocProc=16;
    byte    invisible, visible;          /* visible */
    fill byte;
    byte    noGoAway, goAway;            /* goAway */
    fill byte;
    unsigned hex longint;                 /* refCon */
    pstring Untitled = "Untitled";       /* title */
};
```

Type and resource statements are explained in detail in the reference section that follows. The SAdRez utility is explained in detail in Chapter 23, "SAdRez."

Resource Description Statements

This section describes the syntax and use of the seven types of resource description statements available for the resource compiler: include, read, data, type, delete, change, and resource.

Syntax notation

The following syntax notation is used to describe the resource description statements:

terminal	Plain text indicates a word that must appear in the statement exactly as shown. Special symbols (i.e., -, *, =) and punctuation (i.e., commas "," and semicolons ";") must also be entered exactly as shown.
<i>nonterminal</i>	Items in italics can be replaced by anything that matches their definition.
[optional]	Square brackets mean that the enclosed elements are optional.

<i>repeated ...</i>	An ellipsis (...), when it appears <i>in the text of this reference only</i> , indicates that the preceding item can be repeated one or more times.
<i>a b</i>	A vertical bar indicates an either/or choice
<i>(grouping)</i>	Parentheses indicate grouping (useful with the and ... notation).
<i>'[x]'</i>	Curly single quotation marks ('...') indicate that one of the syntax notation characters (for example, [or]) must be written as a literal. In this example, the brackets would be typed literally. They do <i>not</i> mean that the <i>x</i> is optional

Spaces between syntax elements, constants, and punctuation are optional. They are shown for readability only.

Tokens in resource description statements may be separated by spaces, tabs, returns, or comments.

Special terms

The following terms represent a minimal subset of the nonterminal symbols used to describe the syntax of commands in the resource description language:

Term	Definition
<i>resource-type</i>	<i>long-expression</i>
<i>resource-name</i>	<i>string</i>
<i>resource-ID</i>	<i>word-expression</i>
<i>ID-range</i>	<i>ID[: ID]</i>

Note: *Expression* is defined later in this chapter under "Expressions."

A full syntax definition can be found at the end of this chapter.

Include — Include resources from another file

The include statement lets you read resources from an existing file and include all or some of them.

Syntax

An include statement can take the following forms:

```
include file [ resource-type ['( resource-name | ID [ :ID ] ')'] ] ;
```

Read the resource of type *resource-type* with the specified resource name or resource ID range in *file*. If the resource name or ID is omitted, read all resources of the type *resource-type* in *file*. If *resource-type* is omitted, read all the resources in *file*.

```
include file not resource-type ;
```

Read all resources **not** of the type *resource-type* in *file*.

```
include file resource-type1 as resource-type2 ;
```

Read all resources of type *resource-type1* and include them as resources of *resource-type2*.

```
include file resource-type1 '(' resource-name | ID [ :ID ] ')'
as resource-type2 '(' ID [, resource-name ] [, attributes... ] ')';
```

Read the resource of type *resource-type1* with the specified name or ID range in *file*, and include it as a resource of *resource-type2* with the specified ID. You can optionally specify a resource name and resource attributes. (Resource attributes are defined below.)

Note: SDeRez ignores all include statements.

Some examples follow:

```
include "otherfile"; /* include all resources from the file */
include "otherfile" 'CODE'; /* read only the CODE resources */
include "otherfile" 'CODE' (128); /* read only CODE resource 128 */
```

AS resource description syntax

The following string variables can be used in the *as* resource description to modify the resource information in include statements:

\$\$Type	Type of resource from include file
\$\$ID	ID of resource from include file
\$\$Name	Name of resource from include file
\$\$Attributes	Attributes of resource from include file

For example, to include all 'DRVr' resources from one file and keep the same information but also set the SYSHEAP attribute:

```
INCLUDE "file" 'DRVr' (0:40) AS
'DRVr' ($$ID, $$Name, $$Attributes | 64) ;
```

The \$\$Type, \$\$ID, \$\$Name, and \$\$Attributes variables are also set and legal within a normal resource statement. At any other time the values of these variables are undefined.

Resource attributes

You can specify **attributes** as a numeric expression (as described in *Inside Macintosh I*, Chapter 5, "The Resource Manager," and *Inside Macintosh VI*, Chapter 13, "The Resource Manager"), or you can set them individually by specifying one of the keywords from any of the following pairs:

Default	Alternative	Meaning
appheap	sysheap	Specifies whether the resource is to be loaded into the application heap or the system heap.
nonpurgeable	purgeable	Purgeable resources can be automatically purged by the Memory Manager.

Default	Alternative	Meaning
unlocked	locked	Locked resources cannot be moved by the Memory Manager.
unprotected	protected	Protected resources cannot be modified by the Resource Manager.
nonpreload	preload	Preloaded resources are placed in the heap as soon as the Resource Manager opens the resource file.
unchanged	changed	Tells the Resource Manager whether a resource has been changed. SAREz does not allow you to set this bit, but SADeRez will display it if it is set.

Bits 0 and 7 of the resource attributes are reserved for use by the Resource Manager and cannot be set by SAREz, but are displayed by SADeRez.

You can specify more than one attribute by separating the keywords with a comma (,).

Read — read data as a resource

The `read` statement lets you read a file's data fork as a resource.

Syntax

```
read resource-type '(' ID [, resource-name ] [, attributes] ')' file ;
```

Description

Reads the data fork from *file* and writes it as a resource with the type *resource-type* and the resource ID *ID*, with the optional resource name *resource-name* and optional resource attributes (as defined in the preceding section). For example,

```
read 'STR ' (-789, "Test String", SysHeap, PreLoad) "Test8";
```

Note: SADeRez ignores all `read` statements.

Data — specify raw data

Use the `data` statement to specify raw data as a sequence of bits, without any formatting.

Syntax

```
data resource-type '(' ID [, resource-name ] [, attributes... ] ')' '{'
    data-string
    '}' ;
```

Description

Reads the data found in *data-string* and writes it as a resource with the type *resource-type* and the ID *ID*. You can optionally specify a resource name, resource attributes, or both.

For example,

```
data 'PICT' (128) {
    $"4F35FF8790000000"
    $"FF234F35FF790000"
};
```

Note: When SAdRez generates a resource description, it uses the data statement to represent any resource type that doesn't have a corresponding type declaration or cannot be disassembled for some other reason. SAdRez ignores any data statements in the resource description files you provide.

Type — declare resource type

A type declaration provides a template that defines the structure of the resource data for a single resource type or for individual resources. If more than one type declaration is given for a resource type, the last one read before the data definition is the one that's used. This lets you override declarations from include files or previous resource description files.

Syntax

```
type resource-type [ '(' ID-range ')' ] '{'
    type-specification...
'}
```

Description

Causes any subsequent resource statement for the type *resource-type* to use the declaration { *type-specification*... }. The optional *ID-range* specification causes the declaration to apply only to a given resource ID or range of IDs.

Type-specification is one of the following:

bitstring[<i>n</i>]	
byte	
integer	
longint	
boolean	
char	
string	
pstring	
wstring	
cstring	
point	
rect	
fill	Zero fill
align	Zero fill to nibble, byte, word, or long word boundary
switch	Control construct (case statement)
array	Array data specification—zero or more instances of data types

These types can be used singly or together in a type statement. Each of these type specifiers is described in the sections that follow.

Note: Several of these types require additional fields. The exact syntax is given in the sections that follow.

You can also declare a resource type that uses another resource's type declaration by using the following variant of the type statement:

```
type resource-type1 [ '(' ID-range ')' ] as resource-type2 [' (' ID ')'];
```

Data-type specifications

A Data-type statement declares a field of the given data type. It can also associate symbolic names or constant values with the data type. The data-type specification can take three forms, as shown in this example:

```
type 'XAMP' { /* declare a resource of type 'XAMP' */
    byte;
    byte off=0, on=1;
    byte = 2;
};
```

- The first byte statement declares a byte field; the actual data is supplied in a subsequent resource statement.
- The second byte statement is identical to the first, except that the two symbolic names *off* and *on* are associated with the values 0 and 1. These symbolic names could be used in the resource data.
- The third byte statement declares a byte field whose value is always 2. In this case, no corresponding statement would appear in the resource data.

Numeric expressions and strings can appear in type statements; they are defined later in this chapter under "Expressions."

Numeric types: The numeric types (*bitstring*, *byte*, *integer*, *longint*) are fully specified like this:

```
[ unsigned ] [ radix ] numeric-type [ =expr | symbol-definition... ];
```

- The *unsigned* prefix signals SDeRez that the number should be displayed without a sign—that the high-order bit can be used for data and the value of the integer cannot be negative. The *unsigned* prefix is ignored by SArez but is needed by SDeRez to correctly represent a decompiled number. SArez uses a sign if it is specified in the data. Precede a signed negative constant with a minus sign (–); \$FFFFFF85 and –\$7B are equivalent in value.
- *Radix* is one of the following string constants:

hex	decimal	octal	binary	literal
-----	---------	-------	--------	---------

You can supply numeric data as decimal, octal, hexadecimal, or literal data.

- *Numeric-type* is one of the following:

bitstring ' <i>length</i> '	Declare a bitstring of <i>length</i> bits (maximum 32).
byte	Declare a byte (8-bit) field. This is the same as bitstring[8].
integer	Integer (16-bit) field. This is the same as bitstring[16].
longint	Long integer (32-bit) field. This is the same as bitstring[32].

SARez uses integer arithmetic and stores numeric values as integer numbers. SARez translates booleans, bytes, integers, and longints to bitstring equivalents. All computations are done in 32 bits and truncated.

An error is generated if a value won't fit in the number of bits defined for the type. The valid ranges for values of byte, integer, and longint constants are as follows:

Type	Maximum	Minimum
byte	255	-128
integer	65,535	-32,768
longint	4,294,967,295	-2,147,483,648

Boolean type: A Boolean is a single bit with two possible states: 0 (or false) and 1 (or true). (True and false are global predefined identifiers.) Boolean values are declared as follows:

```
boolean [ = constant | symbolic-value... ];
```

The type boolean declares a 1-bit field; this is equivalent to

```
unsigned bitstring[1]
```

Note: This type is not the same as a Boolean variable as defined by Pascal.

Character type: Characters are declared as follows:

```
char [ = string | symbolic-value... ];
```

Type char declares an 8-bit field (this is the same as writing string[1]).

Here is an example:

```
type 'SYMB' {
    char dollar = "$", percent = "%";
};

resource 'SYMB' (128) {
    dollar
};
```

String type: String data types are specified like this:

string-type ['[' *length* ']'] [= *string* | *symbol-value*...];

String-type is one of the following:

Type	Description
[hex] string	Plain string (no length indicator or termination character is generated). The optional hex prefix tells SAdRez to display it as a hex string. String[<i>n</i>] contains <i>n</i> characters and is <i>n</i> bytes long. The type char is shorthand for String[1].
pstring	Pascal string (a leading byte containing the length information is generated). Pstring[<i>n</i>] contains <i>n</i> characters and is <i>n</i> +1 bytes long. Pstring has a built-in maximum length of 255 characters, the highest value the length byte can hold. If the string is too long to fit the field, a warning is given and the string is truncated.
wstring	Word string is a very large pstring. Its length is stored in the first two bytes. Therefore, a word string can contain up to 65,535 characters. Wstring[<i>n</i>] contains <i>n</i> characters and is <i>n</i> +2 bytes long.
cstring	C string (a trailing null byte is generated). cstring[<i>n</i>] contains <i>n</i> -1 characters and is <i>n</i> bytes long. A C string of length 1 can be assigned only the value "", because cstring[1] has room only for the terminating null.

Each string type may be followed by an optional *length* indicator in brackets ([*n*]). *Length* is an expression indicating the string length in bytes. *Length* is a positive number in the range $1 \leq \text{length} \leq 2147483647$ for string and cstring, and in the range $1 \leq \text{length} \leq 255$ for pstring, and in the range $1 \leq \text{length} \leq 65535$ for wstring.

Note: You cannot assign the value of a literal to a string type.

If no length indicator is given, a pstring, wstring, or cstring stores the number of characters in the corresponding data definition. If a length indicator is given, the data may be truncated on the right or padded on the right. The padding characters for all string types are nulls. If the data contains more characters than the length indicator provides for, the string is truncated and a warning message is given.

Warning: A null byte within a cstring is a termination indicator and may confuse SAdRez and C programs. However, the full string, including the explicit null and any text that follows it, will be stored by SAdRez as input.

Resource description statements. Point and rectangle types: Because points and rectangles appear so frequently in resource files, they have their own simplified syntax:

```
point [ = point-constant | symbolic-value... ];
rect  [ = rect-constant | symbolic-value... ];
```

where

```
point-constant = '{' x-integer-expr, y-integer-expr '}'
```

and

```
rect-constant = '{' integer-expr, integer-expr, integer-expr, integer-expr '}'
```

These type statements declare a point (two 16-bit signed integers) or a rectangle (four 16-bit signed integers). The integers in a rectangle definition specify the rectangle's upper-left and lower-right points, respectively.

Fill and align types

The resource created by a resource definition has no implicit alignment. It's treated as a bit stream, and integers and strings can start at any bit. The `fill` and `align` type specifiers are two ways of padding fields so that they begin on a boundary that corresponds to the field type. `Align` is automatic and `fill` is explicit. Both `fill` and `align` generate zero-filled fields.

Fill specification: The `fill` statement causes SAREz to add the specified number of bits to the data stream. The fill is always 0. The form of the statement is

```
fill fill-size [ '[' length ']' ] ;
```

where *fill-size* is one of the following strings:

```
bit          nibble          byte          word          long
```

These declare a fill of 1, 4, 8, 16, or 32 bits (optionally multiplied by the *length* modifier). *Length* is an expression ≤ 2147483647 .

The following `fill` statements are equivalent:

```
fill word[2];
fill long;
fill bit[32];
```

The full form of a type statement specifying a fill might be:

```
type 'XRES' { data-type specifications; fill bit[2]; };
```

Note: SAREz supplies zeros as specified by `fill` and `align` statements. SAREz does not supply any values for `fill` or `align` statements; it just skips the specified number of bits, or until data is aligned as specified.

Align specification: Alignment causes SAREz to add fill bits of zero value until the data is aligned at the specified boundary. An alignment statement takes the following form:

```
align align-size ;
```

where *align-size* is one of these strings:

```
nibble      byte      word      long
```

Alignment pads with zeros until data is aligned on a 4-, 8-, 16-, or 32-bit boundary. This alignment affects all data from the point where it is specified until the next align statement.

Array type

An array is declared as follows:

```
[ wide ] array [ array-name | '[' length ']' ] '[' array-list '];
```

The *array-list*, a list of type specifications, is repeated zero or more times. The wide option outputs the array data in a wide display format (in SAREz)—the elements that make up the array-list are separated by a comma and space instead of a comma, return, and tab. Either *array-name* or [*length*] may be specified. *Array-name* is an identifier.

If the array is named, then a preceding statement should refer to that array in a constant expression with the `$$CountOf(array-name)` function; otherwise SAREz will treat the array as an open-ended array. For example,

```
type 'STR#' {          /* define a string list resource */
  integer = $$CountOf(StringArray);
  array StringArray {
    pstring;
  };
};
```

The `$$CountOf` function returns the number of array elements (in this case, the number of strings) from the resource data.

If [*length*] is specified, there must be exactly *length* elements.

Array elements are generated by commas. Commas are element separators. Semicolons are element terminators. In this example, however, it may be a good idea to use semicolons as element separators:

```
type 'xyzy' {
  array Increment {
    integer = $$ArrayIndex(Increment);
  };
};
```

```

resource 'xyzy' (0) {
    { /* zero elements */
    }
};

resource 'xyzy' (1) {
    { /* two elements */
    '
    }
};

resource 'xyzy' (3) {
    } /* two elements */
    ;;
}
};

/* The only way to specify one element in an array that has all
   constant elements, is to use a semicolon terminator.
*/

resource 'xyzy' (4) {
    { /* one element */
    ;
    }
};

```

Switch type

The `switch` statement specifies a number of case statements for a given field or fields in the resource. The format is:

```
switch '{' case-statement... '}';
```

where a *case-statement* has this form:

```
case case-name : [ case-body ; ]...
```

Case-name is an identifier. *Case-body* may contain any number of type specifications and must include a single constant declaration per case, in this form:

```
key data-type = constant
```

Which case applies is based on the key value. For example,

```
type 'DITL' { /* dialog item list declaration from Types.r */
  ...type specifications...
  switch { /* one of the following */
    case Button:
      boolean enabled, disabled;
      key bitstring[7] = 4; /* key value */
      pstring;
    case CheckBox:
      boolean enabled, disabled;
      key bitstring[7] = 5; /* key value */
      pstring;
      ...and so on.
  };
};
```

Sample type statement

The following sample type statement is the standard declaration for a 'WIND' resource, taken from the Types.r file:

```
type 'WIND'{
  rect; /* bounds */
  integer documentProc, dBoxProc, plainDBox, /* procID */
    altDBoxProc, noGrowDocProc,
    zoomProc=8, rDocProc=16;
  byte invisible, visible; /* visible */
  fill byte;
  byte noGoAway, goAway; /* close box */
  fill byte;
  unsigned hex longint; /* refCon */
  pstring Untitled = "Untitled"; /* title */
};
```

The type declaration consists of header information followed by a series of statements, each terminated by a semicolon (;). The header of the sample window declaration is

```
type 'WIND'
```

The header begins with the Type keyword followed by the name of the resource type being declared—in this case, a window. You may specify a standard Macintosh resource type, as shown in the *Inside Macintosh I*, Chapter 5, "The Resource Manager," and *Inside Macintosh VI*, Chapter 13, "The Resource Manager," or you may declare a resource type specific to your application.

The left brace [{] introduces the body of the declaration. The declaration continues for as many lines as necessary until a matching right brace } is encountered. You can write more than one statement on a line, and a statement may be on more than one line (like the integer statement above). Each statement represents a field in the resource data. Recall that comments may appear

anywhere where white space may appear in the resource description file; comments begin with `/*` and end with `*/` as in C.

Symbol definitions

Symbolic names for data type fields simplify the reading and writing of resource definitions. Symbol definitions have the form

```
name = value [, name = value ]...
```

For numeric data, the “= *value*” part of the statement can be omitted. If a sequence of values consists of consecutive numbers, the explicit assignment can be left out—if *value* is omitted, it’s assumed to be one greater than the previous value. (The value is assumed to be zero if it’s the first value in the list.) This is true for bitstrings (and their derivatives, byte, integer, and longint). For example,

```
integer documentProc, dBoxProc, plainDBox,
      altDBoxProc, noGrowDocProc,
      zoomProc=8, rDocProc=16;
```

In this example, the symbolic names `documentProc`, `dBoxProc`, `plainDBox`, `altDBoxProc`, and `noGrowDocProc` are automatically assigned the numeric values 0, 1, 2, 3, and 4.

Memory is the only limit to the number of symbolic values that can be declared for a single field. There is also no limit to the number of names you can assign to a given value; for example,

```
integer documentProc=0, dBoxProc=1, plainDBox=2, altDBoxProc=3,
      rDocProc=16, Document=0, Dialog=1, DialogNoShadow=2,
      ModelessDialog=3, DeskAccessory=16;
```

Delete — delete a resource

Sometimes you may want to delete a resource without switching to ResEdit. Some resource operations, such as those needed by “internationalizing” system disks and applications need to translate menu and dialog text, and hence require deleting or changing resources.

Syntax

```
delete resource-type ['(' resource-name | ID[:ID] ')'];
```

Description

Delete the resource of type *resource-type* from the output file with the specified resource name or resource ID range. If the resource name or ID is omitted, all resources of type *resource-type* are deleted.

Note: The delete function is valid only when you are appending to an existing file. It makes no sense to delete resources while creating a new resource file from scratch. Also, SAdRez ignores all delete commands.

You can delete resources that have their protected bit set only if you select the OK to Replace Protected Resources option in SAREz.

Change — change a resource's vital information

You can change a resource's vital information by using this function. Vital information includes the resource type, ID, name, attributes, or any combination of these at once.

Syntax

```
change resource-type1 [ '(' resource-name | ID[:ID] ')' ]
    to resource-type2 '(' ID [, resource-name ] [, attributes...] ')';
```

Description

Change the resource of type *resource-type1* from the output file with the specified resource name or resource ID range to a resource of type *resource-type2* with the specified ID. You can optionally specify a resource name and resource attributes. If the resource name or attributes are not specified, the name and attributes are not changed.

For example, this statement sets the protected bit on all code resources:

```
change 'CODE' to $$type ($$ID,$$Attributes | 8);
```

Note: The change function is only valid when you are appending to an existing file. It makes no sense to change resources while creating a new resource file from scratch. Also, SAREz ignores all change commands.

Resource — specify resource data

Resource statements specify actual resources, based on previous type declarations.

Syntax

```
resource resource-type '(' ID [, resource-name ] [, attributes] ')' '{'
    [ data-statement [ , data-statement ]... ]
    '}';
```

Description

Specifies the data for a resource of type *resource-type* and ID *ID*. The latest type declaration declared for *resource-type* is used to parse the data specification. *Data-statements* specify the actual data; *data-statements* appropriate to each resource type are defined in the next section.

The resource definition causes an actual resource to be generated. A resource statement can appear anywhere in the resource description file, or even in a separate file specified on the command line or as an #include file, as long as it comes after the relevant type declaration.

Note: SAREz ignores all resource statements.

Data statements

The body of the data specification contains one data statement for each declaration in the corresponding type declaration. The base type must match the declaration.

Base type	Instance types
string	string, cstring, pstring, wstring, char
bitstring	boolean, byte, integer, longint, bitstring
rect	rect
point	point

Switch data: Switch data statements are specified by using this format:

switch-name data-body

For example, the following could be specified for the 'DITL' type given earlier:

```
...
CheckBox { enabled, "Check here" },
...
```

Array data: Array data statements have this format:

{' [array-element [, array-element]...] '}

where an *array-element* consists of any number of data statements separated by commas.

For example, the following data might be given for the 'STR#' resource defined earlier:

```
resource 'STR#' (280) {
    { "this",
      "is",
      "a",
      "test"
    }
};
```

Sample resource definition

This section describes a sample resource description file for a window. (See the chapter *Inside Macintosh I*, Chapter 9, "Window Manager," and *Inside Macintosh IV*, Chapter 6, "The Window Manager," for information about resources in windows.)

Here, again, is the type declaration given above under "Sample Type Statement":

```
type 'WIND'{
    rect;                                /* bounds    */
    integer documentProc, dBoxProc, plainDBox, /* procID   */
        altDBoxProc, noGrowDocProc,
        zoomProc=8, rDocProc=16;
    byte invisible, visible;             /* visible   */
    fill byte;
    byte noGoAway, goAway;               /* close box */
    fill byte;
    unsigned hex longint;                /* refCon    */
    pstring Untitled = "Untitled";      /* title     */
};
```

Here is a typical example of the window data corresponding to this declaration:

```
resource 'WIND' (128,"My window",appheap,preload) {
    {40,80,120,300}, /* Status report window */
    documentProc,    /* Bounding rectangle    */
    Visible,         /* documentProc etc..   */
    goAway,          /* Visible or Invisible */
    0,               /* GoAway or NoGoAway   */
    "Status Report"  /* Reference value RefCon */
};
```

This data definition declares a resource of type 'WIND', using whatever type declaration was previously specified for 'WIND'. The resource ID is 128; the resource name is *My window*. Because the resource name is represented by the Resource Manager as a *pstring*, it should not contain more than 255 characters. The resource name may contain any character including the null character (\$00). The resource will be placed in the application heap when loaded, and it will be loaded when the resource file is opened.

The first statement in the window type declaration declares a bounding rectangle for the window:

```
rect;
```

The rectangle is described by two points: the upper-left corner and the lower-right corner. The points of a rectangle are separated by commas like this:

```
{ top, left, bottom, right }
```

An example of data for these coordinates is

```
{ 40, 80, 120, 300 }
```


Symbolic names: Symbolic names may be associated with particular values of a numeric type. Notice that a symbolic name is given for the data in the second, third, and fourth fields of the window declaration. For example,

```
integer documentProc=0, dBoxProc=1, plainDBox=2,
      altDBoxProc=3, noGrowDocProc=4,
      zoomProc=8, rDocProc=16;    /* windowType */
```

This statement specifies a signed 16-bit integer field with symbolic names associated with the values 0 to 4 and 16. The values 0 through 4 need not be indicated in this case; if no values are given, symbolic names are automatically given values starting at 0, as explained previously.

In the sample window declaration, we gave the values `True` (1) and `False` (0) to two different byte variables. For clarity, we used those symbolic names in the window's resource data; that is,

```
visible,
goAway,
```

instead of their equivalents

```
TRUE,
TRUE,
```

or

```
1,
1,
```

Labels

Labels support some of the more complicated resources such as 'NFNT' and color QuickDraw resources. Use labels within a resource type declaration to calculate offsets and permit accessing of data at the labels.

Syntax

```
label      ::= character {alphanum}* ':'
character  ::= '_' | A | B | C ...
number     ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
alphanum   ::= character | number
```

Description

Labeled statements are valid only within a resource type declaration. Labels are local to each type declaration. More than one label can appear on a statement.

Labels may be used in expressions. In expressions, use only the identifier portion of the label (that is, everything up to, but excluding, the colon). See "Declaring Labels Within Arrays" later in this chapter for more information.

The value of a label is always the offset, *in bits*, between the beginning of the resource and the position where the label occurs when mapped to the resource data. In this example,

```
type 'cool' {
    cstring;
endOfString:
    integer = endOfString;
};

resource 'cool' (8) {
    "Neato"
}
```

the integer following the cstring would contain:

$$(\text{len}(\text{"Neato"}) [5] + \text{null byte} [1]) * 8 [\text{bits per byte}] = 48.$$

Built-In functions to access resource data

In some cases, it is desirable to access the actual resource data that a label points to. Several built-in functions allow access to that data:

\$\$BitField(*label*, *startingPosition*, *numberOfBits*)

Returns the *numberOfBits* (maximum of 32) bitstring found *startingPosition* bits from *label*.

\$\$Byte(*label*)

Returns the byte found at *label*.

\$\$Word(*label*)

Returns the word found at *label*.

\$\$Long(*label*)

Returns the longword found at *label*.

For example, the resource type 'STR' could be redefined without using a pstring. Here is the definition of 'STR' from Types.r:

```
type 'STR' {
    pstring;
}
```

Here is a redefinition of 'STR' using labels:

```
type 'STR' {
    len:    byte = (stop - len) / 8 - 1;
           string[$$Byte(len)];
    stop:   ;
};
```

Declaring labels within arrays

Labels declared within arrays may have many values. For every element in the array, there is a corresponding value for each label defined within the array. Use array subscripts to access the individual values of these labels. The subscript values range from 1 to n where n is the number of elements in the array. Labels within arrays that are nested in other arrays require multidimensional subscripts. Each level of nesting adds another subscript. The rightmost subscript varies most quickly. Here is an example:

```
type 'test' {
  integer = $$CountOf(array1);
  array array1 {
    integer = $$CountOf(array2);
    array array2 {
foo:      integer;
    };
  };
};

resource 'test' (128) {
  {
    {1,2,3},
    {4,5}
  }
};
```

In the above example, the label `foo` takes on these values:

```
foo[1,1] = 32    $$Word(foo[1,1]) = 1
foo[1,2] = 48    $$Word(foo[1,2]) = 2
foo[1,3] = 64    $$Word(foo[1,3]) = 3
foo[2,1] = 96    $$Word(foo[2,1]) = 4
foo[2,2] = 112   $$Word(foo[2,2]) = 5
```

A new built-in function may be helpful in using labels within arrays:

`$$ArrayIndex (arrayname)`

This function returns the current array index of the array *arrayname*. An error occurs if this function is used anywhere outside the scope of the array *arrayname*.

Label limitations

Keep in mind the fact that SAREz and SAdRez are basically one-pass compilers. This will help you understand some of the limitations of labels.

Note: To decompile (or “DeRez”) a given type, that type must not contain any expressions with more than one undefined label. An undefined label is a label that occurs lexically after the expression. To define a label, use it in an expression before the label is defined.

This example demonstrates how expressions can only have only one undefined label:

```
type 'test' {
    /* In the expression below, start is defined, next is undefined.*/
start: integer = next - start;
    /* In the expression below, next is defined because it was used
       in a previous expression, but final is undefined.*/
middle: integer = final - next;
next: integer;
final:
};
```

Actually, SAREz can compile types that have expressions containing more than one undefined label, but SAdRez cannot decompile those resources and simply generates data resource statements.

Note: The label specified in `$$BitField()`, `$$Byte()`, `$$Word()`, and `$$Long()` must occur lexically before the expression; otherwise, an error is generated.

Using labels: two examples

The first example shows the modified 'ppat' declaration using the new SAREz labels. Boldface text in the example indicates everything that is different between the 2.0 and 3.0 versions of 'ppat' because of the use of labels. Without using labels, the whole end section of the resource would have to be combined into a single hex string (everything following the PixelData label). Using labels, the complete 'ppat' definition can be expressed in SAREz language.

```

type 'ppat' {
    /* PixPat record */
    integer oldPattern,          /* Pattern type */
    newPattern,
    ditherPattern;
    unsigned longint = PixelFormat / 8; /* Offset to pixmap */
    unsigned longint = PixelData/8; /* Offset to data */
    fill long;                  /* Expanded pixel image */
    fill word;                  /* Pattern valid flag */
    fill long;                  /* expanded pattern */
    hex string [8];             /* old-style pattern */
                                /* PixMap record */

PixelFormat:
    fill long;                  /* Base address */
    unsigned bitstring[1] = 1; /* New pixmap flag */
    unsigned bitstring[2] = 0; /* Must be 0 */
    unsigned bitstring[13];    /* Offset to next row */
    rect;                      /* Bitmap bounds */
    integer;                   /* pixmap vers number */
    integer unpacked;          /* Packing format */
    unsigned longint;          /* size of pixel data */
    unsigned hex longint;      /* h. resolution (ppi) (fixed) */
    unsigned hex longint;      /* v. resolution (ppi) (fixed) */
    integer chunky, chunkyPlanar, planar; /* Pixel storage format */
    integer;                   /* # bits in pixel */
    integer;                   /* # components in pixel */
    integer;                   /* # bits per field */
    unsigned longint;          /* Offset to next plane */
    unsigned longint = ColorTable / 8; /* Offset to color table */
    fill long;                 /* Reserved */

PixelData:
    hex string [(ColorTable - PixelData) / 8];

```

```

ColorTable:
    unsigned hex longint;          /* ctSeed      */
    integer;                       /* transIndex  */
    integer = $$Countof(ColorSpec) - 1; /* ctSize     */
    wide array ColorSpec {
        integer;                   /* value       */
        unsigned integer;          /* RGB: red    */
        unsigned integer;          /* green       */
        unsigned integer;          /* blue        */
    };
};

```

Here is another example of a new resource definition with the new features in bold. In this example, the `$$BitField()` function is used to access information stored in the resource, in order to calculate the size of the various data areas added at the end of the resource. Without labels, all data would have to be combined into one hex string.

```

type 'cicn' {
    fill long;
    unsigned bitstring[1] = 1;
    unsigned bitstring[2] = 0;
pMapRowBytes: unsigned bitstring[13];
Bounds: rect;
    integer;
    integer unpacked;
    unsigned longint;
    unsigned hex longint;
    unsigned hex longint;
    integer chunky, chunkyPlanar, planar;
    integer;
    integer;
    integer;
    unsigned longint;
    unsigned longint;
    fill long;

    fill long;
maskRowBytes: integer;
    rect;

    fill long;
iconBMapRowBytes: integer;
    rect;
    fill long;

    /* IconPMap (pixMap) record */
    /* Base address */
    /* New pixMap flag */
    /* Must be 0 */
    /* Offset to next row */
    /* Bitmap bounds */
    /* pixMap vers number */
    /* Packing format */
    /* Size of pixel data */
    /* h.resolution (ppi) (fixed) */
    /* v.resolution (ppi) (fixed) */
    /* Pixel storage format */
    /* # bits in pixel */
    /* # components in pixel */
    /* # bits per field */
    /* Offset to next plane */
    /* Offset to color table */
    /* Reserved */

    /* IconMask (bitMap) record */
    /* Base address */
    /* Row bytes */
    /* Bitmap bounds */

    /* IconBMap (bitMap) record */
    /* Base address */
    /* Row bytes */
    /* Bitmap bounds */
    /* Handle placeholder */
}

```

```

/* Mask data */
hex string [$$Word(maskRowBytes) * ($$BitField(Bounds, 32, 16)
/*bottom*/
    - $$BitField(Bounds, 0, 16) /*top*/)];

/* BitMap data */
hex string [$$Word(iconBMapRowBytes) *
    ($$BitField(Bounds, 32, 16)/*bottom*/
    - $$BitField(Bounds, 0, 16) /* top */)];

/* Color Table */
unsigned hex longint;          /* ctSeed */
integer;                      /* transIndex */
integer = $$Countof(ColorSpec) - 1; /* ctSize */
wide array ColorSpec {
    integer;                  /* value */
    unsigned integer;         /* RGB: red */
    unsigned integer;         /* green */
    unsigned integer;         /* blue */
};

/* PixelMap data */
hex string [$$BitField(pMapRowBytes,0,13) *
    ($$BitField(Bounds,32,16) /* bottom */
    - $$BitField(Bounds, 0, 16) /*top*/)];
};

```

Preprocessor Directives

Preprocessor directives substitute macro definitions and include files and provide if-then-else processing before other SAREz processing takes place.

The syntax of the preprocessor is very similar to that of the C-language preprocessor. Preprocessor directives must observe these rules and restrictions:

- Each preprocessor statement must be expressed on a single line, beginning on a new line and terminated by a return character.
- The pound sign (#) must be the first character on the line of the preprocessor statement (except for spaces and tabs).
- *Identifiers* (used in macro names) may be letters (A–Z, a–z), digits (0–9), or the underscore character (_).
- Identifiers may be any length.
- Identifiers may not start with a digit.
- Identifiers are not case sensitive.

Variable definitions

The `#define` and `#undef` directives let you assign values to identifiers:

```
#define  macro  data
#undef   macro
```

The `#define` directive causes any occurrence of the identifier *macro* to be replaced with the text *data*. You can extend a macro over several lines by ending the line with the backslash character (`\`), which functions as the SAREz escape character. For example,

```
#define poem "I wander \
thro\' each \
charter\'d street"
```

(Quotation marks within strings must also be escaped.)

`#undef` removes the previously defined identifier *macro*. Macro can also be defined and undefined in the Preprocessor... dialog in SAREz or in the Preprocessor box in SAdRez.

The following predefined macros are provided:

Variable	Value
true	1
false	0
rez	1 or 0 (1 if SAREz is running, 0 if SAdRez is running)
derez	1 or 0 (0 if SAREz is running, 1 if SAdRez is running)

Include directives

The `#include` directive reads a text file:

```
#include file
```

Include the text file *file*. The maximum nesting is to ten levels. For example,

```
#include "MyTypes.r"
```

Note that the `#include` preprocessor directive (which includes a file) is different from the previously described `include` statement, which copies resources from another file.

If-Then-Else processing

These directives provide conditional processing:

```
#if expression
[ #elif expression ]
[ #else ]
#endif
```


Note: *Expression* is defined later in this chapter. When used with the `#if` and `#elif` directives, *expression* may also include this expression:
defined *identifier* or defined '(' *identifier* ')

The following may also be used in place of `#if`:

```
#ifndef macro
#endif macro
```

For example,

```
#define Thai
Resource 'STR ' (199) {
#ifdef English
    "Hello"
#elif defined (French)
    "Bonjour"
#elif defined (Thai)
    "Sawati"
#elif defined (Japanese)
    "Konnichiwa"
#endif
};
```

Print directive

The `#printf` directive is provided to aid in debugging resource description files:

```
#printf(formatString, arguments...)
```

The format of the `#printf` statement is exactly the same as the `printf` statement in the C language, with one exception: There can be no more than 20 arguments. This is the same restriction that applies to the `$$format` function. The `#printf` directive writes its output to diagnostic output. Note that the `#printf` directive *does not* end with a semicolon.

Note: `SARez` and `SADeRez` will not print the results of the `#printf` directive if you do not select an error file or if there are no errors. (`SARez` and `SADeRez` don't print error files if there are no errors to report.)

For example:

```
#define Tuesday 3
#ifdef Monday
    #printf("The day is Monday, day %d\n", Monday)
#elif defined(Tuesday)
    #printf("The day is Tuesday, day %d\n", Tuesday)
#elif defined(Wednesday)
    #printf("The day is Wednesday, day %d\n", Wednesday)
#elif defined(Thursday)
    #printf("The day is Thursday, day %d\n", Thursday)
```

```
#else
#printf("DON'T KNOW WHAT DAY IT IS!\n")
#endif
```

The above file generates this text:

The day is Tuesday, day #3

Resource Description Syntax

This section describes the details of the resource description syntax.

Numbers and Literals

All arithmetic is performed as 32-bit signed arithmetic. The basic constants are shown below:

Numeric type	Form	Meaning
Decimal	<i>nnn...</i>	Signed decimal constant between 4,294,967,295 and -2,147,483,648.
Hex	<i>0xhhh...</i>	Signed hexadecimal constant between 0X7FFFFFFF and 0X80000000.
	<i>\$hhh...</i>	Alternate form for hexadecimal constants.
Octal	<i>0ooo...</i>	Signed octal constant between 017777777777 and 020000000000.
Binary	<i>0Bbbb...</i>	Signed binary constant between 0B11111111111111111111111111111111 and 0B10000000000000000000000000000000.
Literal	<i>'aaaa'</i>	A literal may contain one to four characters. Characters are printable ASCII characters or escape characters. If there are fewer than four characters in the literal, then the characters to the left (high bits) are assumed to be \$00. Characters that are not in the printable character set, and are not the characters \ ' and \\ (which have special meanings), can be escaped according to the character escape rules. (See "Strings" later in this section.)

Literals and numbers are treated in the same way by the resource compiler. A **literal** is a value within single quotation marks; for instance, 'A' is a number with the value 65; on the other hand, "A" is the character A expressed as a string. Both are represented in memory by the bitstring 01000001. (Note, however, that "A" is not a valid number and 'A' is not a valid string.) The following numeric expressions are all equivalent:

```
'B'
66
'A'+1
```

Literals are padded with nulls on the left side so that the literal 'ABC' is stored as shown below.

"ABC" =	\$00	A	B	C
---------	------	---	---	---

Expressions

An expression may consist of simply a number or literal. Expressions may also include numeric variables, labels, and system functions.

This table lists the operators in order of precedence with highest precedence first—groupings indicate equal precedence. Evaluation is always left to right when the priority is the same. Variables are defined following the table.

Operator	Meaning
1. (<i>expr</i>)	Parentheses can be used in the normal manner to force precedence in expression calculation
2. - <i>expr</i>	Arithmetic (two's complement) negation of <i>expr</i>
~ <i>expr</i>	Bitwise (one's complement) negation of <i>expr</i>
! <i>expr</i>	Logical negation of <i>expr</i>
3. <i>expr1</i> * <i>expr2</i>	Multiplication
<i>expr1</i> / <i>expr2</i>	Division
<i>expr1</i> % <i>expr2</i>	Remainder from dividing <i>expr1</i> by <i>expr2</i>
4. <i>expr1</i> + <i>expr2</i>	Addition
<i>expr1</i> - <i>expr2</i>	Subtraction
5. <i>expr1</i> << <i>expr2</i>	Shift left—shift <i>expr1</i> left by <i>expr2</i> bits
<i>expr1</i> >> <i>expr2</i>	Shift right—shift <i>expr1</i> right by <i>expr2</i> bits
6. <i>expr1</i> > <i>expr2</i>	Greater than
<i>expr1</i> >= <i>expr2</i>	Greater than or equal to
<i>expr1</i> < <i>expr2</i>	Less than
<i>expr1</i> <= <i>expr2</i>	Less than or equal to
7. <i>expr1</i> == <i>expr2</i>	Equal
<i>expr1</i> != <i>expr2</i>	Not equal
8. <i>expr1</i> & <i>expr2</i>	Bitwise AND

Operator	Meaning
9. <i>expr1</i> ^ <i>expr2</i>	Bitwise XOR
10. <i>expr1</i> <i>expr2</i>	Bitwise OR
11. <i>expr1</i> && <i>expr2</i>	Logical AND
12. <i>expr1</i> <i>expr2</i>	Logical OR

The logical operators !, >, >=, <, <=, ==, !=, &&, and || evaluate to 1 (true) or 0 (false).

Variables and functions

Some resource compiler variables contain commonly used values. All SAREz variables start with \$\$ followed by an alphanumeric identifier.

The following variables and functions have string values (typical values are given in parentheses):

\$\$Date

Current date. Useful for putting timestamps into the resource file. The format is generated through the ROM call IUDateString. ("Thursday, May 20, 1987")

\$\$Format ("formatString", arguments)

Works just like the #printf directive except that \$\$format returns a string rather than printing to standard output. (See the section "Print Directive" earlier in this chapter.)

\$\$Name

Name of resource from the current resource. The current resource is the resource being generated in a resource statement, being included from an include statement, being deleted from a delete statement, or changed in a change statement.

For example, to include all 'DRVR' resources from one file and keep the same information, but also set the SYSHEAP attribute:

```
INCLUDE "file" 'DRVR' (0:40) AS
'DRVR' ($$ID, $$Name, $$Attributes | 64);
```

The \$\$Type, \$\$ID, \$\$Name, and \$\$Attributes variables are undefined outside of a change, delete, include, or resource statement.

\$\$Resource ("filename", 'type', ID | "resourceName")

Reads the resource 'type' with the ID ID or the name "resourceName" from the resource file "filename", and returns a string.

\$\$Shell ("stringExpr")

Included for MPW-compatibility. In SAREz and SAdRez, this returns the null string (""). In MPW's SAREz and SAdRez, it returns the current value of the exported Shell variable {stringExpr}. Note that the braces are omitted, and the double quotation marks must be present.

\$\$Time

Current time. Useful for time-stamping the resource file. The format is generated through the ROM call `IUTimeString`. ("7:50:54 AM")

\$\$Version

Version number of SAREz. ("V3.0")

These variables and functions have numeric values:

\$\$Attributes

Attributes of resource from the current resource. See the `$$Name` string variable.

\$\$BitField(*label*, *startingPosition*, *numberOfBits*)

Returns the *numberOfBits* (maximum of 32) bitstring found *startingPosition* bits from *label*.

\$\$Byte(*label*)

Returns the byte found at *label*.

\$\$Day

Current day. Range 1–31.

\$\$Hour

Current hour. Range 0–23.

\$\$ID

ID of resource from the current resource. See the `$$Name` string variable.

\$\$Long(*label*)

Returns the longword found at *label*.

\$\$Minute

Current minute. Range 0–59.

\$\$Month

Current month. Range 1–12.

\$\$PackedSize(*Start*, *RowBytes*, *RowCount*)

Given an offset (*Start*) into the current resource and two integers, *RowBytes* and *RowCount*, this function calls the Toolbox routine `UnpackBits()` *RowCount* times. `$$PackedSize()` returns the unpacked size of the data found at *start*. Use this function only for decompiling resource files. An example of this function is found in `Pict.r`.

\$\$ResourceSize

Current size of resource in bytes. When decompiling, `$$ResourceSize` is the actual size of the resource being decompiled. When compiling, `$$ResourceSize` returns the number of bytes that have been compiled so far for the current resource. (See the 'KCHR' resource in `SysTypes.r` for an example.)

\$\$Second

Current second. Range 0–59.

\$\$Type

Type of resource from the current resource. See the \$\$Name string variable.

\$\$Weekday

Current day of the week. Range 1–7 (that is, Sunday–Saturday).

\$\$Word (*label*)

Returns the word found at *label*.

\$\$Year

Current year.

Strings

There are two basic types of strings:

Text string	" <i>a...</i> "	The string can contain any printable character except ' ' and '\'. These and other characters can be created through escape sequences. (The section "Escape Sequences" below lists all the escape sequences.) The string "" is a valid string of length 0.
Hex string	\$" <i>hh...</i> "	Spaces and tabs inside a hexadecimal string are ignored. There must be an even number of hexadecimal digits. The string \$" is a valid hexadecimal string of length 0.

Any two strings (hexadecimal or text) will be concatenated if they are placed next to each other with only white space in between. (In this case, returns and comments are considered white space.)

The picture below shows a Pascal string declared as

```
pstring [10];
```

whose data definition is

```
"Hello"
```

\$05	H	e	l	l	o	\$00	\$00	\$00	\$00	\$00
------	---	---	---	---	---	------	------	------	------	------

In the input file, string data is surrounded by double quotation marks ("). You can continue a string on the next line. A separating token (for example, a comma) or brace signifies the end of the

string data. A side effect of string continuation is that a sequence of two quotation marks ("") is simply ignored. For example,

```
"Hello ""out "
"there."
```

is the same string as

```
"Hello out there.";
```

To place a quotation mark character within a string, precede the quotation mark with a backslash like this

```
(\").
```

Escape characters

The backslash character (\) is provided as an escape character to allow you to insert nonprintable characters in a string. For example, to include a newline character in a string, use the escape sequence \n.

These are the valid escape sequences:

Escape sequence	Name	Hex value	Printable equivalent
\t	Tab	\$09	None
\b	Backspace	\$08	None
\r	Return	\$0A	None
\n	Newline	\$0D	None
\f	Form feed	\$0C	None
\v	Vertical tab	\$0B	None
\?	Rubout	\$7F	None
\\	Backslash	\$5C	\
\'	Single quotation mark	\$3A	'
\"	Double quotation mark	\$22	"

You can also use octal, hexadecimal, decimal, and binary escape sequences to specify characters that do not have predefined escape equivalents. The forms are:

Base	Number form	Digits	Example
2	\0Bbbbbbbb	8	\0B01J000001
8	\ooo	3	\101
10	\0Dddd	3	\0D065
16	\0Xhh	2	\0X41
16	\\$hh	2	\\$41

Here are some examples:

```
\077          /* 3 octal digits */
\0xFF         /* '0x' plus 2 hex digits */
\$F1\$F2\$F3   /* '$' plus 2 hex digits */
\0d099        /* '0d' plus 3 decimal digits */
```

Note to C programmers: An octal escape code consists of exactly three digits. For instance, to place an octal escape code with a value of 7 in the middle of an alphabetic string, write AB\007CD, not AB\7CD.

You can select the Don't Escape Characters option in SADeRez to print characters that would otherwise be escaped (characters preceded by a backslash, for example). Normally, only characters with values between \$20 and \$D8 are printed as Macintosh characters. With this option, however, all characters (except null, newline, tab, backspace, form-feed, vertical tab, and rubout) will be printed as characters, not as escape sequences. See Chapter 23 "SADeRez" for details.

Using SAREz 22

Introduction

This chapter describes SAREz, a utility that creates resources from a textual description.

SAREz started out life as Rez, a tool in Apple's Macintosh Programmer's Workshop (MPW). MPW has a command-line interface, much like UNIX. At times it will seem SAREz works a bit oddly or has options you can't use. This is because of its heritage. But, despite its odd upbringing, SAREz is still a powerful and useful tool.

Topics covered in this chapter

- What is SAREz?
- Choosing input files
- Choosing an output file
- Setting options
- Saving and restoring options
- The message window

What Is SAREz?

SAREz (which means **Stand-Alone Rez** and is pronounced "SayRez") creates the resource fork of a file from a textual description. That textual description is found in one or more **resource description files**. The format for resource description files is described in Chapter 21.

If you followed the directions in Chapter 2, "Installing THINK Pascal," SAREz is in the folder Rez Utilities inside THINK Pascal 4.0 Utilities.

Running SAREz

When you run SAREz, you see the dialog below. To compile a resource description file, you open a SAREz options file you previously saved, or you can manually set the input files, output files, and other options. When you click the button titled "Sarez," SAREz compiles your files. If there are no errors, it exits. If there are errors, it displays them in the message window, described below. You can then edit, print, or save the contents of the messages window. To cancel without compiling, click the button titled "Cancel," or choose **Quit** from the **File** menu. More information on setting and saving your options is below.

The SAREz dialog box is titled "Untitled". It contains the following elements:

- Sarez Options:**
 - Resource Output File:** A text field containing "Rez.out", a "Type" dropdown set to "APPL", and a "Creator" field showing "????".
 - Rewrite resource file:** A radio button is selected. Below it is an unchecked checkbox for "Make resource file read-only".
 - Resource Alignment:** A group box containing three radio buttons: "Byte" (selected), "Word", and "LongWord".
 - Merge resources into resource file:** An unchecked radio button. Below it is an unchecked checkbox for "OK to replace protected resources".
- Checkboxes (right side):**
 - ☐ Progress information
 - ☐ Redeclared types ok
 - ☐ Modification date
- Buttons (right side):**
 - Description Files...
 - #Include Paths...
 - Include Paths...
 - Preprocessor...
 - Redirection...
- Command Line:** A text field containing the text "SAREz".
- Help:** A text field containing the text "Rez is a tool used to compile resources.".
- Bottom Right:** "Cancel" and "Sarez" buttons.
- Bottom Center:** The version number "3.2".

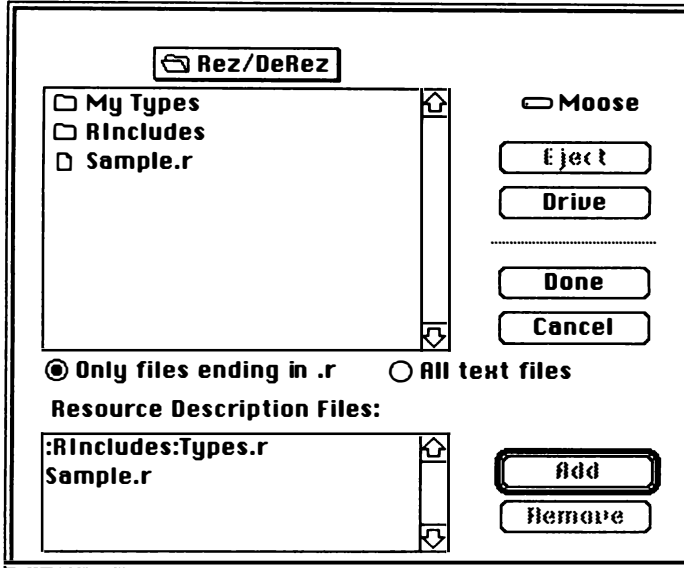
The SAREz dialog has two parts. In the first part, enclosed in a box titled "Sarez Options," you choose your files and options for SAREz's operation. In the second part, containing the boxes named "Command Line" and "Help," you can see how the options work.

The Command Line box may seem out of place in a Macintosh program. But, remember SAREz started life as Rez, an MPW tool. The Command Line box displays a Rez command line that has the same options set as the SAREz dialog. You cannot edit this line.

The Help box gives you information about the dialog. When you click on an option or a box, a description of it appears in this box.

Choosing Input Files

To choose the description files to use as input, click on the Description Files... button to display a dialog. The one shown below has `Types.r` (in the `RIncludes` folder) and `Sample.r` chosen as the description files.



These radio buttons determine which files appear in the top list:

Button	Description
Only Files Ending in .r	Displays files with names ending in .r, Such as <code>Sample.r</code>
All Text Files	Displays all text files, regardless of their names

You can add and remove files with these commands:

- To add a description file, select it from the standard file list in the top pane, and click Open. Its name is added to the list in the lower pane. You can also double-click on the file to add it to the bottom list.
- To remove a description file from the bottom list, select it, and click Remove.
- When you've selected all your description files, click Done.
- To cancel what you've done and leave the list of files as it was, click Cancel.

Note: If you don't select any files in the Description Files... dialog, SAREz will use the alternate input file set in the Redirection... dialog. If you don't select an alternate file in the Redirection... dialog, SAREz will act as if it were reading an empty file. Most of the time, you'll want to use the Description Files... dialog.

Choosing an Output File

To choose the file SAREz writes your resources to, use the Resource Output File box, shown below. This box is part of the SAREz dialog.

Resource Output File

Rez.out **Type** **APPL**
Creator **????**

☒ **Rewrite resource file**
☐ **Make resource file read-only**

Resource Alignment
☒ **Byte** ☐ **Word** ☐ **LongWord**

☐ **Merge resources into resource file**
☐ **OK to replace protected resources**

To choose the file, click on the button in the upper left hand corner of the box. This pop-up menu appears.

✓ Write output to Rez.out
Select an existing output file...
Write output to a new file...

This is what the choices mean:

Option	Description
Write output to Rez.out	Writes to file called Rez.out, creating it if necessary. This is the default
Select an existing output file...	Displays a standard file dialog that lets you choose an existing file. In the dialog, you can choose what it displays: only MPW tools and applications or all files.
Write output to a new file...	Displays a standard file dialog that lets you enter the name of a new file to create.

In the Type and Creator boxes, you enter the type and creator of the file. These must be four-letter values. If you are writing to an existing file, SAREz changes the type and creator of the file to the new values.

The rest of the buttons control how SAREz treats the resource fork it creates. These two determine whether to overwrite the existing resource fork:

Option	Description
Rewrite Resource File	Erases the file's resource fork and replaces it with the resources created from your description files. This is the appropriate choice for creating a new file or overwriting an old one.
Merge Resources into Resource File	Keeps the resource fork and appends your resources to them. If one of your resources have the same type and ID as an existing resource and the existing resource isn't protected, your resource will overwrite it.

If you choose Rewrite Resource File, you'll see two more options:

Option	Description
Make Resource File Read-only	Sets the mapReadOnly flag in the resource map.
Resource Alignment	Lets you choose how you want your resources aligned: along byte, word, or longword boundaries. The default is byte

If you choose Merge Resources into Resource File, you'll see this option:

Option	Description
OK to Replace Protected Resources	Overrides the protected bit in resources. Even if an existing resource has the protected bit set, SAREz will overwrite it when a new resource has the same type and ID.

Setting Options

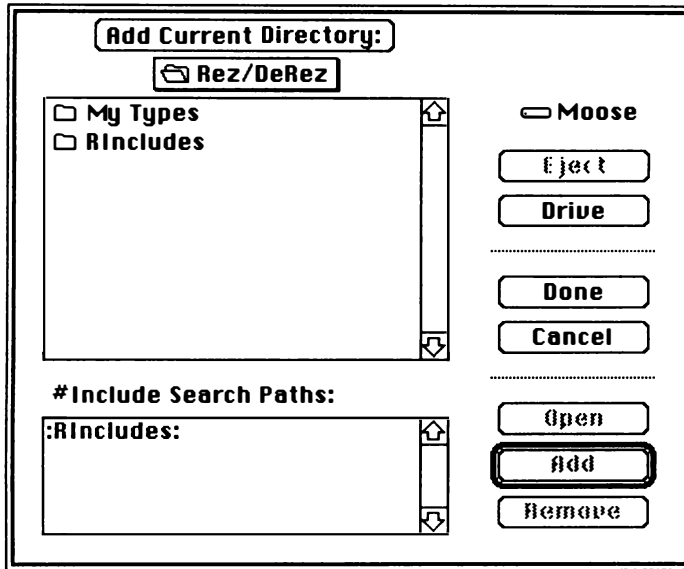
The SAREz Options box contains several check boxes for setting some options, several buttons that display dialogs for more detailed information, and the Resource Output File box for setting the output file. This section describes all the buttons and check boxes in the right column, except for Description Files..., which you've already seen in the section "Choosing Input Files." And the Resource Output File box is described in the section "Choosing Output Files" above.

The three check boxes in the upper-right hand corner control these options:

Choice	Description
Progress information	If selected, SAREz writes information on each type and resource created and the SAREz version number to the error file (selected in the Redirection... dialog).
Redeclared types OK	If selected, SAREz will not print warning messages to the error file when a resource type is redeclared.
Modification date	If selected, SAREz doesn't change the output file's modification date. Be careful. If an error occurs, SAREz sets the output file's modification date to zero, even if you select this option.

Setting the search paths for #Include and Include

The #Include Paths... and Include Paths... dialogs let you specify folders (also called directories or search paths) that SAREz will search when it looks for an include or #include file. The #Include Paths... dialog below has the folder RIncludes in the list of folders to be searched. The Include Paths... dialog is similar.



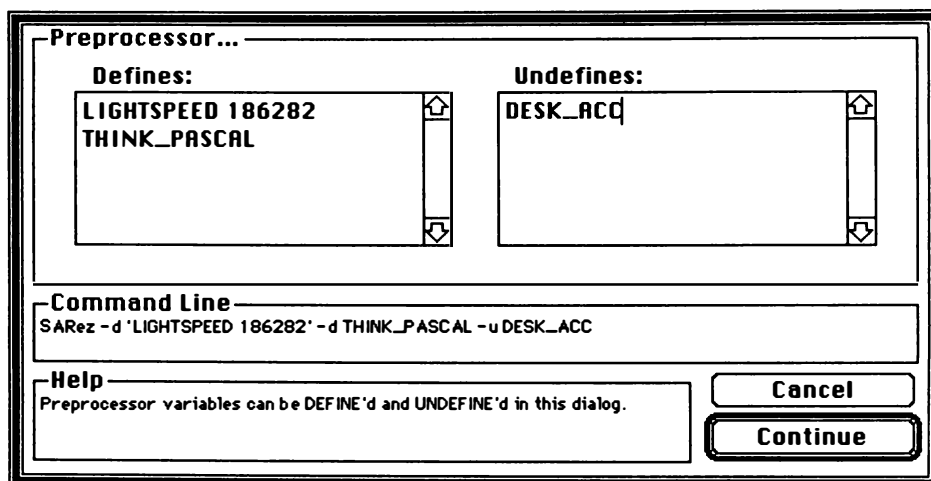
You can add and remove folders with these commands:

- To add a folder to the list of folders searched, select it from the standard file list in the top pane, and click Add. It's name is added to the list below.
- To add the folder you're currently in, click Add Current Directory. For example, clicking Add Current Directory in Figure 10-6 would add Rez/DeRez to the list.
- To open a folder to see the other folders in it, select it, and click Open. You can also double click on it.
- To remove a description file from the bottom, select it, and click Remove.
- When you've selected all the folders you want, click Done.
- To cancel what you've done and leave the list of folders as it was, click Cancel.

Defining and undefining macros

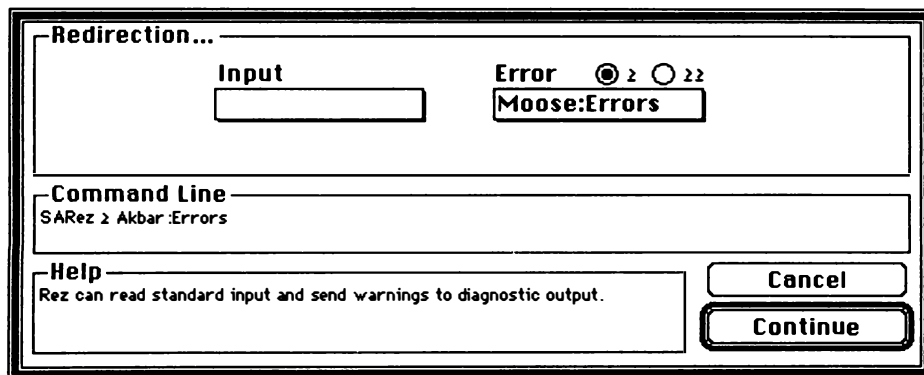
The Preprocessor... dialog lets you define and undefine macro variables. In the Defines box, enter the variables you want defined and their values. Use a new line for each pair of variables and values. If you don't enter a value, the variable is set to the null string (""). In the Undefines box, enter the variables you want undefined. For example, setting up the Preprocessor... dialog below is like adding these lines to the beginning of each description file:

```
#define LIGHTSPEED 186282
#define THINK_PASCAL
#undef DESK_ACC
```

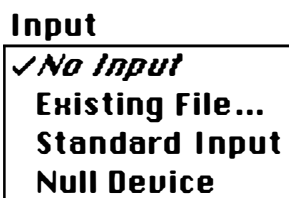


Choosing an error and alternate Input files

The Redirection... dialog lets you choose an alternate input file and the error file. The alternate input file is what SAREz reads if there are no description files set in the Description Files... dialog. The error file is an additional place to write out error, warning, and status messages. SAREz always writes its messages to the messages window, described below. The dialog below sets the error file to be Errors (on a disk named Moose) and sets no alternate input file.



Clicking on the button under Input brings up the pop-up menu below.



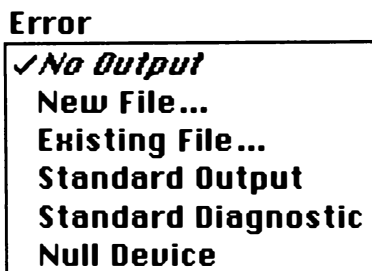
This is what the options mean:

Choice	Description
No Input	The same as setting the alternate input file to be an empty file.
Existing File...	Lets you select a file to read from.
Standard Input	These act the same as No Input .
Null Device	

Note: SAREz uses the input file you set here only if you don't choose description files in the Description Files... dialog. Most of the time, you'll want to use the Description Files... dialog and leave the alternate input file set to **No Input**.

The **Error** menu lets you choose an additional place to write error, warning, and status messages. SAREz always writes its messages to the messages window, described below. You might use this menu if you always want your messages saved to a particular file.

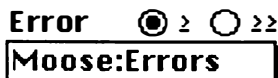
Clicking on the button under Error brings up the pop-up-menu below.



This is what the options mean:

Choice	Description
No Output	Writes messages only to the messages window.
New File...	Lets you create a file to write messages to.
Existing File...	Lets you choose an existing file to write messages to.
Standard Output Standard Diagnostic Null Device	These act the same as No Output

If you select an existing file as your error file, you'll see two radio buttons above the Error menu, shown below.



You can choose whether or not to overwrite the existing file:

Choice	Description
>	The new messages will overwrite the existing file.
>>	The new messages will be appended to the end of the existing file.

Saving and Restoring Options

SAREz lets you save your option settings in a SAREz options file. This options file contains the names of your input and output files, in addition to all the other option settings. When the SAREz

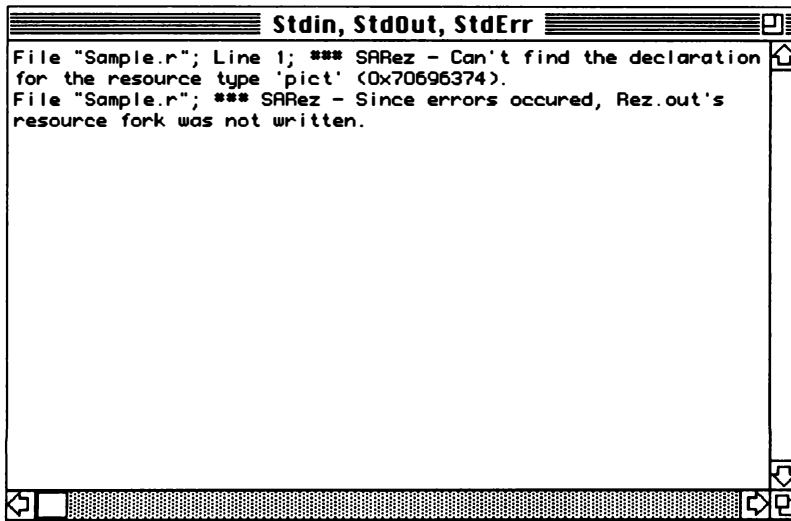
dialog box is the front window, the **File** menu contains these commands to let you save and restore your settings:

Command	Description
New	Clears your current settings and lets you create a new options file.
Open...	Lets you choose a file you previously saved and restores those settings.
Save	Saves the current settings to the options file you're using. If you aren't using an options file (that is, you haven't opened or saved one), it lets you create one.
Save As...	Lets you create an options file containing your current settings.

When you can double-click on a SAREz options file in the Finder, SAREz doesn't display its dialog. It just compiles your files, displays your errors (if any), and exits.

The Messages Window

If SAREz encounters no problems while compiling your files, it will just exit to the Finder when it's done. But if SAREz needs to display any its error, warning, or status messages, it uses the messages window, shown below.



You can freely edit the contents of the message window, as if it were a text editor. You can add your own comments and cut, copy, or paste, using the **Edit** menu.

When the messages window is in the front, the **File** menu contains these commands to let you save and print your messages:

Command	Description
Close	Closes the messages window.
Save	Saves your messages to a file. If you haven't chosen a file yet, it lets you choose one.
Save As...	Lets you choose a file to save your messages to.
Page Setup...	Displays the standard Page Setup... dialog for your printer.
Print...	Lets you print the contents of the messages window.

With the **Font** and **Size** menus, you can change the font SAREz displays and prints your messages with.

Note: If you always save your messages to a particular file, you can use the Redirection... dialog to choose an error file. SAREz will write your messages to both the messages window and the error file.

Using SAdRez

23

Introduction

This chapter describes SAdRez, which creates a text representation of the resource fork of a file.

SAdRez started out life as DeRez, a tool in Apple's Macintosh Programmer's Workshop (MPW). MPW has a command-line interface, much like UNIX. At times it will seem SAdRez works a bit oddly or has options you can't use. This is because of its heritage. But, despite its odd upbringing, SAdRez is still a powerful and useful tool.

Topics covered in this chapter

- What is SAdRez?
- Choosing input files
- Choosing output files
- Setting options

What Is SAdRez?

SAdRez (which stands for **Stand-Alone DeRez** and is pronounced "Say Dee' Rez") creates a text representation (a resource description file) of the resource fork of a file, according to the resource type declarations in one or more resource description files.

If you followed the directions in Chapter 2, "Installing THINK Pascal," SAdRez is in the folder Rez Utilities inside THINK Pascal 4.0 Utilities.

A **resource description file** is a file of type declarations in the format used by the resource compiler, Rez. The type declarations for standard Macintosh resources are in the files `Types.r` and `SysTypes.r`, in the `{RIncludes}` folder. If no resource description file is specified, the output consists of data statements giving the resource data in hexadecimal form, without any additional format information. The format for resource description files is described in Chapter 21.

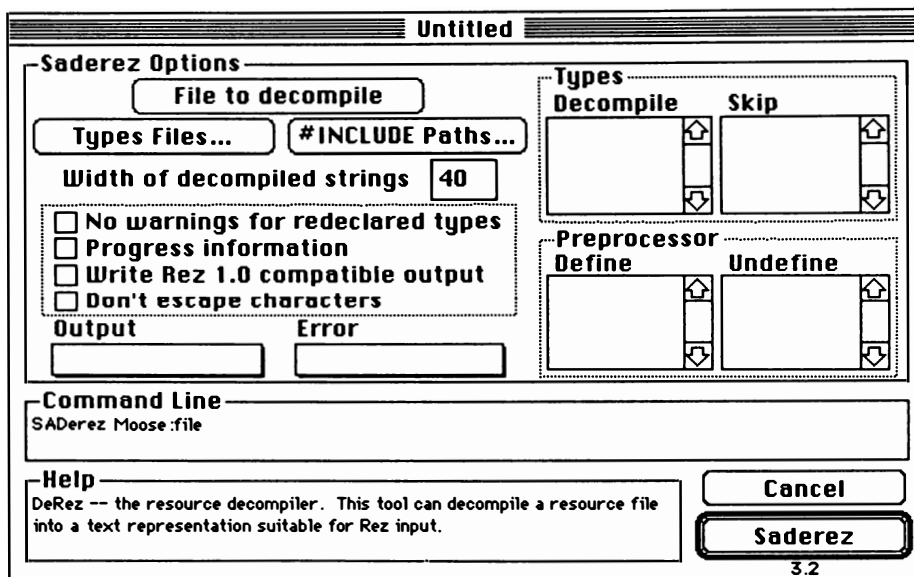
Note: SAdRez uses resource description files as both input and output. The input description files define resource types (such as 'WIND'), and the output description file defines actual resources (such as the document window for your application)

If the output of SAdRez is used as input to SAdRez, with the same resource description files, it produces the same resource fork that was originally input to SAdRez. SAdRez is not guaranteed to be able to run a declaration backwards; if it can't, it produces a data statement instead of the appropriate resource statement.

\SADeRez ignores all include (but not #include), read, data, change, delete, and resource statements found in the resource description files. (But it still parses these statements for correct syntax.)

Running SADeRez

When you run SADeRez, its dialog is greyed out, except for the File To Decompile button. You can either open a SADeRez options file you previously saved or choose a file to decompile. Now, the dialog looks like the one below. To continue, you can change your options and click the button titled "Saderez." If there are no errors, SADeRez compiles your file and exits. If there are errors, SADeRez displays them in the messages window, described below. To cancel without decompiling, click the button titled "Cancel" or choose **Quit** from the **File** menu. More information on choosing and saving your options follows.



The SADeRez dialog has two major parts. In the first part, enclosed in a box titled "Saderez Options," you set options for SADeRez's operation. There is more information about these options below. In the second part, containing the boxes named "Command Line" and "Help," you can see how the options work.

The Command Line box may seem out of place in a Macintosh program. But, remember SADeRez started life as DeRez, an MPW tool. The Command Line box displays a DeRez command line that has the same options set as the SADeRez dialog. You cannot edit this line.

The Help box gives you information about the dialog. When you click on an option or a box, a description of it appears in this box.

Choosing Input Files

You must choose at least two input files. First, choose a resource file, a file containing resources you want decompiled. Next, choose one or more resource description files, containing definitions of resource types. You specify these with the buttons in the upper-left-hand corner of the SDeRez dialog, as shown below.

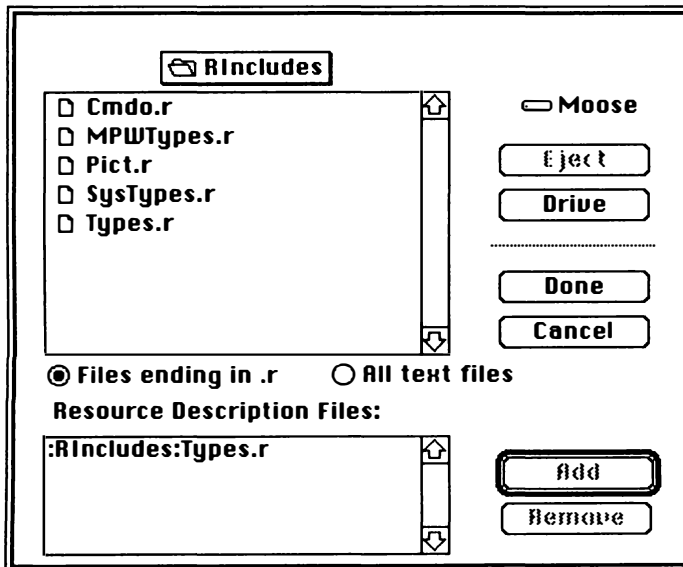


Choosing the resource file

First, you'll need to choose a resource file. The File To Decompile button displays a standard file dialog that lets you select one. The dialog offers to let you select any file or only an application or MPW tool.

Choosing a description file

The Types Files... button displays a dialog that lets you select one or more description files. The dialog below has Types.r (in the RIncludes folder) selected as the description file.



These buttons choose which files appear in the top list:

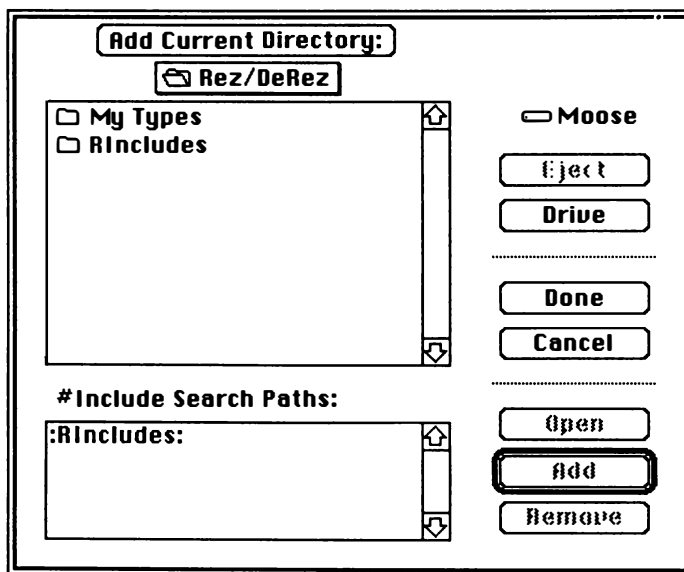
Option	Description
Only Files Ending in .r	Display only files that end in .r, such as Types.r
All Text Files	Display all text files, regardless of their names

You can add and remove files with these commands:

- To add a description file to the list in the bottom pane, select it in the standard file list in the top pane, and click Open. It's name is added to the list below. You can also double-click on the file to add it to the bottom list.
- To remove a description file from the bottom list, select it, and click Remove.
- When you've selected all your description files, click Done.
- To cancel what you've done and leave the list of files as it was, click Cancel.

Choosing #Include paths

If any of your description files contain #include statements, you'll need to select the folders (also called directories or search paths) in which SADeRez can find them. To choose these, click on the #INCLUDE Paths... button to display a dialog. The dialog below has the folder RIncludes in the list of folders to be searched.



Note: SADeRez ignores include statements in the resource description files.

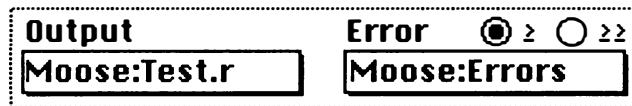
You can add and remove folders with these commands:

- To add a folder to the list of folders searched in the bottom pane, select it in the standard file list in the bottom pane, and click Add. Its name is added to the list in the bottom pane.
- To add the folder you're currently in, click Add Current Directory. In the dialog above, clicking Add Current Directory would add Rez/DeRez to the bottom list.
- To open a folder to see the other folders in it, select it, and click Open. You can also double click on it.
- To remove a description file from the bottom list, select it, and click Remove.
- When you've selected all the folders you want, click Done.
- To cancel what you've done and leave the list of folders as it was, click Cancel.

Choosing Output Files

You need to choose an output description file. This will contain the definitions of the resources in your resource file. You can also choose an error file, an additional place to write error, warning, and status messages. SDeRez always writes its messages to the messages window, described below. You would choose an error file if you always want your messages saved to a particular file.

You choose these files with the buttons in the lower-left-hand corner of the SDeRez Options box. The buttons in Figure 11-5 set the description file to be the new file `Test.r` and the error file to be the existing file `Errors` (both on the disk `Moose`).



Clicking on the button under either Output or Error brings up this pop-up menu.



This is what the options mean:

Option	Description
No Output	Doesn't write out the output.
New File...	Lets you create a file to write to.
Existing File...	Lets you select an existing file to write to.
Standard Output Standard Diagnostic Null Device	These act the same as No Output

If you select an existing file as an output file, you'll see these two radio buttons above the menu. (In the Output menu, the buttons are labeled ">" and ">>").



These let you choose whether or not to overwrite the existing file:

Option	Description
≥ or >	The new output will overwrite the existing file.
≥≥ or >>	The new output will be appended to the end of the existing file.

Setting Options

The rest of the SADeRez Options box lets you select options describing how to decompile your file. The list of options, shown below, is between the input and output buttons.

Width of decompiled strings

☐ **No warnings for redeclared types**
☐ **Progress information**
☐ **Write Rez 1.0 compatible output**
☐ **Don't escape characters**

This is what the options mean:

Option	Description
Width of decompiled strings	Sets the maximum string size. It must be in the range 2–120. This controls string width in the output.
No warnings for redeclared types	If selected, SAdRez will not print a warning message to the error file when a resource type is redeclared in the description files.
Progress information	If selected, SAdRez writes its version number and information on the decompiled resources to the error file.
Write Rez 1.0 compatible output	If selected, SAdRez will generate a description file that is backward compatible with Rez 1.0.
Don't escape characters	If selected, characters that are normally escaped (such as \0xff) are no longer escaped. Instead they are printed as extended Macintosh characters. (Note: Not all fonts have all the characters defined.) Normally, only characters with values between \$20 and \$D8 are printed as Macintosh characters. With this option, however, all characters (except null, newline, tab, backspace, form feed, vertical tab, and rubout) are printed as characters, not as escape sequences.

Choosing the types to decompile

In the Types box, you can choose which types of resources SAdRez decompiles. This box contains two lists:

Option	Description
Decompile	If you want to decompile only a few resource types, enter their names here. For example, with the box below, SAdRez will decompile resources of type 'WIND' only.

Types	
Decompile	Skip
WIND	

Option	Description
Skip	If you want to decompile every resource type with a few exceptions, enter the exceptions here. For example, with the box below, SAdRez will decompile all resources, except those of type 'PICT'.

Types	
Decompile	Skip
	PICT

You can use only one list at a time. Whenever one has anything in it, the other is greyed out. If you don't use either list, SAdRez will decompile everything in the input file.

Preprocessor

In the Preprocessor box, you can define and undefine macro variables. In the Defines list, enter the variables you want defined and their values. Use a new line for each pair of variables and values. If you don't enter a value, the variable is set to the null string (""). In the Undefines list, enter the variables you want undefined. For example, setting up the Preprocessor box as shown below is like adding these lines to the beginning of each resource description file:

```
#define c 186282
#define PASCAL
#undef DESK_ACC
```

Preprocessor	
Define	Undefine
c 186282	DESK_ACC
PASCAL	

Saving and Restoring Options

SAdRez lets you save your option settings in a SAdRez options file. This options file contains the names of your input and output files, in addition to all the other option settings.

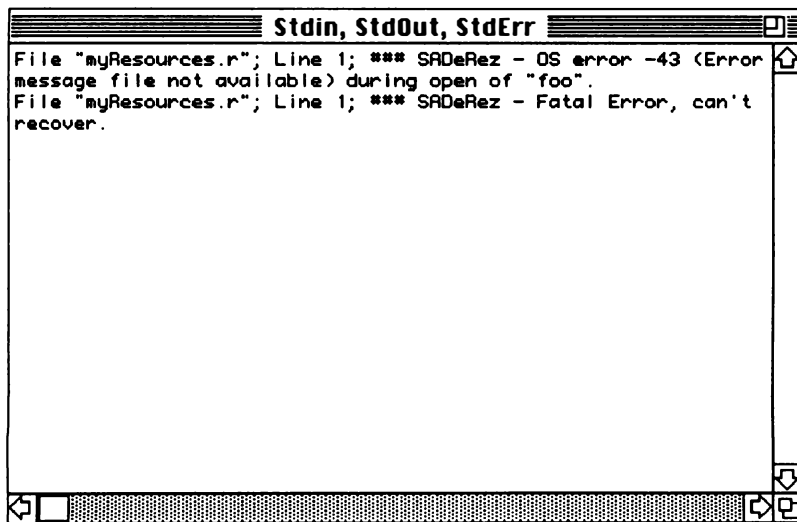
When the SDeRez dialog box is in the front, the **File** menu contains these commands to let you save and restore your settings:

Command	Description
New	Clears your current settings and lets you create a new options file.
Open...	Lets you choose a file you previously saved and restores those settings.
Save	Saves the current settings to the options file you're using. If you aren't using an options file (that is, you haven't opened or saved one), it lets you create one.
Save As...	Lets you create an options file containing your current settings.

When you can double-click on a SDeRez options file in the Finder, SDeRez doesn't display its dialog. It just compiles your files, displays your errors (if any), and exits.

The Messages Window

If SDeRez encounters no problems while decompiling your file, it will just exit to the Finder when it's done. But if SDeRez needs to display any its error, warning, or status messages, it uses the messages window, like this one:



You can freely edit the contents of the message window, as if it were a text editor. You can add your own comments and cut, copy, or paste, using the **Edit** menu.

When the messages window is in the front, the **File** menu contains these commands to let you save and print your messages:

Command	Description
Close	Closes the messages window.
Save	Saves your messages to a file. If you haven't chosen a file yet, it lets you choose one.
Save As...	Lets you choose a file to save your messages to.
Page Setup...	Displays the standard Page Setup... dialog for your printer.
Print...	Lets you print the contents of the messages window.

With the **Font** and **Size** menus, you can change the font SDeRez displays and prints your messages with.

Note: If you always save your messages to a particular file, you can use the Redirection... dialog to choose an error file. SDeRez will write your messages to both the messages window and the error file.

Using SPostRez 24

Introduction

This chapter describes SPostRez, which converts a resource file for use with a MacApp program.

Topics covered in this chapter

- What is SPostRez?
- Using SPostRez

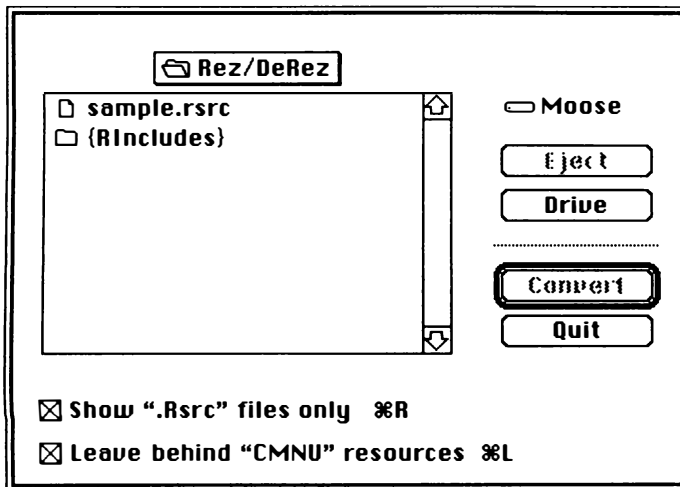
What is SPostRez?

SPostRez (which stands for **Stand-Alone PostRez** and is pronounced “Say Post Rez”) converts each 'cmnu' resource in a file into a 'MENU' resource and creates an 'mntb' resource. Your MacApp program uses the mntb resource to translate menu choices to command numbers. For more information on cmnu and mntb resources, see your MacApp documentation.

If you followed the directions in Chapter 2, “Installing THINK Pascal,” SPostRez in the folder Rez Utilities inside THINK Pascal 4.0 Utilities.

Using SPostRez

To start SPostRez, double-click on its icon. You see this dialog :



This is what the options mean:

Option	Description
Show ".Rsrc" files only	If selected, only files ending in .rsrc, like sample.rsrc, appear in the file list.
Leave behind "CMNU" resources	If selected, SAPostRez doesn't remove the 'cmnu' resources from the resource file after it converts them. This is useful if you're still developing your program and may change your program's menus or command numbers.

To convert the 'cmnu' resources in a file, select the file and click on the Convert button. SAPostRez converts the 'cmnu' resources and displays its dialog again. SAPostRez does not create a new file but adds or changes resources in the file you select.

To quit from SAPostRez, click on the Quit button.

THINK PascalTM

P A R T S I X

Appendices

- A What's New
- B ANS Pascal Compatibility
- C Porting to THINK Pascal
- D Error Messages

What's New A

Introduction

This chapter describes many of the new features in THINK Pascal 4.0.

Topics covered in this appendix

- Compatibility with earlier releases
- System 7.0 compatibility
- Working with large projects
- Working with small projects
- New and improved commands
- Enhanced THINK Class Library
- Other changes

Compatibility with Earlier Releases

THINK Pascal 4.0 is completely compatible with THINK Pascal 2.0 and later versions. The new version automatically converts project documents created with the older versions, discarding all the object code. THINK Pascal 4.0 can read THINK Pascal 2.0 and 3.0 libraries and source files (even those saved as Entire Document) without modification.

System 7.0 Compatibility

THINK Pascal 4.0 works under System 7.0. It lets your applications take full advantage of the new system software. THINK Pascal 4.0 does the following:

- Runs with 32-bit addressing
- Produces applications that run with 32-bit addressing (like all versions since THINK Pascal 2.0)
- Runs with virtual memory
- Handles the required AppleEvents
- Recognizes all Toolbox routines in Inside *Macintosh VI*
- Contains the THINK Class Library 1.1, with System 7.0 support.

THINK Pascal lets you work with the alias of a project file. The project tree begins where the original project is. However, THINK Pascal does not support aliases in these cases:

- Putting aliases in a project
- Using an alias as a project's resource file
- Inserting an alias of a folder in your THINK Pascal tree or project tree.

Working with Large Projects

THINK Pascal 4.0 makes it easier to work with large projects. It supports applications with larger jump tables, multi-segment code resources, and some extensions to the **uses** clause.

Far CODE

In previous versions of THINK Pascal, large programs could run out of space for the jump table. In THINK Pascal 4.0, you have an almost unlimited amount of space for the jump table with the "Far CODE" option in the **Set Project Type...** dialog. For more information, see "Building applications with large jump tables" in Chapter 12, "Building Projects."

Multi-segment code resources

THINK Pascal 4.0 lets you create multi-segment code resources, such as cdevs, INITs, XCMDs, and XFCNs. THINK Pascal already supports multi-segment applications, device drivers, and desk accessories. Now, any kind of THINK Pascal project can have multiple segments. For more information, see "Multi-segment code resources" in Chapter 12, "Building Projects."

Extensions to the USES clause

Specifying the dependencies between the files in large projects is easier with some extensions to the **uses** clause. If you turn on the "USES Extensions" option in the **Compiler Options...** dialog, THINK Pascal lets you use these features:

- **Propagated uses.** If your unit uses other units, any unit that uses your unit also uses those units automatically.
- **Implementation uses.** You can put a uses clause in a unit's implementation section.

For more information, see "The uses clause," in Chapter 10, "Units and Libraries."

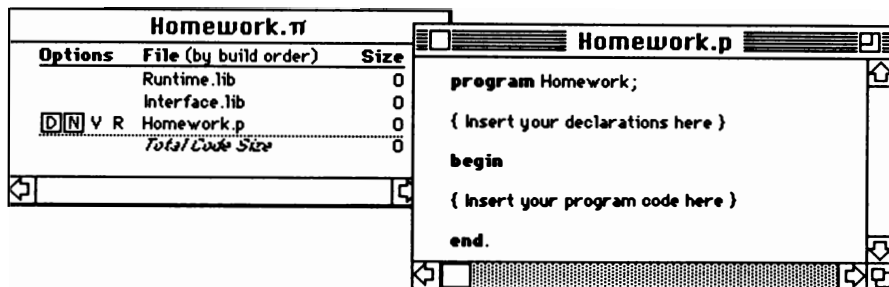
Working with Small Projects

THINK Pascal 4.0 helps out programmers who write smaller projects. It can automatically create a skeleton for small projects, print and save the Text and Drawing Windows, and includes an improved AppleEdit DA.

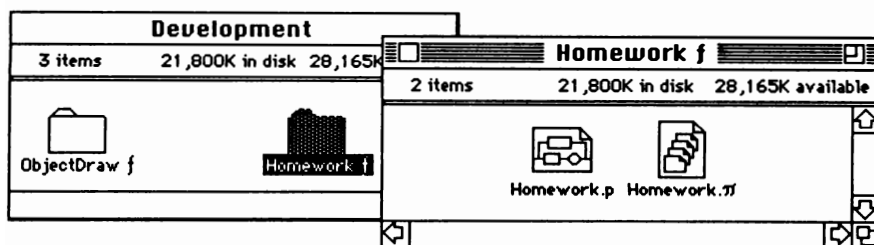
Instant project

If you're writing a small project, with only one or two source files, the Instant Project option in the **New Project...** dialog gets you started programing quicker. This option creates all this for you automatically: an empty source file, a project file that contains the source file, and a folder to hold the project file and source file.

For example, if you name your program Homework, THINK Pascal creates the project file `Homework.π` and source file `Homework.p`:



And it creates the folder `Homework f`:



For more information, see “Creating small projects” in Chapter 7, “Working with Projects.”

The Text and Drawing Windows

If your programs use the Text or Drawing windows, you'll appreciate THINK Pascal's enhanced support. You can now do the following:

- Print the windows with the **Print...** command.
- Save the contents of the windows with the **Save As...** command. THINK Pascal saves the contents of the Text Window as a Teach Text file and the contents of the Drawing Window as an ObjectDraw file. (ObjectDraw is the drawing program you create in Chapter 4, “Tutorial: ObjectDraw.”)

New AppleEdit DA

The AppleEdit DA (AppleEdit) that comes with THINK Pascal in the THINK Pascal 4.0 Utilities folder has these new features:

- It's 32-bit clean, so you can run it under System 7.0 in 32-bit addressing mode.
- It lets you print your files.

New and Improved Commands

THINK Pascal 4.0 makes it easier to select menu commands with these enhancements:

- In previous versions of THINK Pascal, many commands performed different actions when you held down the Shift key. In THINK Pascal 4.0, these commands now have different names when you hold down the Shift key. There's more information on these commands later in this section.
- After you select a menu, you can hold down a modifier key to see additional choices. Previously, you had to hold down the modifier key *before* you selected the menu. For example, select the **Debug** menu. Then hold down the Option key. You see **Pull Stops** become **Pull All Stops**. Release the Option key and hold down the Shift key. You see **LightsBug** become **New LightsBug** and **Monitor** become **Use Monitor**.
- You can use the Option and Shift keys in command-key combinations. For example, to choose **Save**, press Command-S; to choose **Save All**, press Command-Option-S.
- The Command-key equivalent for the **Print...** command is now Command-P. The command-key for bringing the project window to the front is changed to Command-0 (Command-Zero).

Dialog command keys

Many dialogs in THINK Pascal let you choose options and click buttons with command keys. In previous versions, you didn't always know what those keys were. THINK Pascal 4.0 lets you see them. After you hold down the Command key for a short time, THINK Pascal shows the command-key equivalents next to the options and buttons. For example, this is what the **Find...** dialog looks like after you hold down the Command key:

The screenshot shows a 'Find' dialog box with the following elements:

- Search for:** A text field containing 'IMCF_INIT'.
- Replace with:** An empty text field.
- Buttons:** 'Find ⌘F', 'Don't Find ⌘D', and 'Cancel ⌘.'.
- Options:**
 - ☒ Whole Words ⌘W
 - ☐ Multi-File Search... ⌘A
 - ☐ Match Case ⌘M

Shift key menu options

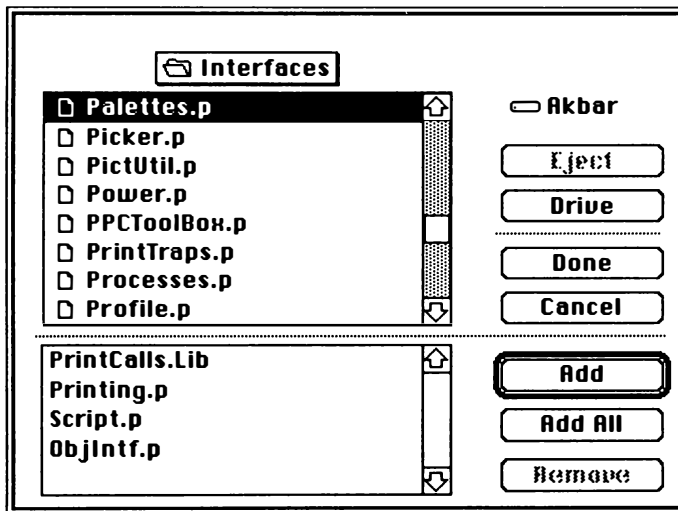
In previous versions of THINK Pascal, many commands performed different actions when you held down the Shift key. In THINK Pascal 4.0, these commands now have different names when you hold down the Shift key. When you hold down the Shift key, **Monitor** (in the **Debug** menu)

becomes **Use Monitor**, **LightsBug** (in the **Debug** menu) becomes **New LightsBug**, and **Check Syntax** (in the **Run** menu) becomes **Compile**. Here is what these commands do:

- The **Use Monitor** command lets you choose which debugger you use when THINK Pascal comes to an exception.
- The **New LightsBug** command creates a new LightsBug window.
- The **Compile** command compiles the current edit window and updates the project document.

Adding multiple files to your project

The **Add Files...** dialog lets you add several files to your project at once. To see **Add Files...**, hold down the Option key as you select the **Project** menu. For more information, see "Adding Files to Projects" in Chapter 7, "Working with Projects."

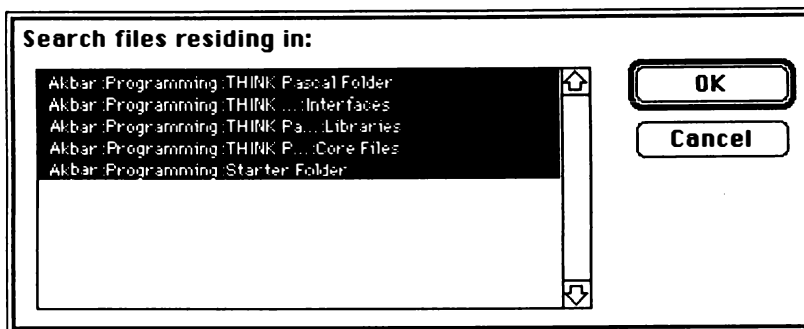


Multi-file searching

Multi-file searching is now faster and gives you more options:

- **Faster** THINK Pascal now searches through your files faster.
- **Specifying Folders** You can specify in which folders THINK Pascal searches, in the dialog below.

- **Find in All Files Command** The **Find in All Files** command automatically opens all the files that contain the search string. Choosing the **Find All Files** command is like choosing the **Find Next File** command several times until you open all the files that contain your search string. To see the **Find in All Files** command, hold down the Option key and select the **Search** menu.

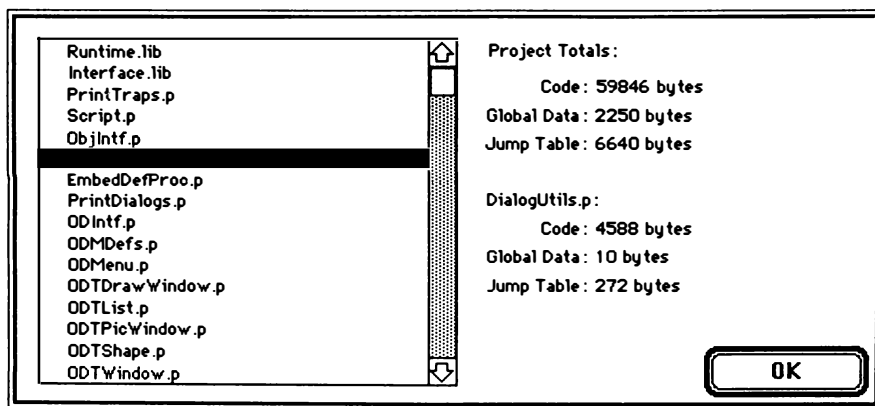


For more information, see “Searching in more than one file” in Chapter 6, “Editing.”

Other new commands

THINK Pascal contains these other new commands:

- **Quietly Auto-Reset** If you're running a program under THINK Pascal and you try to change your source files or the project, THINK Pascal displays a warning and asks if you want to cancel your change or reset the program. If this option is on, THINK Pascal won't warn you and will automatically reset your program. This command is in the **Debug** menu.
- **Pull All Stops** This command removes all Stop Signs from all the files in the project. To see this command, hold down the Option key as you select the **Debug** menu.
- **Get Info** This command shows how much code and data each file in your project produces. It's in the **Project** menu. It brings up this dialog:



Enhanced THINK Class Library

THINK Pascal 4.0 includes the THINK Class Library 1.1, which contains many new and improved classes, such as

- Improved text-handling, including more Undo support and a styled text pane
- System 7.0 support, including AppleEvents, aliases, and more
- Improved error-handling
- Larger panes, with 32-bit coordinates
- Improved printing support
- Multi-window document support.
- Classes for dialogs, tables, and pop-up menus

For more information, see the *Object-Oriented Programming* Manual, Chapter 7, "Using the THINK Class Library."

Other Changes

THINK Pascal includes many other changes, including a faster compiler, larger limit on open windows, and an easier-to-use profiler.

Faster compiler

THINK Pascal 4.0 now compiles many projects faster, by as much as 20% to 30%.

New limit on open windows

You can now work with more open windows and documents at a time. THINK Pascal 4.0 lets you have up to 16 windows open, instead of 10.

Easier-to-use profiler

Profiling your code is now easier under THINK Pascal 4.0. You don't need to add any function calls to your program to use the profiler. You just need to turn on the "Profile" option in the **Compile Options...** dialog. The profiler is also more accurate, measuring time in milliseconds, and lets you profile stand-alone applications. For more information, see Chapter 19, "The Profiler."

ANS Pascal Compatibility

B

Introduction

This appendix describes the relationship between THINK Pascal and the requirements of ANSI/IEEE770X3.97-1983, American National Standard Pascal (ANS Pascal). (*An American National Standard IEEE Standard Pascal Computer Programming Language* (IEEE, Wiley-Interscience))

Topics covered in this appendix

- Exceptions to ANS Pascal requirements
- Extensions to ANS Pascal
- Implementation-dependent features
- Treatment of errors

Exceptions to ANS Pascal Requirements

THINK Pascal complies with the requirements of ANSI/IEEE770X3.97-1983 with the following exceptions:

- In ANS Pascal, the special-symbol @ is an alternative representation for the special-symbol ^, and is required to be treated identically to ^ wherever it appears. In THINK Pascal, the special-symbol @ is an operator and is never treated identically to ^.
- In ANS Pascal, identifiers may be of any length and all characters are significant. In THINK Pascal, all characters in identifiers are significant, but the largest identifier is restricted to 255 characters.
- In ANS Pascal, a character-string of length 1 is a char-type value and a character-string of length n is a value of a packed-string-type (a **packed array** [1..n] of char — referred to as a string-type in ANS Pascal) with n components. In THINK Pascal, all quoted character-strings are string-type values. However, the compatibility and assignment-compatibility rules in THINK Pascal make its behavior with respect to character-strings compatible with ANS Pascal.
- In ANS Pascal, all values of a tag type must appear once for a given variant part. In THINK Pascal, this requirement is not enforced.
- In ANS Pascal, a function block must contain at least one assignment statement assigning a value to the function identifier. In THINK Pascal, this requirement is not enforced.
- In ANS Pascal, a field that is the selector of a variant part of a record may not be an actual variable parameter. In THINK Pascal, this requirement is not enforced.

- In ANS Pascal, no statements that threaten the value of a control-variable of a for-statement are allowed. In THINK Pascal, this requirement is not enforced. Changing the value of the control variable produces indeterminate results.
- In THINK Pascal, the functions `pack` and `unpack` are not available.
- In THINK Pascal, only the standard file variables `input` and `output` are allowed as program parameters.
- In THINK Pascal, the range of integers is `-32768..32767` (`-maxint-1..maxint`)
- In THINK Pascal, the `mod` performs a simple remainder, not a true modulo operation.

Note: There is no automatic means, in THINK Pascal, of determining whether or not a program violates any of the exceptions listed above.

Extensions to ANS Pascal

The following THINK Pascal features are extensions to Pascal as specified by ANSI/IEEE770X3.97-1983:

Note: These extensions are described more fully in Appendix C, "Porting to THINK Pascal."

- The following are word-symbols in THINK Pascal:

implementation	otherwise
inherited	string
inline	unit
interface	uses
object	
- An identifier may have an underscore appearing anywhere following the initial letter of the identifier.
- THINK Pascal supports relaxation of the ordering of declarations. There may be any number of declaration parts in any order.
- THINK Pascal supports the additional integer-type `longint` and the additional real-types `double`, `computational`, and `extended`.
- A signed constant-identifier may denote a value of type `integer`, `longint`, or `extended`.
- In THINK Pascal, the result of arithmetic performed on `integer` operands is `integer`. The result of arithmetic performed on `longint` operands is `longint`. All mixed `integer` and `longint` operands are converted to `longint` before arithmetic is performed, and the result is `longint`. A `longint` value may be used wherever an `integer` value is required if the value falls in the range `-maxint..maxint`.

- All integer-type and real-type operands are converted to extended before any real arithmetic is performed, and the result is always extended. An extended value may be used wherever a real, double, or computational value is required, provided the value falls within the range of values permissible.
- THINK Pascal supports string-types, which are compatible with other string-types, packed-string-types, and char-type.
- In THINK Pascal, the assignment-compatibility rules have been extended to allow the mixing of string-types, packed-string-types, and char-type where appropriate.
- The result-type of a function is not restricted to simple and pointer-types; functions may return values of any type.
- Individual char-type components of string-type variables and constants may be referenced as though the string were a one-dimensional array.
- String-types may be compared with char-type and packed-string-type values.
- The @ operator is provided for obtaining the address of a variable, procedure, or function.
- THINK Pascal has an optional **otherwise** clause for the case-statement.
- THINK Pascal supports the use of the indefinite-string-type, i.e. the word-symbol **string**, as a value or variable-parameter type.
- An optional second parameter may be given to **reset** and **rewrite** to associate a file variable with an external file.
- Instead of **reset** or **rewrite**, a file may be opened with **open** to allow random read/write access to a file.
- An explicit **close** procedure is supplied for those file variables associated with external files.
- A **seek** procedure may be used for random-access to file components.
- A **filepos** function returns the component number of the current file position, a value that may be used in subsequent **seeks**.
- String-type and enumerated-type values may be read from textfiles with **read** and **readln**.
- String-type and enumerated-type values may be written to textfiles with **write** and **writeln**.
- Lazy I/O is used to permit interactive and non-interactive I/O to be treated identically.
- In THINK Pascal, the **ord** function may be applied to a pointer-type value, facilitating address arithmetic.

- The `ord4` function is provided in THINK Pascal for converting an ordinal-type or pointer-type value to a `longint`.
- The `pointer` function is provided in THINK Pascal for converting an integer-type value to a pointer-type value.
- THINK Pascal includes a set of string procedures and functions.
- The `sizeof` function is provided for obtaining the storage size of a variable or type.
- Inline routines are supported for imbedding machine code in a program.
- Units are supported for modular construction of programs and separate compilation.
- THINK Pascal supports type-casting of values.
- THINK Pascal supports all of the Macintosh Toolbox/OS constants, types, variables, procedures, and functions as predefined identifiers.
- THINK Pascal supports Object Pascal extensions as described in *Object Pascal Report*, by Larry Tesler. Apple Computer 1985.
- THINK Pascal allows constant expressions in `const` declarations.
- THINK Pascal allows ranges for case labels.
- THINK Pascal supports short circuit boolean operators `&` and `|`.
- THINK Pascal supports the predefined `cycle`, `exit`, `leave`, and `halt`.

Note: There is no automatic means, in THINK Pascal, of distinguishing between a program that uses extensions and one that does not.

Implementation-Dependent Features

The effect of using an implementation-dependent feature of Pascal, as defined by ANSI/IEEE770X3.97-1983, is unspecified as described in the preface to the Language Reference.

Treatment of Errors

This section defines those errors listed in Appendix D of the ANS Pascal standard that are *not* automatically detected and reported by THINK Pascal. The number of each error listed below is the number under which the error is listed in the standard's appendix. The wording of the description of the error, however, differs from the wording in the standard.

2. If `t` is the tag-field of a variant-part, and if `f` is a field within the currently active variant of that variant-part, then it is an error to attempt to alter the value of `t` while a reference to `f` exists.

4. If p is a pointer-type value, it is an error to reference p^{\wedge} if the value of p is undefined.
5. If p is a pointer-type value, it is an error to attempt to dispose of p while a reference to p^{\wedge} exists.
6. If f is a file-type variable, it is an error to attempt to close f or alter the current file position of f while a reference to f^{\wedge} exists.
19. If a pointer-type variable p is assigned a value by `new (p, c1, c2, . . . , cn)`, it is an error to attempt to make any other variants than those selected by the case-constants in `new` become the active variant.
20. If a pointer-type variable p is assigned a value by `new (p, c1, c2, . . . , cn)`, it is an error to attempt to dispose of p without supplying the same list of case-constants in the same order.
21. Same as 20.
22. Same as 20.
24. It is an error to attempt to dispose of p if the value of p is undefined.
25. If a pointer-type variable p is assigned a value by `new (p, c1, c2, . . . , cn)`, it is an error to reference the entire variable p^{\wedge} in an expression, as an actual variable-parameter, or as the destination of an assignment-statement.
37. For `chr (x)`, the function returns a result of char-type which is the value whose ordinal number is equal to the value of the expression x if such a character exists. It is an error if such a character value does not exist.
38. For `succ (x)`, the functions yields a value whose ordinal number is one greater than that of x , if such a value exists. It is an error if such a value does not exist.
39. For `pred (x)`, the functions yields a value whose ordinal number is one less than that of x , if such a value exists. It is an error if such a value does not exist.
43. It is an error to reference a variable in an expression if the value of that variable is undefined.
48. It is an error if the result of the activation of a function is undefined when that activation is complete.
51. It is an error if none of the case constants is equal to the value of the case index upon entry to a case statement.

Porting to THINK Pascal C

Introduction

Pascal was designed by Professor Niklaus Wirth primarily as a tool for teaching systematic programming. It has evolved into one of the most popular programming languages, the main attraction being its emphasis on rigid structure and strong typing. In the evolution process most Pascal implementations have a number of language extensions. Since most extensions are unique to an implementation, you may need to do some work to port Pascal programs from one computer to another.

Different implementations have interpreted the semantics of the unadorned language in different ways. This issue was addressed by the Joint ANSI/X3J9-IEEE Pascal Standards Committee, which in 1982 approved a standard language (informally called ANS Pascal) to promote the portability of Pascal programs between implementations.

THINK Pascal is intended to conform, as closely as possible, to ANS Pascal (see Appendix B), while providing extensions necessary for the development of serious Macintosh applications. THINK Pascal tries to keep such extensions to a minimum, choosing features common to many Pascal implementations, while retaining compatibility with existing systems, notably Macintosh Programmer's Workshop (MPW) and Macintosh Pascal.

This appendix is designed to help you port existing programs to THINK Pascal. It provides a comprehensive list of areas in which THINK Pascal may differ from other implementations, along with suggestions for modifying programs to run under THINK Pascal.

Topics covered in this appendix

- Identifier length
- Reserved words
- Comments and directives
- The **uses** clause
- Types
- Data representation
- Data initialization
- Operators
- Integer arithmetic
- Program parameters
- Predefined procedures and functions
- Standard units
- Input/output
- Run time environment
- Extensions
- Using .o files

Identifier Length

Most Pascal compilers only check the spelling of identifiers up to 8 or 16 characters. In THINK Pascal, all characters in an identifier are significant, up to the maximum length (255). For example, many compilers don't report an error with this code:

```
var
    aVeryVeryLongName: integer;
begin
    aVeryVeryLongNme := 0;
end;
```

THINK Pascal reports the misspelling as an undeclared identifier.

Reserved Words

THINK Pascal has added the following word-symbols which may not be used as identifiers:

implementation	otherwise
inherited	string
inline	unit
interface	univ
object	uses

If you try to use one of these reserved words as an identifier, THINK Pascal reports an error.

Comments and Directives

Some Pascal compilers treat the comment delimiter pairs { } and (* *) differently, allowing comments which use one delimiter to be "commented out" with the other delimiter. For example, in MPW Pascal the code sequence

```
if ResError <> noErr then {report resource error}
    alertStatus := Alert (MyRsrcAlert, nil);
```

could be commented out like this:

```
(* if ResError <> noErr then {report resource error}
    alertStatus := Alert (MyRsrcAlert, nil);
*)
```

THINK Pascal treats the different comment delimiter pairs the same, so you can't nest comments. Also, comments may not cross line boundaries. THINK Pascal converts multi-line comments into several single line comments. Nested comments may not be converted correctly.

You can use the conditional compilation commands to comment out large blocks of code or to simulate multi-line comments like this:

```
{ $IFC FALSE}
...
  code to ignore
...
{ $ENDC}
```

Compiler directives

Most Pascal implementations support special **compiler directive** comments. These comments are typically of the form { \$x+ } or { \$x- }, where x is a one or two character mnemonic for the particular option. THINK Pascal supports a number of compiler directives; these are described in detail in Chapter 15.

Compiler directives are not addressed in the ANS Pascal standard, so they vary greatly between implementations. For example, the MPW Pascal compiler supports:

{ \$I <i>filename</i> }	include <i>filename</i> in text
{ \$S <i>segmentname</i> }	place code in segment <i>segmentname</i>
{ \$OV+ }	turn on overflow checking
{ \$R- }	turn off range checking

MPW also supports other compiler directives, most of which have no analog in THINK Pascal. It is best to assume that all compiler directives are not portable, and should be changed or removed.

Note: You can enable and disable most of the THINK Pascal compiler directives from the Project window.

THINK Pascal doesn't support the { \$I *filename* } directive. However, when you turn on the "USES Extensions" option, you may be able to replace a { \$I } directive in an MPW Pascal program with a **uses** statement in THINK Pascal. For more information, see Chapter 10, "Units and Libraries."

The Pascal Source Converter helps translate programs written for Apple's MPW Pascal for use with THINK Pascal. It converts MPW Pascal directives to THINK Pascal directives, processes MPW Pascal \$I include directives, and more. For more information, see Chapter 20, "The Pascal Source Converter."

The Uses Clause

Like many other Pascal compiler, THINK Pascal has a **uses** clause to specify the dependencies between the files that make up a program. When you turn on the "USES Extensions" option in the **Compiler Options...** dialog, THINK Pascal interprets the **uses** clause differently from many other compilers. If your program must be compatible with one of these compilers, be sure to turn off the "USES Extensions" option. For more information, see Chapter 10, "Units and Libraries."

Types

Different implementations of Pascal place different limitations on predefined and user defined data types. These limitations typically reflect the chosen representation of data, or constraints imposed by the underlying hardware architecture. For example, the ANS standard requires that the type `integer` contain all values in the range `-maxint` to `maxint`, but accepts the fact that these values are machine dependent and may differ across implementations.

THINK Pascal imposes the following data type limitations:

- Integers are limited to the range `-32768` to `32767`.
- Enumerated types may contain no more than 256 distinct values.

You can declare array types larger than 32K, but the total size of all the variables in a block cannot exceed 32K. Suppose you wanted to create a large array. Here's one way:

```

type
  BigArray = array [0..99999] of integer;
  BigArrayPtr = ^BigArray;
  BigArrayHandle = ^BigArrayPtr;

var
  myBigArray : BigArray;           { this is NOT allowed! }
  myBigArrayH : BigArrayHandle;    { but this is }
  i : longint;

begin
  myBigArrayH := BigArrayHandle(NewHandle(sizeof(BigArray)));
  for i := 0 to 99999 do
    myBigArrayH^[i] := 0;
end;
```

There are also areas in which THINK Pascal exceeds limitations imposed by other Pascal implementations. For example:

- Support for the full `set of integer` including negative-value elements.
- Support for floating-point values of single, double, computational and extended precision.
- String-types, packed-string-types, and char-type are completely compatible.

For a complete description of all THINK Pascal data types, refer to Section 3 of Chapter 17, "Language Reference."

Data Representation

The way a Pascal compiler represents data depends on hardware differences and personal judgment. THINK Pascal follows the Macintosh Toolbox conventions for ordinal types. For example, the `integer` type is a 16 bit value to take advantage of the more efficient 16 bit instructions of the

MC68000. The 32 bit integer type `longint` is provided for dealing with values which exceed the range of 16 bit integers.

The data type `char` is also implemented as a 16 bit (rather than the more natural 8 bit) value. It occupies 8 bits only when packed. There are some subtle ramifications of this implementation. The predefined type `text` is usually assumed to have the same representation as **file of char**; Macintosh Pascal assumes that this is the case. In THINK Pascal, `text` has the same representation as **packed file of char**. This may cause difficulty when porting programs which use **file of char**.

The representation of data in THINK Pascal is described more completely in Chapter 13, "Assembly Language."

Representation of packed records

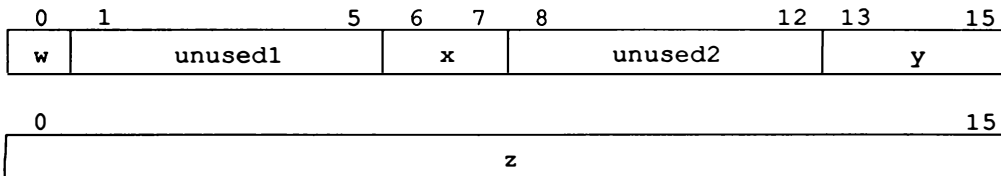
The different Pascal compilers for the Macintosh use different packing algorithms. For most packed records — including all those used by the Toolbox routines — the algorithms pack records the same way. The differences cause a problem only when you write a file of packed records with a program created with one compiler and try to read the file with a program created with the other compiler.

If you must be certain that your data structures are the same between compilers, always specify dummy fields to pad out the unused bits. For example, this declaration creates a structure that looks the same, no matter which compiler you use:

```
type
  Color = (Red, Blue, Green);
  Day = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

  MyPackedRecord = packed record
    w:      Boolean;      { 1 bit  }
    unused1: 0..31;       { 5 bits }
    x:      Color;        { 2 bits }
    unused2: 0..31;       { 5 bits }
    y:      Day;          { 3 bits }
    z:      Integer;      { 16 bits }
  end;
```

This record for that declaration looks like this:



Data Initialization

Some Pascal implementations guarantee specific values for uninitialized variables. For example, Macintosh Pascal initializes all global and local variables (and function results) to 0. THINK Pascal initializes only global variables to 0. Programs which (inadvertently) depend on other variables being initialized may run into unusual run-time behavior.

Operators

The implementation of operators in THINK Pascal conforms for the most part to the ANSI standard, differing only in areas where THINK Pascal provides additional data types (e.g. **string**, **longint**, **extended**). Porting difficulties may occur in expressions involving integer arithmetic overflow (see the following section).

For compatibility with MPW Pascal and other Pascal compilers, THINK Pascal implements the **mod** operator as a simple remainder operator.

Note: THINK Pascal 1.0 implemented the **mod** operator according to the rules for modulo arithmetic.

Many compilers support an exponentiation operator ****** for integer or floating-point operands. In THINK Pascal, floating-point exponentiation can be easily implemented, based on the following identity:

$$x ** y = \text{Exp} (\text{Ln} (x) * y)$$

If you port a program which uses integer exponentiation, you may have to write a simple exponentiation function.

Integer Arithmetic

THINK Pascal performs all integer arithmetic in either 16 bits or 32 bits depending on the type of the operands. For binary operators, if both operands are 16 bits (or smaller) the operation is performed in 16 bits; if either or both operands is 32 bits, the operation is performed (less efficiently) in 32 bits.

Consider the implications of these arithmetic rules on this piece of code:

```
var
    i, j: integer;
begin
    i := 20000;
    j := 30000;
    writeln (i+j);
end;
```

This program fragment will cause an overflow error in THINK Pascal because the sum 20000+30000 is too large to be represented in 16 bits. Other implementations (notably Macintosh Pascal) may perform the operation in 32 bits, avoiding the overflow at the expense of execution efficiency. You can force the arithmetic to be performed in 32 bits by casting the operands to `longint` or by using the `Ord4` function.

A similar problem arises in programs ported from Macintosh Pascal which use inline procedures and functions (`InlineP`, `BInlineF`, `WInlineF`, `LInlineF`). The statement

```
InlineP ($A9D1, 0+0, 65535, hTE);
```

attempts to call `TESetSelect` with a longword 0 for `selStart`. In THINK Pascal, this results in a *word* 0 being pushed instead, misaligning the stack and (probably) resulting in a fatal error. You can cast the integer value to a `longint`. Statements like the one above are better replaced by a call to the equivalent Macintosh Toolbox routine:

```
TESetSelect (0, 65535, hTE);
```

This call performs the proper argument type checking and conversion, and it's easier to understand and more efficient.

Program Parameters

THINK Pascal only allows the standard file-variables `input` and `output` as program-parameters. Occurrences of other file-variables in a program heading are reported as errors. These can be removed without affecting the semantics of the program.

If you are porting code from a compiler that automatically opens up program parameters as file variables, you need to do a little rewriting. For example, if your program statement looks like this:

```
program MyProgram (data, input, output);
```

you have to rewrite it like this:

```
program MyProgram (input, output);
var
  data: text;
  ...
begin
  open(data, 'data file');
  ...
  close(data);
end.
```

Predefined Procedures and Functions

THINK Pascal, like Macintosh Pascal, provides a large number of predefined procedures and functions in addition to those required by the standard. Routines have been added to facilitate address

arithmetic, string handling, I/O, and other common tasks. On top of this, nearly all of the procedures and functions (and constants and types) described in *Inside Macintosh* are supplied as predefines (some are supplied as units: see below).

Different implementations of Pascal are likely to provide different predefined procedures and functions, many of which have functional analogs in THINK Pascal. Programs that depend on non-standard predefines will require modification. For example, the following MPW Pascal predefines are not supported:

fillchar	release
mark	scaneq
moveleft	scanne
moveright	

THINK Pascal also does not support the functions `pack` and `unpack`, which packs an unpacked array and unpacks a packed array.

For a complete description of the THINK Pascal predefines, see §10 of Chapter 17, "Language Reference."

Toolbox Interfaces

The interface files in the `Interfaces` folder are compatible with the latest from Apple. These are the interfaces discussed in Chapter 11. If you're porting an older program over, you may need to change some of the interface files. In particular,

If you used...	Now use...
Memtypes.p	Types.p
OSIntf.p	Files.p, Devices.p, etc.
ToolIntf.p	Windows.p, Menus.p, etc.
PackIntf.p	Packages.p
MacPrint.p	Printing.p
PickerIntf.p	Picker.p
SCSIIntf.p	SCSI.p
VideoIntf.p	Video.p

For more information on the THINK Pascal Toolbox interfaces, see "Calling Macintosh Toolbox Routines" in Chapter 11.

Standard Units

Pascal compilers for the Macintosh typically provide a standard set of units, libraries, or include files which define some or all of the constants, types, procedures, etc., described in *Inside Macintosh*, and whose declarations can be made available to any program or unit. For example, most MPW Pascal programs begin with the following **uses** clause:

```
uses
    MemTypes, QuickDraw, OSIntf, ToolIntf, PackIntf;
```


Similarly, Macintosh Pascal programs often start with:

uses

QuickDraw1, QuickDraw2;

In THINK Pascal, most of the declarations in *Inside Macintosh* are predefined, and these **uses** clauses are not required.

Note: You can use MemTypes, QuickDraw, OSIntf, ToolIntf, and PackIntf if you add these files to your project. (They're provided in your THINK Pascal package in either the Interfaces or Old Interfaces folders.) Most of these files contain no definitions. They're included to make it easier to port from MPW Pascal.

Some of the *Inside Macintosh* declarations, such as the AppleTalk Manager, *are* supplied as libraries. Each is supplied as two files: a source (interface) file, and a library (code) file. Both of these files must be added to the project. The interface to the library is in the form of a **unit**. The unit name must appear in a **uses** clause in the file that accesses the library.

For more information on THINK Pascal's Toolbox interfaces, see Chapter 11, "Using Predefined Routines." The exact syntax for units is described in §8.3 of Chapter 17, "Language Reference."

Input/Output

This is an area in which many Pascal implementations diverge, particularly in their treatment of interactive I/O. THINK Pascal supports I/O with ANS Pascal semantics and the following extensions:

- An optional second parameter may be given to `reset` and `rewrite` to associate a file variable with an external file. A file opened with `reset` or `rewrite` is read-only or write-only respectively, and in both cases the file may only be accessed sequentially.
- Instead of `reset` and `rewrite`, a file may be opened with `open` to allow random read/write access.
- The `seek` procedure may be used for random-access to file components.
- The `filepos` function returns the component number of the current file position, a value that may be used in a subsequent `seek`.
- String-type and enumerated-type values may be read from text files using `read` and `readln`.
- String-type and enumerated-type values may be written to text files using `write` and `writeln`.
- Lazy I/O is used to permit both interactive and non-interactive I/O to be handled identically.

THINK Pascal is identical to Macintosh Pascal in its treatment of I/O. Other compilers are likely to have different extensions. Programs which read and write to the predefined file variables `input` and `output` interactively may cause some difficulty; see Section 9 of Chapter 17, "Language Reference," for a complete explanation of Lazy I/O and its ramifications.

Run Time Environment

Macintosh programs usually begin with a sequence of initialization calls to various portions of the Macintosh Toolbox. In THINK Pascal, the following initializations are performed automatically for your application:

```
InitGraf (@thePort);
InitFonts;
InitWindows;
InitMenus;
TEInit;
InitDialogs (nil);
SetApplLimit (current value of A7 - Run Options... stack size);
for i := 1 to 10 do
    MoreMasters;
```

If you initialize the Macintosh Toolbox manager explicitly in your program, use the `{ $I- }` directive to turn off the automatic initialization (see Chapter 15, "Compiler Directives").

Extensions

THINK Pascal supports several extensions to the Pascal language.

Control procedures

These procedures let you control the flow of your program without resorting to `goto` statements.

<code>cycle</code>	go to the next repetition of the enclosing while , repeat , or for statement
<code>leave</code>	go to the statement following the enclosing while , repeat , or for statement
<code>exit (procname)</code>	exit the named (enclosing) procedure
<code>halt</code>	exit the program

Constant expressions

Constant expressions are allowed in `const` declarations. They must evaluate to a set, string-type, integer, char, boolean, or enumerated type.

Example:

```
const
    limit = maxSize - 1;
    CR = Chr ($0D);

    letters = ['a'..'z', 'A'..'Z'];
    digits = ['0'..'9'];
    alphanumerics = letters + digits;
```

Case label subranges

A case label in a **case** statement may be a range of constant values:

```
case today of
    monday..friday: ...
    saturday: ...
    sunday: ...
end;
```

Generalized function results

Function calls may be treated as simple variables or used as l-values (variable-references appearing on the left-hand side of an assignment).

Example:

```
function InfoScrap: PScrapStuff;      {a predefined Toolbox routine}
external;

...
var
    myHandle: Handle;
begin
    myHandle := InfoScrap^.scrapHandle;
    ...
    with InfoScrap^ do
        ...
```

Inside a function, be careful when you recursively call the function in an l-value context. If the function name is qualified by "^", the function is called recursively and the qualifier is applied to the return value. If the function name isn't qualified or is qualified by "." or "[...]", THINK Pascal assumes you are assigning the return value. For example:

```
function Foo: Ptr;
begin
    Foo := @Bar;      { Makes @Bar the function's return value. }
    Foo^ := 0;        { Calls Foo recursively and assigns 0 to      }
                     { the byte that the result points to.        }
end;
```

Type casting

Type casts may be applied to l-values (variable-references appearing on the left-hand-side of an assignment, or as an actual **var** parameter) as well as r-values (variable-references appearing in expressions). For example:

```
var
    myHandle: Handle;
    hTE: TEHandle;
    p: Point;
begin
...
    myHandle := hTE^.hText;
    CharsHandle (myHandle)^^[0] := Chr($0D);
    DrawChar (CharsHandle (myHandle)^^[0]);
    DisposHandle (myHandle);
...
    p := Point(0);
...
```

Short circuit Booleans

THINK Pascal uses the **&** and **|** operators for minimum evaluation of boolean expressions. In this statement:

```
if (p <> nil) & (p^.name = '') then ...
```

The second test (**p^.name = ''**) is performed only if the first test (**p <> nil**) evaluated to true.

Using .o Files

THINK Pascal can read .o files, but it can't use some .o files made with high-level languages (like MPW C or Pascal). These .o files call routines in the compiler's runtime library, even though there may not be explicit calls in the source code. These calls create link errors. For example, to handle operations on sets or 32-bit integers, most Pascal compilers call the runtime library. To avoid these problems, port .o files created with C or Pascal by compiling the source code with THINK C or Pascal.

Note: Assembly language is *not* a high-level language, so THINK Pascal can use a .o file created with it.

For the same reason, other compilers can't use some THINK Pascal libraries, even though THINK Pascal creates libraries in a .o-compatible form.

Error Messages

D

Introduction

This appendix is a guide to the error messages THINK Pascal generates. Compile, link, and the common runtime error messages, with their explanations, are arranged in alphabetical order. There are cross references to other chapters in this manual and examples of incorrect and correct code fragments.

Error messages appear in an alert box, unless an error occurs while evaluating an expression in the Observe window. In that case, an abbreviated error message appears in the left cell of the Observe window.

The examples have also taken advantage of THINK Pascal's relaxation of order and number of declarations in a block. The examples assume that the following constants, types, and variables are defined::

```
const
    aConstant = 42;
    aCharConstant = 'c';
    aStringConstant = 'hi there';
type
    ColorType = (red, orange, yellow, green, blue, violet);
    WeightType = (LIGHT, MEDIUM, HEAVY);
    aType = integer;
var
    aBool : boolean;
    aChar : char;
    anInt : integer;
    aLong : longint;
    aReal : real;
    aDouble : double;
    anExt : extended;
    aString : string;
    aColor : ColorType;
    aPtr : Ptr;                                { Ptr is a predefined type      }
                                              { that points to SignedByte    }

    ColorSet : set of ColorType;
    TextFile : Text;
    FileOfInt : file of integer;
```

action will Reset your program. Continue anyway?

While your program is running under the THINK Pascal environment, THINK Pascal stops running your program if you perform certain actions, including editing your program, modifying the project, closing the project, or quitting THINK Pascal). If you click Yes, THINK Pascal will continue

the action and stop your program. If you click No, THINK Pascal will cancel the action and let your program continue running.

"symbol" is already declared at this level.

See 2.2.4 of Chapter 17. Example:

```
var
    aDupName : integer;
    aDupName : real;           { Error: aDupName already declared }

type
    recordType = record
        aDupName : integer;    { OK: first time in the record }
        aDupName : real;      { Error: already declared in record }
    end;

procedure aDupName;           { Error: already declared as a variable }
begin
end;

procedure P1 (aDupName:char); { OK: aDupName is not declared }
                        { at this level }
begin
end;

procedure P2 (X:char; X:real); { Error: X declared twice }
                        { in parameter list }
begin
end;
```

"symbol" is not declared.

- Check your spelling. Be careful to distinguish between 1 (one), l (lower case L), and I (upper case i); and between 0 (zero) and O (oh). (Pascal is not case sensitive.)
- Make sure the symbol is declared before it is used.
- Make sure the symbol is visible at the level you are trying to use it.
- If the symbol is in another unit, don't forget to use the unit name in a **uses** clause.

In almost all cases, a name must be defined (or predefined) before it can be used. See Sections 3, 4, and 7 of Chapter 17. Example:

```
program HasUndeclaredSymbols;
type
    BadType = TypeYetToBeDefined; { Error }
    TypeYetToBeDefined = integer; { This must come before }
                                { the previous statement }
    RecPtrType = ^RecType;        { OK: a pointer type to a }
                                { type to be defined later }
    RecType = record
        PtrToARecord : ^UndefinedType; { Error }
        PtrToNextRecord : RecPtrType; { OK }
        aField : integer;
    end;
```

```

var
    r : RecType;
procedure Proc;
    var
        PrivateToProc : integer;
begin
end;
begin
    PrivateToProc := 4;
    aField := 4;
    r.aField := 4;
    UndeclaredVariable := 4;
    UndeclaredProcedure;
    r.aField := UndeclaredFunc;
end.

```

{ Error: PrivateToProc isn't }
 { declared at this level }
 { Wrong }
 { Right }
 { Error }
 { Error }
 { Error }

“symbol” Isn’t In the current project, hasn’t been successfully compiled, or is in the wrong build order.

When you use a unit, the unit must be compiled in the project. The unit being used must appear before the units that use when you see the project by build order. See Chapter 7 and §8.5 of Chapter 17. Example:

```

unit anUnit;
interface
    uses
        NonExistantUnit, UncompiledUnit, UnitFollowingThisOne;
implementation
end.

```

{ Error }

“symbol” looks like it’s being used as a function, but it isn’t a function name.

See 5.2 of Chapter 17. Example:

```

program test;
const
    x = 5;
var
    i: integer;
begin
    i := x;
    i := x(1);
end;

```

{ Correct }
 { ERROR }

“symbol” looks like it's being used as a procedure, but it isn't a procedure name.

See 6.1.2 of Chapter 17. Example:

```

type
  IntPtrType = ^integer;
var
  anIntPtr : IntPtrType;
function Func : char;
begin
end;

begin
  aChar;                                { Error }
  aChar := Func;                        { Right }
  anIntPtr := pointer(aPtr);           { See §10.2.6 of Chapter 17 }
  anIntPtr^ := 3                       { Right: use temp variable anIntPtr }
end;

```

“symbol” was previously declared as a function, not a procedure.

When a function is declared in an **interface** part or as a forward function, it cannot be defined later as a procedure. See 7.1.1 and 8.1 of Chapter 17. Example:

```

unit MisdefinedFunctions;
interface
  function InterfaceFunc : char; { To be defined in IMPLEMENTATION }
implementation
  procedure InterfaceFunc;       { Wrong: FUNCTION declaration expected }
  begin
  end;
  function InterfaceFunc;       { Right }
  begin
  end;
  function ForwardFunc : char;  { To be defined below }
  forward;
  procedure ForwardFunc;       { Wrong: FUNCTION declaration expected }
  begin
  end;
  function ForwardFunc;       { Right }
  begin
  end;
end.

```


“symbol” was previously declared as a procedure, not a function.

When a procedure is declared in an **interface** part or as a forward procedure, it cannot be defined later as a function. See 7.1.1 and 8.1 of Chapter 17. Example:

```
unit MisdefinedProcedures;
interface
  procedure InterfaceProc;           { To Be defined in IMPLEMENTATION part }
implementation
  function InterfaceProc : char;      { Wrong: PROCEDURE definition }
                                     {      expected      }

  begin
  end;
  procedure InterfaceProc;           { Right }
  begin
  end;

  procedure ForwardProc;             { To Be defined below }
  forward;
  function ForwardProc : char;        { Wrong: PROCEDURE definition }
                                     {      expected      }

  begin
  end;
  procedure ForwardProc;             { Right }
  begin
  end;
end.
```

@ can only be applied to variable references, or to top-level procedure or function names.

The address operator @ can only be applied to objects that actually have memory allocated to them. For example, constants, types, etc., do not allocate memory. Furthermore, @ cannot be applied to sub-level subroutines, because references to the variables in the outer scope cannot be set up properly. See 5.1.6 and 5.1.6.4 of Chapter 17. Example:

```
program AtSigns;
  procedure Proc;
    procedure SubProc;
    begin
      aPtr := @Proc;           { OK }
      aPtr := @SubProc;        { Error: SubProc is not }
                               {      a top level procedure }
    end;
  begin { Proc }
  end;
begin
  aPtr := @anInt;              { OK: memory is allocated for anInt }
  aPtr := @Proc;               { OK }
  aPtr := @aConstant;          { Error: aConstant is a constant }
  aPtr := @orange;             { Error: orange is an enumerated constant }
  aPtr := @ColorType;          { Error: ColorType is a type }
  aPtr := @integer;            { Error: integer is a type }
end.
```

@ can't be applied to a component of a packed type.

@ cannot be applied to *components* of packed structures, because the actual data representation of a component in a packed structure may be different from that actual data representation in an unpacked structure. However, @ *can* be applied to the packed structure as a whole. See 5.1.6.2 of Chapter 17.

@ can't be applied to predefined or inline routines.

The @ operator can only be applied to actual routines, not predefined, inline, or Macintosh Toolbox routines (which don't generate real subroutine calls). However, you can write a wrapper routine which wraps itself around the desired routine. Then, the @ operator can be applied to that wrapper. See 5.1.6.4 of Chapter 17. Example:

```

procedure NOP;
inline
$4e71;                { M68000 NOP instruction }
procedure WritelnWrap (s : string);
begin
    writeln(s);
end;
procedure NOPWrap;
begin
    NOP;
end;
function ButtonWrap : boolean;
begin
    ButtonWrap := Button;
end;
begin
    aPtr := @writeln;           { Wrong: writeln is predefined procedure }
    aPtr := @writelnWrap;       { Right: writeln is in a wrap routine }
    aPtr := @NOP;               { Wrong: NOP is an inline procedure }
    aPtr := @NOPWrap;           { Right: NOP is in a wrap routine }
    aPtr := @Button;            { Wrong: Button is an Toolbox routine }
    aPtr := @ButtonWrap;        { Right: Button is in a wrap routine }
end.

```

Array Index type Incompatibility.

The type of the expression that indexes an array must be compatible with the type of the index in the array declaration. See 4.3.1 of Chapter 17. Example:

```

var
    ai : array[1..10] of char;
    aC : array[ColorType] of char;
begin
    ai[4] := 'a';               { OK: 4 is within 1..10 }
    ai[4 + anInt] := 'a';       { OK: type of (4+anInt) is }
                                { compatible with 1..10 }

```

```

ai[aReal]:= 'x';           { Wrong }
ai[round(aReal)]:= 'x';    { Right }
ai[aColor]:= 'x';          { Wrong }
ai[ord(aColor)]:= 'x';     { Right: but why bypass strong typing? }
aC[aColor]:= 'x';          { Right: Best }
ai['c']:= 'x';             { Error }
ai[4 + aReal]:= 'x';       { Error: type of 4+aReal is Real, not 1..10 }
ai[succ(aColor)]:= 'x';    { Error: succ(aColor) is ColorType }
ai[ai[anInt]]:= 'x';       { Error: ai[anInt] is char, not 1..10 }
aChar:=aString[aChar];     { Wrong: string index must be 1..255 }
aChar:=aString[ord(aChar)]; { Right }

```

end;

Array Index type is not Integer, char, enumerated, or subrange.

See 3.2.1 of Chapter 17. Example:

```

var
  Good_1 : array['A'..'Z'] of char; { OK: char subrange }
  Good_2 : array[ColorType] of char; { OK: enumerated type }
  Good_3 : array[1..10] of char;     { OK: integer subrange }
  Bad_1 : array[real] of char;       { Error }

```

Assignment type Incompatibility.

You can't put a square peg in a round hole. Pascal catches illogical statements that try to assign an expression of one type into a variable of an assignment-incompatible type. On occasion, this rule needs to be broken. The following examples show common errors and methods to break the rules. Use the methods at your own risk. Assignment compatibility can be subtle. See 3.5.3 and 5.4 of Chapter 17. Example:

```

type
  CanonicalType = array[1..4] of char; { a type }
  IdenticalType = CanonicalType;       { identical to CanonicalType }
  AnotherType = array[1..4] of char;   { a type DIFFERENT from }

  { CanonicalType }

var
  AnonArray : array[1..4] of char; { an array of anonymous type }
  CanonicalArray : CanonicalType;  { a variable of a named type }
  IdenticalArray : IdenticalType;  { a variable of a same type }
  AnotherArray : AnotherType;      { a variable of a different }
                                   { named type }

  AC : array[1..4] of char;
  PAC : packed array[1..4] of char;
  Rec : packed record
    i : integer;
  end;
  pInt : ^integer;
  pChar : ^char;

begin
  aString:=AC; { Wrong: can't assign UNpacked array of char }
               { to strings }
  aString:=PAC; { Right: OK to assign pack array of char to strings }

```

```

PAC:=aString;      { Right: OK to assign strings to pack array of char }
AC:=PAC;           { Error: can't assign packed to unpacked array }
anInt:=aReal;      { Wrong: Pascal doesn't convert float to integer }
anInt:=round(aReal); { Right: explicit conversion }
anInt:=trunc(aReal); { Right }
aReal:=anInt;      { OK: Pascal converts integer to float }
aReal:=3;          { OK }
aColor:=green;     { OK }
anInt:=green;      { Wrong: can't assign enumerated type to integer }
anInt:=ord(green); { Right: but why bypass strong typing? }
aColor:= anInt;    { Wrong: can't assign integer to enumerated }
aColor:=ColorType(anInt); { Right: a CAST works but why }
                    { bypass strong typing? }
AC:=Rec;           { Error: can't assign totally different types}
pInt:=pChar;       { Wrong: can't assign different pointer types}
pInt:=pointer(pChar); { Right: a CAST works but why }
                    { bypass strong typing? }
pInt := 7;         { Wrong: can't assign a pointer a numerical value }
pInt^ := 7;        { Right }
CanonicalArray := IdenticalArray; { OK }
CanonicalArray := AnotherArray;   { Error: the types aren't }
                                   { assignment compatible }
CanonicalArray := AnonArray;      { Error: anonymous types are never }
                                   { assignment compatible }
end;

```

At A-Trap

This runtime message appears in the Observe window when the **Break At A-Traps** command is checked, and the expression being observed, directly or indirectly, calls a Macintosh Toolbox routine. See Chapter 14.

At least one comment of more than 255 characters has been truncated to 255.

THINK Pascal only supports comment lines which have fewer than 256 characters.

At least one Identifier, literal string, or other token of more than 255 characters has been truncated to 255.

THINK Pascal only supports identifiers, string literals, and other tokens which have fewer than 256 characters.

At least one valid constant declaration must follow CONST.

See 1.7 and 2.1 of Chapter 17. Example:

```

const
    BadConstant : false; { Wrong: Colon(:) instead of Equal(=) }
    GoodConstant = true; { Right }
const
    { Error: must have at least one CONST declaration }
begin
end;

```

At least one valid type declaration must follow TYPE.

See 2.1 and 3 of Chapter 17. Example:

```

type
  BadType : integer;      { Wrong: Colon(:) instead of Equal(=) }
  GoodType = integer;    { Right }
type
  { Error: must have at least one TYPE declaration }
begin
end;
```

At least one valid variable declaration must follow VAR.

See 2.1 and 4.1 of Chapter 17.

```

var
  BadVariable = integer;  { Wrong: = instead of : }
  GoodVariable : integer; { Right }
var
  { Error: must have at least one VAR declaration }
begin
end;
```

AutoInterrupt exception

This runtime message is given when the programmer's switch is hit, but a low level debugger (Macsbug or TMON) is not installed. This should only be used as a last resort. Since you cannot be sure of the state of your program, it may crash if you restart or reset it. Therefore, save all your files before continuing. It's better to run with code compiled with the Debug Option (see Chapter 15) and click on the Bug Spray Can to stop, or always have a low level debugger installed.

Available memory for variables declared at this level has been exhausted.

Because of the MC68000 architecture, the local storage per function or procedure must not exceed 32766 bytes. Similarly, the total global storage must not exceed 32766 bytes. Excessive global storage may not be detected until link time.

In practice, even less storage may be available. Subroutines require local storage for all temporary variables generated by the compiler. For applications, QuickDraw globals are included in global storage. Furthermore, libraries may also require some global storage.

If you need more memory, you must use storage allocation subroutines. See 10.1 and 10.2 of Chapter 17. For large data structures, the use of handles is highly recommended. See *Inside Macintosh*. THINK Pascal lets you declare types that are bigger than 32K, so it's easy to use the Memory Manager to get huge arrays.

```

program MemoryHog;
const
    HalfTooBig = 16500; { The Max global size is about 32000 bytes }
    TooBig = 33000;
type
    BigArrayType = packed array[0..HalfTooBig] of char;
    HugeArrayType = packed array[0..TooBig] of char;           { OK }
    HugePtr = ^HugeArrayType;
    HugeHandle = ^HugePtr;
var
    BigGlobalArray1: BigArrayType;      { OK }
    BigGlobalArray2: BigArrayType;      { Error : Too much memory      }
    { Better: }
    BigArrayPtr1: ^BigArrayType;       { Allocate Pointers instead    }
    BigArrayPtr2: ^BigArrayType;
    TooBigArray: HugeArrayType;         { Error: Way too big          }
    { Much Better }
    hTooBigArray: HugeHandle;           { OK }
procedure aProc;
var
    BigLocalArray1: packed array[0..HalfTooBig] of char;      { OK }
    BigLocalArray2: packed array[0..HalfTooBig] of char;      { Error }
begin
end;
begin
    New(BigArrayPtr1);           { Allocate memory                }
    New(BigArrayPtr2);
    BigArrayPtr1^[15000] := 'c'; { Can access large array        }
                                { with pointers                    }
    hTooBigArray := HugeHandle(NewHandle(sizeof(HugeArrayType))); { OK }
    hTooBigArray^[33000] := 'z';
end.

```

Bad actual parameter for formal VAR, procedural, or functional parameter.

See sections 7.3.2, 7.3.3, and 7.3.4 of Chapter 17. Example:

```

procedure VarProc (var i : integer);
begin
  end;
procedure Proc (i : integer);
begin
  end;
procedure CallP ( procedure P (i : integer));
begin
  end;
function Func (c : char) : char;
begin
  end;
procedure CallF ( function F (c : char) : char);
begin
  end;
begin
  VarProc(anInt + 4);           { Error: expressions can't be passed }
                                {      as VAR parameters           }
  VarProc(anInt);              { OK }
  CallP(Proc(3));              { Wrong: procedural parameters aren't }
                                {      called with parameters       }
  CallP(Proc);                 { Right }
  CallF(Func('c'));            { Wrong: functional parameters aren't }
                                {      called with parameters       }
  CallF(Func);                 { Right }
end;

```

Bad compiler directive

You probably forgot something in a compiler directive

```

program test;
begin
  {$SETC compiler_var = false} { Correct }
  {$IFC compiler_var}          { Correct }
    writeln('This line will not be compiled.');
```

{\$ENDC}	
{\$IFC}	{ ERROR: missing expression }
{\$SETC}	{ ERROR: missing assignment }

```

end.

```

Bad decimal-places expression in WRITE, WRITELN, or STRINGOF call.

The decimal-places expression is only valid for real output expressions and must evaluate to a positive integer expression. See 9.4.3.1, 9.4.3.2, 9.4.3.3, 9.4.3.4, 9.4.3.5, 9.4.3.6, and 9.4.3.7 of Chapter 17. Example:

```

writeln(anInt : 7 : 3);           { Error }
writeln(aString : 7 : 3);         { Error }
writeln(aReal : 7 : aString);     { Error }

```

```
aString := stringof(anInt : 7 : 3);    { Error }
aString := stringof(aString : 7 : 3); { Error }
```

Bad enumerated value.

During runtime, the procedure `read`, `readln`, or `readstring` expected an enumerated value for a specific type, but did not get one. See 9.4.1.5 of Chapter 17. Example:

```
readstring('paisley', aColor);    { Error }
readstring('plaid', aColor);      { Error }
readstring('violet', aColor);     { OK }
readstring('yeLLow', aColor);     { OK: case not important }
```

Bad expression type for READSTRING, or for READ or READLN from a text file.

Text files and strings can only contain characters. Values that have a string representation (except for packed arrays of characters) can be read from a text file or extracted from a string. For example, an integer can be represented by a string of digits and can be read from a text file. On the other hand, a record cannot be read from a text file, because it is a collection of data which does not have a simple string representation. You might think it should be represented in a particular string format, but there are no rules for it in Pascal.

Values that cannot be read from a text file, can be read from a **file of** that value's type. For example, an entire record can be read from a **file of** that record's type. If you really want to read a record from a text file or a string, then read each field of the record from the file or from string. See 9.4.1, 9.4.2, and 10.7.6 of Chapter 17. Example:

type

```
RecType = record
    Field_1 : integer;
    Field_2 : char;
end;
```

var

```
InString : string;    { Input String that readstring reads from }
TextFile : text;      { Input that readln reads from }
FileOfREC : file of RecType;
IntPtr : ^integer;
anArray : array[1..2] of char;
PAC : packed array[1..6] of char;
Rec : RecType;
```

begin

```
open(TextFile, 'text file');
open(FileOfREC, 'file of records');
readln(TextFile, aChar, aString, aColor, anInt, aLong, aReal); {OK}
readln(TextFile, IntPtr);    { Wrong }
readln(TextFile, aLong);     { OK, but probably not }
IntPtr := pointer(aLong);    { what you really want to do }
readln(TextFile, IntPtr^);   { Right }
readln(TextFile, anArray);   { Wrong }
readln(TextFile, anArray[1], anArray[2]); { Right }
```



```

readln(TextFile, Rec);           { Wrong: can't read a record      }
                                {   from a text file          }
readln(FileOfREC, Rec);         { Wrong: can't input a record  }
                                {   with a readln           }
read(FileOfREC, Rec);           { Right: OK to read a record   }
                                {   from a file of records      }
readln(TextFile, Rec.Field_1, Rec.Field_2); { Right: OK to read fields }
                                {   from TextFile             }
readstring(InString, aChar, aString, aColor, anInt, aLong, aReal);
                                {OK}
readstring(InString, anArray);   { Error }
readstring(InString, Rec);       { Error }
end;

```

Bad expression type for STRINGOF, or for WRITE or WRITELN to a text file.

Text files can only contain characters. Values that have a string representation can be written to a text file or copied to a string. For example, an integer can be represented by a string of digits and can be written to a text file. On the other hand, a record cannot be read from a text file because it is a collection of data that does not have a simple string representation. You might think it should be represented in a particular string format, but there are no rules for it in Pascal.

Values that cannot be written out to a text file can be written out to a **file of** that value's type. For example, a record can be written to a **file of** that record's type. If you really want to write a record to a text file, then write each field of the record to the text file or copy each field to a string. See 9.4.3, 9.4.4, and 10.7.5 of Chapter 17. Example:

```

type
  RecType = record
    Field_1 : integer;
    Field_2 : char;
  end;

var
  IntPtr : ^integer;
  anArray : array[1..2] of char;
  PAC : packed array[1..6] of char;
  aRec : RecType;
  FileOfREC : file of RecType;
  Result : string;
begin
  open(TextFile, 'text file');
  open(FileOfREC, 'file of records');
  writeln(TextFile, aBool, aChar, aString, aColor,
    anInt, aLong, aReal, PAC);      {OK}
  Result := stringof(aBool, aChar, aString, aColor,
    anInt, aLong, aReal, PAC);      {OK}
  writeln(TextFile, ord4(IntPtr));   { OK, probably not what      }
                                    {   you really want to do      }
  writeln(TextFile, IntPtr^);       { Right: outputs what IntPtr  }
                                    {   points to                    }
  writeln(TextFile, anArray);        { Wrong }
  writeln(TextFile, anArray[1], anArray[2]); { Right }

```

```

write(TextFile, aRec);           { Wrong: can't write a record }
                                { to a Text file }
writeln(FileOfREC, aRec);       { Wrong: can't output a record }
                                { with a writeln }
write(FileOfREC, aRec);         { Right: OK to write a record }
                                { to a file of records }
write(TextFile, aRec.Field_1, aRec.Field_2 ); { Also Right }
writeln(ColorSet);              { Wrong: can't write out a set directly }
{ Right: find elements in set and print them }
  for aColor := red to violet do
    if aColor in ColorSet then
      write(aColor);

Result := stringof(anArray);     { Error }
Result := stringof(TextFile);   { Error }
Result := stringof(aRec);        { Error }
end;
```

Bad field-width expression in WRITE, WRITELN, or STRINGOF call.

The field-width expression must evaluate to an integer expression greater than 0. Also, it is forbidden to use field widths when writing to non-text files. See 9.4.3.1, 9.4.3.2, 9.4.3.3, 9.4.3.4, 9.4.3.5, 9.4.3.6, and 9.4.3.7 of Chapter 17. Example:

```

anInt := 7;
writeln(anInt : 1 + anInt);      { OK }
writeln(anInt : aString);       { Error: field width expression }
                                { must be an integer }
writeln(anInt : aReal);         { Error: field width expression }
                                { must be an integer }
aString := stringof(anInt : aString); { Error }
aString := stringof(anInt : aReal);  { Error }
write(FileOfInt, anInt : 4);      { Wrong: can't specify field }
                                { width for non-text files }
write(FileOfInt, anInt);         { Right }
```

Bad Integer value.

When reading a value from a text file into a variable, if the text cannot be converted to a value for the variable type, then this runtime error occurs. See 9.4.1, 9.4.1.1, 9.4.1.2, 9.4.1.3, 9.4.1.4,, 9.4.1.5, and 9.4.2 of Chapter 17.

```

reset(TextFile, 'file containing ONLY letters');
readln(TextFile, anInt);         { Error: can't convert letters into integers }
readln(TextFile, aString);      { OK: any ASCII text can go into a string }
```

Bad object type in method declaration

The object type modifier in a method declaration is not an object name.

```

program test;
  uses
    ObjIntf;
  type
    T = integer;
  procedure T.x;           { ERROR: 'T' is not an object type }
  begin
  end;

begin
end.

```

Bad pointer in DISPOSE.

This runtime error most frequently occurs when you try to dispose an uninitialized or nil pointer or when you try to dispose the same memory twice. A less likely cause is that the heap is corrupted. This can happen if memory is used *after* it has been disposed. Use pointers with care. See 10.1.2 of Chapter 17.

Bad real value.

Your program attempted to read in a real value but found non-real characters instead.

```

program test;
  var
    x: real;
    textfile: text;

begin
  reset(textfile, 'reals');
  while not eof(textfile) do
    readln(textfile, x);           { ERROR when 'wxyz' is read }
  close(textfile);
end.

```

contents of file reals:

```

12.3
-12.3
wxyz
13

```

Bad set constructor.

See 5.3 of Chapter 17.

```

ColorSet := [[]];           { Wrong: can't have a set of an empty set }
ColorSet := [];             { Right: Empty Set }

```

Bad SIZEOF parameter.

The predefined function `sizeof` can only take variable or type names. (Note: the size of a string is not the same as its length, see 3.3 and 4.3.1 of Chapter 17). `sizeof` is described in 10.7.1 of Chapter 17. Example:

```

type
  LongType = longint;    { sizeof = 4 }
var
  size : integer;
  string19 : string[19];
  string20 : string[20];
  pr2 : packed record    { sizeof = 2 : packed chars are 1 byte apiece }
    c1, c2 : char;
  end;
  pr3 : packed record    { sizeof = 4 : extra byte to align even word }
    c1, c2, c3 : char;
  end;
  ur2 : record           { sizeof = 4 : unpacked chars take up 2 bytes apiece }
    c1, c2 : char;
  end;
  array10 : array[1..10] of integer;
procedure aProcedure;
begin
end;

function aFunction : boolean;
begin
  size := sizeof(anInt + aLong);      { Error: sizeof doesn't work }
                                     { with expressions }
  size := sizeof(aProcedure);         { Error: sizeof doesn't work }
                                     { with procedures }
  size := sizeof(aFunction);          { Error: sizeof doesn't work }
                                     { with functions }
  size := sizeof(aConstant);          { Error: sizeof doesn't work }
                                     { with constants }
  size := sizeof(LongType);           { OK: size = 4 }
  size := sizeof(string19);           { OK: size = 20 }
  size := sizeof(string20);           { OK: size = 22 }
                                     { 1 len + 20 data + 1 padbyte }
  size := sizeof(pr2);                { OK: size = 2 }
  size := sizeof(pr3);                { OK: size = 4 }
                                     { pad byte to align even word }
  size := sizeof(ur2);                { OK: size = 4 unpacked chars }
                                     { take up 2 bytes apiece }
  size := sizeof(array10);            { OK: size = 20 }
  size := sizeof(size);               { OK: size = 2 }
  size := sizeof(boolean);            { OK: size = 1 }

```

```

size := sizeof(char);      { OK: size = 2, unpacked chars    }
                             { take up two bytes          }
size := sizeof(integer);   { OK: size = 2                      }
size := sizeof(longint);   { OK: size = 4                      }
size := sizeof(real);      { OK: size = 4                      }
size := sizeof(double);    { OK: size = 8                      }
size := sizeof(extended);  { OK: size = 10                     }
end;
```

Boolean expression required.

Boolean expressions are required in control statements and with boolean operators. See 6.2.2.1, 6.2.3.1, 6.2.3.2, 5, and 5.1.3 of Chapter 17. Example:

```

aBool := 3 or 4;           { Error: the numbers 3 and 4 aren't booleans }
if 3 + 4 then              { Error: (3+4) doesn't evaluate to True or False }
  repeat
  until aReal;             { Error: aReal doesn't evaluate to True or False }
while green do             { Error: green doesn't evaluate to True or False }
;
```

Can't convert .o file

The MPW object file that THINK Pascal is trying to load contains some constructs that THINK Pascal can't convert to its format. For instance, THINK Pascal cannot use MPW object files that contain computed references or initialized data.

Can't EXIT procedure "*procedure_name*" from here

The procedure name in an Exit statement must enclose the statement.

```

program test;
  procedure proc;
  begin
    showtext;
    writeln('This will be written');
    exit(proc); { CORRECT }
    writeln('This will not be written');
  end;
begin
  proc;
  exit(proc);   { ERROR: "proc" does not enclose this statement }
end.
```

Can't find the file "*filename*". Would you like to look for it?

THINK Pascal remembers where a project's files are by pathname. When this dialog box appears, the file is no longer where the project thinks it is. Either you have deleted the file or you have moved it to a new location. If you click OK and find the file, the project will remember its new location and ask you:

Change "*old location*" to "*new location*" in all SUBSEQUENT (by build order) Project entries also?

THINK Pascal thinks if you have moved one file, so it is likely that you have moved other files too. As a convenience, you can change the references from the old location to the new location, but only by build order starting with files *after* the one you just changed.

Can't load a library with FAR relocations unless "Far Code" Is turned on In the "Set Project Type..." dialog.

When the "Far Code" option in the **Set Project Type...** dialog is turned off, THINK Pascal cannot load THINK Pascal libraries which were built with the "Far Code" option on or .o files generated with MPW's -model FAR option.

Can't read Keyboard or Modem.

This runtime error occurs when an expression in the Observe window tries to get input from the keyboard or modem. For example, the value cell of the Observe window will display this error when evaluating either of the following functions:

```
var
  s : string;
  f : text;
function ReadKeyBoard : integer;
begin
  readln(s);
  ReadKeyboard := length(s);
end;
function ReadModem : integer;
begin
  reset(f, 'MODEM:');
  readln(f, s);
  ReadModem := length(s);
close(f);
end;
```

Case constant Incompatible with tag-type or selector expression.

See 3.2.2 and 6.2.2.2 of Chapter 17. Example:

```
type
  BadVariantRecordType = record
    case Tag : ColorType of
      red : ( { OK: red is a ColorType }
        i : integer;
      );
      15 : ( { Error: 15 is not a ColorType }
        j : real;
      );
    end;
```

```

begin
  case aColor of
    red :           { OK: red is a ColorType           }
      writeln('OK');
    15 :           { Error: 15 is not a ColorType      }
      writeln('Error');
  end
end;

```

Case constant needed here.

See 6.2.2.2 of Chapter 17. Example:

```

var
  aVariantRecord : record
    case boolean of
      : ( { Wrong: missing case constant before the colon (:) }
        i : integer
      );
      true : ( { Right: true is a valid case constant }
              c : char
            );
    end;
begin
  case anInt of
    :
      writeln('Wrong: missing case constant before colon (:)');
    2 :
      writeln('Right: 2 is a valid case constant');
    3..6 :
      writeln('Right: Subranges in Case List allowed ');
    3, 4, 5, 6 :
      writeln('Right: 3,4,5,6 are valid case constants');
  end; { of case }
  case anInt of
    { Error: at least one case constant required }
  end;
  { of empty case }
end;

```

Change “old location ” to “new location ” in all SUBSEQUENT (by build order) Project entries also?”

See the error message: “Can't find the file *filename*. Would you like to look for it?”

Changing compile options will Reset your program. Continue anyway?

See “*action* will Reset your program. Continue anyway?”

Changing segmentation will Reset your program. Continue anyway?

See “*action* will Reset your program. Continue anyway?”

Changing segment attributes will Reset your program. Continue anyway?

See “*action* will Reset your program. Continue anyway?”

Changing the build order will Reset your program. Continue anyway?

See “*action* will Reset your program. Continue anyway?”

Changing the “Far Code” switch will require that all files be recompiled or reloaded. Do you want to do this?

When you changing the setting of the “Far Code” option in the **Set Project Type...** dialog, THINK Pascal must recompile all source files and reload all libraries. If you click Yes, THINK Pascal changes the setting of the “Far Code” option and removes all the object code from your project. If you click No, THINK Pascal returns you to the **Set Project Type...** dialog.

Code segment *n* (name) is too large (actual size *x* bytes, limit is *y* bytes)

The code segment is too large and you must move files out of it. See “Segmenting a Project” in Chapter 7, “Working with Projects,” for more information.

When you’re running a program under THINK Pascal, code segments must be 65,534 bytes or less. When you’re building a project, code segments must be 32,766 bytes or less, with one exception. If the “Far Code” option is one, the `%_MethTables` segment must be 65,534 bytes or less.

Colon (;) required on this line or above.

Colons are needed in labeled statements and case statements. See 3.2.2, 6.1.3, and 6.2.2.2 of Chapter 17. Example:

```
procedure Proc;
  label
    1, 2;
begin
  goto 1;
1      { Wrong: missing colon after the label 1 }
  writeln;
  goto 2;
2 :    { Right }
  writeln;
  end;
end;
```

Compiler variable is not defined

Before you can use a compiler variable in a `{ $IFC }` directive, you must define it.

```
program test;
{$SETC defined_var = false}      { Definition of 'defined_var'           }
{$IFC defined_var}               { Correct }
{$ENDC}
{$IFC undefined_var}             { ERROR: 'undefined_var' is not defined }
{$ENDC}
begin
end.
```


Component type of a file can't contain a file type.

See 3.2.4 of Chapter 17. Example:

```

type
  FileType = file of integer;
  Bad_NestedFileType = file of FileType; { Error: FILE OF a      }
                                     {      file type      }
  Bad_NestedFileRecordType = file of record
    F : FileType;           { Error: FILE OF a type containing file }
end;
  Bad_FileOfText = file of Text; { Error: Text is a file type      }

```

Constant expected. "symbol" isn't a constant.

At the erroneous statement, the compiler needs a constant to generate code for storage allocation, case statements, or variant records. See section 3 and 6.2.2.2 of Chapter 17. Example:

```

var
  NotAConstant : integer;
  aBadString : string[NotAConstant];           { Wrong }
  aGoodString : string[aConstant];             { Right }
  Bad : array[1..NotAConstant] of char;        { Wrong }
  Good : array[1..aConstant] of char;         { Right }

begin
  case anInt of
    NotAConstant : { Wrong: case tags must be constants }
      writeln('Equal to NotAConstant');
    aConstant : { Right: this case tags is constants }
      writeln('Equal to aConstant');
  otherwise
    writeln('Otherwise something else')
  end;
  { If you really want to have variable case tags, }
  { use a series of else-ifs }
  if anInt = NotAConstant then                 { Right }
    writeln('Equal to NotAConstant')
  else if anInt = aConstant then              { Right }
    writeln('Equal to aConstant')
  else
    writeln('Otherwise something else')        { Right }
  end;

```

Constant or expression (whose ordinal value is *Value*) is out of range.

This compile time error is detected only when the Range compiler directive is on. See Chapter 15. Example:

```

var
  a : array[1..10] of 1..6;
  r : real;
  SmallInt : 1..10;
  Cool : green..violet;

```

```

begin
  SmallInt := 11;           { Error: 11 is too big for SmallInt      }
  SmallInt := 0;           { Error: 0 is too small for SmallInt    }
  SmallInt := 5 + 6;       { Error: 11 is too big for SmallInt    }
  a[11] := 1;             { Error: 11 is out of range for Index  }
  a[10] := 7;             { Error: 7 is out of range for Element  }
  writeln(r : -3);        { Error: field width must be positive  }
  writeln(r : 4 : 0);     { Error: 2nd field width must be positive }
  Cool := red;            { Error: red is out of range for Cool.  }
                          {      ord(red) = 0      }
  SmallInt := 10;
  a[SmallInt + 11] := 1;   { This error is not caught at compile time }
end;

```

Constant or type can't be defined in terms of itself.

See 1.7 and 3 of Chapter 17. Example:

```

const
  aConst = -aConst;       { Error }
type
  TheType = TheType;      { Error }
  RecordPtrType = ^RecordType; { OK: type pointed to is
                              {   declared later   }

  RecordType = record
    BadNextRecord: ^RecordType; { Wrong: can't point to
                                {   defining type   }
    NextRecord: RecordPtrType;  { Right: must use previously
                                {   declared type   }
    NestedRecord : RecordType;  { Error: recursively nested
                                {   record           }

end;

```

Constant value is not numeric and must not have a sign.

See 1.7 and 5.1.2 of Chapter 17. Example:

```

const
  Dwarves = 7;
  MinusDwarves = -Dwarves; { OK }
  MinusRed = -red;        { Error: red is non-numeric }
  MinusChar = -aCharConstant; { Error: non-numeric }

```

Constant, expression, or packed type component was passed to a formal VAR parameter.

When a parameter is passed to a formal **var** parameter, the called subroutine can modify the actual parameter passed. Therefore, constants and expressions cannot be passed as **var** parameters, because they cannot be changed.

Components of packed structures cannot be passed as formal **var** parameters, because the actual data representation of a component in a packed structure may be different from the actual data representation in an unpacked structure. (For this Pascal implementation, only the representation

of packed and unpacked chars are different.) You can pass the entire packed structure as a **var** parameter.

Note: The predefined procedures `read`, `readln`, and `readstring` are special because they can be passed components of packed records. See 7.3.2 of Chapter 17.

Example:

```

type
    PackedArrayType = packed array[1..4] of integer;
var
    PackedRec : packed record
        i1, i2 : integer;
    end;
    Rec : record
        i1, i2 : integer;
    end;
    PackedAI : PackedArrayType;
    AI : array[1..4] of integer;
procedure NextInt (var i : integer);
begin
    i := i + 1;
end;
procedure ChangePAI (var thePAI : PackedArrayType );
begin
end;
begin
    NextInt (5);           { Error }
    NextInt (anInt+5);     { Error }
    NextInt (PackedRec.i2); { Wrong }
    NextInt (Rec.i2);      { Right: works with an unpacked }
                           { record component }
    NextInt (PackedAI[2]); { Wrong }
    NextInt (AI[2]);       { Right: works with an unpacked }
                           { array component }
    ChangePAI (PackedAI);  { OK: entire packed array can }
                           { be passed as a VAR parameter }
    readln (PackedAI[2]);  { OK: readln works on packed }
                           { structures components }
end;

```

Control variable of FOR statement is not of ordinal type.

An ordinal type is one of integer, longint, char, or enumerated. See 3.1.1.1 and 6.2.3.3 of Chapter 17. Example:

```

for aReal := 1 to 10 do           { Error: Reals are not ordinal }
    writeln('This will not work');
for anInt := 1 to 10 do           { OK }
    writeln('anInt =', anInt);
for aLong := $10000 to $10020 do { OK }
    writeln('aLong =', aLong);

```

```
for aChar := 'a' to 'z' do           { OK }
    writeln('The letters are ', aChar);
for aColor := violet downto red do   { OK }
    writeln('The colors are ', aColor);
```

CYCLE or LEAVE is not in a loop

You can use the Cycle and Leave procedures only in a **for**, **while**, or **repeat** loop.

```
program test;
    var
        x: integer;
begin
    for x := 1 to 10 do
        begin
            writeln('This line will be written once. ');
            leave;
            writeln('This line will not be written. ');
        end;
    for x := 1 to 5 do
        begin
            writeln(x : 1, ' : This line will be written five times. ');
            cycle;
            writeln('This line will not be written. ');
        end;
    cycle;           { ERROR }
    leave;           { ERROR }
end.
```

Data relocations not supported in Drivers and Code Resources.

When you use a library from another compiler in a desk accessory, device driver, or code resource, you need to be careful about the data initializations in the library. THINK Pascal does not honor initializations that set a variable to be a pointer to a function or a pointer to another variable (that is, initializations that require runtime relocations). For example, THINK Pascal would not honor this initialization in a THINK C library:

```
static ProcPtr myHook = &myFunction;
```

But THINK Pascal would honor this initialization:

```
char myString[] = "\psome string";
```

Divide by zero.

An integer-type division resulted in a division by zero. If the division was floating point, the "SANE Floating Point Error" would occur instead. Example:

```
anInt := anInt div 0;
```

Division by zero attempted.

Your program attempted to divide by zero.

```

program test;
  var
    x: integer;
begin
  x := 5 div 0;           { ERROR }
  x := 5 mod 0;          { ERROR }
end.

```

Drivers, and code resources of an owner type must have a resource ID between 0 and 63, Inclusive.

The Resource Manager requires these types of code segments have a resource ID between zero and 63:

- Those owned by a driver.
- Those of type WDEF, CDEF, MDEF, PDEF, or PACK

Duplicate case constant in this variant-part or CASE statement.

Case constants must be unique within a case statement or record-variant. See 3.2.2 and 6.2.2.2 of Chapter 17. Example:

```

type
  BadVariantRecordType = record
    case tag : integer of
      0 : (           { OK }
        i : integer;
      );
      0 : (           { Error: second 0 case }
        r : real;
      );
    end;
begin
  case anInt of
    2 :
      writeln('two');      { OK }
    2 :
      writeln('twice');    { Error: second 2 case }
  end
end;

```

Editing will Reset your program. Continue anyway?

See "*action* will Reset your program. Continue anyway?"

END needed to complete the above record declaration.

See 3.2.2 of Chapter 17. Example:

```

procedure P1;
  type
    GoodRecordType = record
      aField : integer;
    end;                                { Right }
    BadRecordType = record
      aField : integer;                { Wrong: Missing end for record declaration }
  begin
  end;

```

END. is required at the end of the file.

You must have **end.** at the end of your file. Make sure all you **begins** are matched by **ends**.

Execution halted.

Statements in the Instant window and expression evaluation in the Observe window usually finish instantly (see Chapter 9). If they don't, you can halt them by clicking on the Bug Spray Can. After halting the Instant window, THINK Pascal displays the message "Execution halted." After halting the Observe window, the message "Execution halted" appears in the value cell of the halted expression.

Expression can't be cast to the specified type.

When two variables are type-incompatible, they cannot be used interchangeably, even if they are the same size (as determined by `sizeof`). *Casting* bypasses this strong type-checking feature of Pascal. A cast changes a variable's type, but not its contents. For most casts, the two types must be of the same size. For example, on the MC68000 pointers and longints are type-incompatible, although both are 32 bits long. Type casting allows them to be used interchangeably.

All casting is machine dependent and potentially dangerous. Use with caution. See 5.4 of Chapter 17. Example:

```

procedure ExamplesOfCasts;
  type
    LargeAType = array[1..1000] of integer;
    BigRType = record
      aComponent : LargeAType;
    end;
    SmallRType = record
      aComponent : integer;
    end;
    Packed4Type = packed array[1..4] of char;
    TwoIntType = array[1..2] of integer;

```

```

var
    Big : BigRType;
    Small : SmallRType;
    Packed4 : Packed4Type;
    TwoInt : TwoIntType;
    Large : LargeAType;

begin
    aLong := longint(aReal);      { Legal, BUT just copies bits,      }
                                { doesn't convert to longint      }
    aLong := round(aReal);        { This converts Real type to      }
                                { equivalent Longint type.      }
    aReal := real(aLong);         { Legal, BUT just copies bits,      }
                                { doesn't convert to real        }
    aReal := aLong;               { This converts Longint type to      }
                                { equivalent Real type            }
    aLong := longint(Packed4);    { OK: aLong and Packed4 are same size }
    Big := BigRType(Small);       { Error: different sized records   }
    Large := LargeAType(Big);     { OK: Big and Large are the same size }
    Small := SmallRType(Big);     { Error: different sized records   }

    anInt := integer(Small);      { OK : anInt and Small are same size }
    Large := LargeAType(Packed4); { Error: different size array      }
    TwoInt := TwoIntType(Packed4); { OK: TwoInt and Packed4          }
                                { are the same size array        }

end;

```

Expression must be constant

THINK Pascal lets you declare constant expressions in your const declarations, but the expressions must be scalar — integer, boolean, char, or set.

```

program test;
var
    x: integer;
const
    a = 1;
    b = 2;
    c = a + b;           { Correct }
    d = ((a + b) * 2) div c; { Correct }
    e = 3.2 + 2.8;       { ERROR: constant expressions must be scalar }
    f = (a + b) / c;     { ERROR: (constant expressions must be scalar }
    g = a + b + x;       { ERROR: 'x' is a variable }

begin
end.

```

Expression too complex.

Try simplifying your expression. Use temporary variables for intermediate results if necessary.

Field name expected. "symbol" Isn't a field of this record.

See 4.3.2 of Chapter 17. Example:

```
var
  NotAField : integer;
  rec_1 : record
    field_1 : integer;
  end;
  REC_2 : record
    FIELD_2 : integer;
  end;
begin
  rec_1.NotAField := 0;           { Wrong: NotAField is not a field      }
                                { of any record                      }
  rec_1.field_1 := 1;           { Right                          }
  REC_2.field_1 := 0;           { Wrong: field_1 is a field of rec_1,  }
                                { not of REC_2                      }
  REC_2.FIELD_2 := 1;           { Right                          }
end;
```

File already open.

A file-variable can only be used once per open file. See 9.2.3 of Chapter 17. Example:

```
var
  NEWFILE, oldFile : Text;
begin
  rewrite(NEWFILE, 'a new file');
  open(NEWFILE, 'an old file'); { Wrong: file-variable NEWFILE      }
                                { is already in use                  }
  open(oldfile, 'an old file'); { Right: use another file variable }
end;
```

File is not open.

This run time error occurs when any input or output operation is done using a file-variable that has not been previously opened or that is already closed. See 9.2.4 of Chapter 17. Example:

```
var
  f : Text;
  TextFile : Text;
begin
  close(f);           { Error: file was not previously opened      }
  rewrite(f, 'new file');
  writeln(f, 'hello');
  close(f);           { OK                                          }
  close(f);           { Error: file closed in previous statement   }

  write(TextFile, anInt); { Error: TextFile was never opened      }
  read(TextFile, anInt); { Error                                  }
  page(TextFile);       { Error: page does an implicit write to a file }
  get(TextFile);        { Error: get does an implicit read of a file  }
end;
```


File is not opened for random access.

Only files opened with the procedure `open` are read-write. The procedure `seek` can be used only with read-write files. See 9.2.8 of Chapter 17. Example:

```
rewrite(FileOfInt, 'new file');
seek(FileOfInt, 0);           { Error }
close(FileOfInt );
reset(FileOfInt, 'old file');
seek(FileOfInt, 0);           { Error }
close(FileOfInt );
open(FileOfInt, 'old file');
seek(FileOfInt, 0);           { OK    }
close(FileOfInt );
```

File is not opened for reading.

Files opened with `rewrite` are write only. See 9.3.1 and 9.4.1 of Chapter 17. Example:

```
REWRITE(TextFile, 'WriteOnly File');
get(TextFile);                { Error }
read(TextFile, anInt);         { Error }
readln(TextFile, anInt);       { Error }
```

File is not opened for writing.

Files opened with `reset` are read-only, so you cannot write to them. See 9.2, 9.3.1, and 9.4.1 of Chapter 17.

```
RESET(FileOfText, 'old file');
put(FileOfText);              { Error }
writeln(FileOfText, 'Error: cannot writeln to ReadOnly file');
```

FOR loop control variable modified

Within a `for` loop, your program is not allowed to alter the value of the control variable. Use `Cycle` or `Leave` to alter control in the loop.

```
for anint := 1 to d do
  anint := 5;                { ERROR: can't modify control variable }
```

FOR statement control variable must be a local variable.

The Pascal language requires `for`-loop control variables to be declared at the same level that they are being used. See 6.2.3.3 of Chapter 17. Example:

```
program ForLoopControlVariables;
  var
    Outside : integer;          { can only be used in main program }
  procedure SubProcedure (aParam : integer);
  var
    local : integer;
  begin
    for integer := 1 to 10 do { Error: integer isn't a variable }
      ;
```

```

    for aParam := 1 to 10 do { Error: aParam isn't a local variable }
    ;
    for Outside := 1 to 10 do { Error: Outside isn't local variable }
    ;
    for local := 1 to 10 do { OK }
    ;
end;
begin
    for Outside := 1 to 10 do { OK: Outside is local to main program }
    ;
end.

```

Formal parameter type or result type should be a named type or STRING.

When a type is used in a parameter list or in a function result declaration, it must be a *named* type. It cannot be an *anonymous* type, although this is allowed in a variable declaration. See 7.2 and 7.3 of Chapter 17. Example:

```

type
    ArrayType = array[1..2] of char;
var
    Array_1 : array[0..2] of set of char; { OK: Array_1 is of
                                           { an anonymous type }
    Array_2 : ArrayType; { OK: Array_2 is of a
                           { named type }
function BadArray : array[1..2] of char; { Wrong }
begin
end;
function GoodArray : ArrayType; { Right }
begin
end;

type
    s25Type = string[25];
function BadString25 : string[25]; { Wrong }
begin
end;
function GoodString25 : s25Type; { Right }
begin
end;
function GoodString : string; { Also Works }
begin
end;

type
    IntegerPtrType = ^integer;
function BadResult : ^integer; { Wrong }
begin
end;
function GoodResult : IntegerPtrType; { Right }
begin
end;

```

```

{ Formal parameters must also have named types }
procedure BadParam (p : ^integer);          { Wrong: ^integer isn't named type }
begin
end;
procedure GoodParam (p : IntegerPtrType);    { Right }
begin
end;

```

Formal value parameter can't be a file type or a type which contains a file type.

For every open file, there must be only one copy of the file-variable. The operating system needs file-variables for file input/output. If a copy of the file-variable could be made and both were used for file I/O, then the file would be inconsistent. Therefore, passing a file-variable parameter by value (which makes a new copy) is forbidden. Passing a file-variable as a **var** parameter is permitted. See 7.3.1 of Chapter 17.

```

type
  IntFile = file of integer;
  RecWithFileType = record
    FileName : string;
    FileParameter : file of real;
  end;
  ArrayOfFileType = array[1..3] of file of integer;

procedure Bad1 (F : IntFile);          { Wrong: passing a filetype by value }
begin
end;
procedure Good1 (var F : IntFile); { Right: pass it as VAR parameter }
begin
end;
procedure Bad2 (R : RecWithFileType);    { Wrong: passing a filetype }
                                         {      by value           }
begin
end;
procedure Good2 (var R : RecWithFileType); { Right }
begin
end;
procedure Bad3 (A : ArrayOfFileType);    { Wrong: passing a filetype }
                                         {      by value           }
begin
end;
procedure Good3 (var A : ArrayOfFileType); { Right }
begin
end;

```

FORWARD or INTERFACE procedures or functions may not later be declared INLINE.

See 7.1.3 of Chapter 17. Example:

```
unit BadInLine;
interface
    function InterfaceFunc (x : char) : char;    { INTERFACE declaration    }
implementation
    function InterfaceFunc;
    inline                                     { Error: inline not permitted with INTERFACE }
    $4E71;                                     {      MC68000 NOP Instruction      }
    procedure ForwardProc (x : char);           { FORWARD declaration            }
    forward;
    procedure ForwardProc;
    inline                                     { Error: inline not permitted with FORWARD }
    $4E71;
end.
```

Function name required as an actual parameter to a formal functional parameter.

Functional parameters are a way of passing functions as parameters. See 7.3.4 of Chapter 17. Example:

```
function Func (i : integer) : integer;
begin
end;
procedure Proc (i : integer);
begin
end;
procedure CallF (function aFunc (i : integer) : integer);
begin
    anInt := aFunc(4);
end;
procedure RecallF (function aFuncParam (i : integer) : integer);
begin
    CallF(Func);           { OK: CallF is passed a function          }
    CallF(aFuncParam);     { OK: CallF is passed a functional parameter }
    CallF(4);              { Error: an integer is passed            }
    CallF(Proc);           { Error: a procedure is passed          }
end;
```

Global data exceeds *max* byte limit (*size* bytes).

The total size of *all* the global variables in your project (*size*) exceeds the limit for your project's type (*max*). These are the limits:

If your project is	its global data must be smaller than
an application or a single-segment desk accessory	32K
a multi-segment desk accessory	32K - the size of the jump table
a code resource	OK

If you need to create variables that require a lot of space, use the Memory Manager to allocate the space on the heap. Remember that THINK Pascal lets you declare array types that are bigger than 32K.

```

unit unit1;
interface
  type
    BigArrayType = packed array[0..16500] of char;
    BAPtr = ^BigArrayType;
    BAHandle = ^BAPtr;
  var
    BigArray1: BigArrayType; { one 16501-byte global variable }
implementation
end.

program test;
uses
  unit1;
var
  BigArray2: BigArrayType; { another 16501-byte global variable }
                           { Error here. }
  BigArray3 : BAHandle;

begin
  BigArray3 := BAHandle(NewHandle(sizeof(BigArrayType))); { OK }
  BigArray3^[16000] := 'x';
end.

```

GOTO out of Observe/Instant not allowed.

Because of the special nature of Observe and Instant windows, non-local **gotos**, which directly or indirectly exit a subroutine's current scope, are not allowed. For example, if you halted on the statement `writeln`; (see below) and then attempt to Observe the value of `Escape`, you will see this message in the value cell because of the `goto 9999` statement. The `goto 1` statement is a local `goto` and does not exit the `Escape`'s current scope. Furthermore, normal execution of `Escape` within your program will work without error.

```

program Escape;
label
  9999;
function Escape:integer;
label 1;
begin
  goto 1; { OK when Escape is called in Observe or Instant }
  anInt := 0;
1: goto 9999; { Error when Escape is called in Observe or Instant }
  Escape := 42; { This statement is never executed }
end;

```

```
begin
    writeln('stop here');
    anInt := Escape;      { OK to call Escape within the program }
9999:
    ;
end.
```

Illegal Instruction Exception

This runtime error indicates your program has run amok. For some reason, your program has jumped off into a random place in memory, or your code has been overwritten (e.g. by an uninitialized pointer), or the Memory Manager may be corrupted. Unless you can reproduce this behavior, it may be very hard to track down the problem and to fix it. You may have to use a low level debugger. (There are several bit patterns that are not legal MC68000 instructions. Your program was trying to execute one of these.)

InitGraf's parameter must be @thePort.

This runtime error occurs when the predefined Macintosh Toolbox call InitGraf is used with a parameter other than @thePort. This error is only detected while running with the THINK Pascal environment. See *Inside Macintosh* for more details. Example:

```
InitGraf(@anInt);      { Wrong }
InitGraf(@thePort);    { Right }
```

Insufficient stack space to invoke procedure or function.

This runtime error occurs when a subroutine, compiled with the Debug compile option, determines that there is not enough stack space remaining for it. It may be that too many nested subroutine calls have used too much stack space for local variables. However, your heap is probably unharmed. The solution may involve fixing an out of control recursive subroutine or increasing your application's stack size with the **Run Options...** command. Use LightsBug to examine the Subroutine Call Chain. See Chapters 13 and 15. Example:

```
procedure EndlessRecursion(i:longint);
begin
    EndlessRecursion(i+1);      { Eventually, stack space will run out here }
end;
```

Integer overflow.

The Overflow compile option inserts code that checks for intermediate arithmetic operation overflows. (See Chapter 15). This runtime error occurs when this check fails. Example:

```
anInt := 21000;
anInt := anInt + 20000;      { Error: the sum 41000 overflows anInt, }
                             { resulting in the erroneous value -24536 }
```

INTERFACE expected here.

See 8.3 of Chapter 17. Example:

```
unit MissingKeywordINTERFACE;
    { Error: missing interface keyword here }
implementation
end.
```

Invalid .o file

The MPW object file THINK Pascal is trying to read is not in a format THINK Pascal understands. It may have been produced by the wrong version of MPW.

Invalid formal parameter list.

The parameter list in a function or procedure declaration is bad. See 7.3 of Chapter 17. Example:

```
procedure BadProc_1 (1, 2 : integer;      { Wrong : bad parameter names      }
    ExtraSemiColon : integer;            { Wrong : extra (;) after last      }
    );                                    { parameter                          }

begin
end;

procedure GoodProc_1 (one, two : integer;  { Right }
    LastParameter : integer               { Right }
    );

begin
end;

procedure BadProc_2 (x, y :
    INTEGER var z : char );              { Wrong: missing ; after          }
begin                                  { INTEGER                          }
end;

procedure GoodProc_2 (x, y : INTEGER;      { Right }
    var z : char);

begin
end;

    { Wrong: aProc is declared as a procedure but has a result type CHAR }
procedure BadCallF ( procedure aProc (i : integer) : CHAR);
begin
    aProc(5);
end;

    { Right: aProc is correctly declared as a procedural parameter      }
procedure GoodCallP ( procedure aProc (i : integer));
begin
    aProc(5);
end;

    { Also Right: aFunc is correctly declared as a functional parameter  }
procedure GoodCallF (function aFunc (i : integer) : char);
begin
    aChar := aFunc(5);
end;
```

Invalid INLINE constant.

Only 16-bit integer values are allowed for inline constants. See 7.1.3 of Chapter 17. Example:

```
{ Inline routine to Call a procedure through a pointer }
procedure Wrong_JSR_To(PtrToProcedure: Ptr);
inline           { Wrong: constants must be word sized }
    $205F4E90;
procedure Right_JSR_To(PtrToProcedure: Ptr);
inline           { Right: each constant is word sized }
    $205F,        { MOVEA.L    (SP)+,A0 }
    $4E90;        { JSR        (A0)    }
procedure Proc;
begin
end;
procedure Call;
begin
    Right_JSR_To(@Proc);
end;
```

Invalid list of field names.

Field names must be legal Pascal identifiers. See 1.2 and 3.2.2 of Chapter 17. Example:

```
type
    InvalidFieldName = record
        Legal, 5, AnotherName : integer; { Error: 5 is bad field name }
    end;
```

Invalid list of variable names.

Variable names must be legal Pascal identifiers. See 4.1 of Chapter 17. Example:

```
var
    a, 1, 2, c : integer; { Error: 1 and 2 aren't legal variable names }
```

Invalid OBJECT typecast.

You can cast an object only to an ancestor object.

```
program test;
uses
    ObjIntf;
type
    AnObj = object (TObject)
    end;
    AnotherObj = object (TObject)
    end;
    ExpandedObj = object (TObject)
        x, y, z: integer;
        function func: boolean;
    end;
```



```

function ExpandedObj.func: boolean;
begin
    func := true
end;

var
    a, w: AnObj;
    b, x: AnotherObj;
    c, y: ExpandedObj;
    d, z: TObj;

begin
    new(w);
    new(x);
    new(y);
    new(z);
    d := TObj(x);      { Correct }
    d := TObj(y);      { Correct }
    d := TObj(z);      { Correct, but z is already a TObj }
    c := ExpandedObj(x); { ERROR if Range checking is on }
    c := ExpandedObj(z); { ERROR if Range checking is on }
    b := AnotherObj(w); { ERROR if Range checking is on }
    b := AnotherObj(y); { ERROR if Range checking is on }
    b := AnotherObj(z); { ERROR if Range checking is on }
end.

```

Invalid PROGRAM parameter list.

See 8.2 of Chapter 17.

Invalid redeclaration of method

You've tried to declare a field of an object as a method.

```

program test;
    uses
        ObjIntf;
    type
        TestObj = object (TObj)
            x: integer;
        end;
    procedure TestObj.x;      { ERROR: 'x' is a field, not a method }
    begin
    end;
begin
end.

```

Invalid result type in function declaration.

See 7.2 of Chapter 17. Example:

```
function BadF1;                                { Wrong: missing result type }
begin
end;
function GoodF1 : char;                        { Right }
begin
end;
function BadF2 (c : char)char;                { Wrong: missing colon in result type }
begin
end;

function GoodF2 (c : char) : char; { Right }
begin
end;
function GoodF3 : char;                        { OK: forward definition of GoodF3 }
forward;
function GoodF3;                                { OK: definition of forward function may leave off }
begin                                         { its result type }
end;
{ Wrong: missing result type for formal functional parameter "aFunc" }
procedure BadCall (function aFunc (c : char));
begin
end;
{ Right }
procedure GoodCall (function aFunc (c : char) : CHAR);
begin
end;
```

Invalid variable, field, or formal parameter list. A colon (:) might be missing.

When a variable name is declared, a corresponding type must be provided. See sections 3 and 4 of Chapter 17. Example:

```
var
  VariableWithoutAType;                        { Wrong: missing type }
  Variable : integer;                          { Right }
  aRecord : record
    FieldWithoutAType;                        { Wrong: missing type }
    GoodField : integer;                     { Right }
  end;
procedure P (aParamWithoutAType;              { Wrong: missing type }
             aGoodParam : integer);          { Right }
begin
end;
```

Invalid variant declaration.

See 3.2.2 of Chapter 17. Example:

```

type
  BadVariantRecordType = record
    case i : integer of
      1 : (
        a, b : char; { OK }
      );
      2 : (
        { Right: OK to have empty variant }
      );
      3 :
        { Wrong: Missing () for empty variant }
    end;

```

Jump Table exceeds 32K limit (size bytes)

See "The jump table is too large."

Label has already been declared.

See 2.2.4 of Chapter 17. Example:

```

program BE_70;
  label
    6;
  procedure SubProc;
  label
    6; { OK: This label is in a different scope }
  procedure SubSubProc;
    label
      6; { OK: This label is in a different scope }
      label
        6; { Error: duplicate label declaration in SubSubProc }

  begin
    6:
      ;
  end; { SubSubProc }
  begin
    6:
      ;
  end; { SubProc }
begin
  6:
    ;
end.

```

Label has already been used to label a statement.

See 2.2.4 of Chapter 17. Example:

```

procedure Proc;
  label
    13;
begin
  13 :
    writeln('OK');
  13 :
    writeln('Error: another statement with a label 13 at this level');
end;

```

Label has not been used to label any statement.

Labels must label a statement at the level where the label is declared. See 2.1 of Chapter 17.

Example:

```

program UsingLabels;
  procedure Zero;
    label
      0; { Error: label is not used in procedure which declared it }
    begin
      writeln;
    end;
begin
end.

```

Label hasn't been declared at this level.

Statements can only be labeled with labels declared at the same level. See 2.1 and 6 of Chapter 17.

Example:

```

program UsingLabels;
  label
    1;
  procedure ZeroAndOne;
    label
      0, 1;

  begin
    0 :
      ;
    1 :
      ;
  end; { ZeroAndOne }

```

```

begin
13 :      { Error: 13 was never declared }
      writeln('Statement Labeled with 13');
0 :      { Wrong: 0 was not at this level }
      writeln('Statement Labeled with 0');
1 :      { Right: irrelevant if 1 was declared in ZeroAndOne }
      writeln('Statement Labeled with 1');
end.

```

Label was not declared at the same level as this statement.

Statements can only be labeled with labels declared at the same level. See 2.1 of Chapter 17.

Example:

```

program UsingLabels;
  label
    1,2;      { declares labels for use in main program }
  procedure SubProc;
  begin
1 : { Wrong: label 1 hasn't been declared at this level }
   goto 2; { OK to GOTO a label at an outer scope }
   end; { SubProc }
begin
1 :      { Right: label 1 must be used at this level }
   writeln('statement labeled with 1');
2 :      { Right: label 2 must be used at this level }
   writeln('statement labeled with 2');
end.

```

Library must contain at least one file.

When you use the **Build Library...** command, the project must contain at least one file. See Chapter 10.

Library must contain only one code segment.

The project you are trying to build into a library contains more than one segment. Before you can build it, you must either (1) combine the segments into a single segment or (2) break up your project into multiple single-segment projects and build a library for each project. To see how your project is segmented, click on the view control in the upper right corner of the project window. See Chapters 7 and 10.

Library must not contain a main program.

Libraries are not allowed to have a main program.

Link Failed: multiply-defined: "*symbol*"

- Have you included two different versions of the same library into your project?
- Does a library contain an extraneous `Runtime.lib` or `Interface.lib`?

Link Failed: undefined: "*symbol*"

- Have you added all necessary libraries to your project?
- Have you declared a subroutine external and never defined it in one of your libraries?

Lower boundary of subrange is greater than upper boundary.

For types, the lower bound must be less than the upper bound. For set constructors, the lower bound can be greater than the upper bound, in which case the set is empty. See 3.1.1.3 and 5.3 of Chapter 17. Example:

```

procedure Proc;
  var
    BadColor : blue..red;      { Error }
    BadSubrange : 3..1;        { Error }
    aSet : set of ColorType;
begin
  aSet := [blue..red];          { OK: aSet is empty }
end;

```

Macintosh System Error: -nnn

One of the more obscure Macintosh errors has occurred. See *Inside Macintosh*.

Macsbug/TMON not Installed

Your program tried to execute a Debugger or DebugStr procedure when a low level debugger wasn't installed.

```

program test;
begin
  Debugger;                      { ERROR if no low-level debugger is installed }
  DebugStr('Break in TEST'); { ERROR: same reason }
end.

```

MC68881: divide by zero

MC68881: Inexact

MC68881: not a number

MC68881: operand error

MC68881: overflow

MC68881: underflow

You'll get these errors only when you have the 68881/68882 option on (see Chapter 15) and you've enabled halts for certain conditions. (See the SetHalt routine in the *Apple Numerics Manual*.)

Method not found.

The method you tried to invoke doesn't exist for the object you're using.

```

program test;
  uses
    ObjIntf;
  type
    AnObj = object (TObject)
      procedure proc;
    end;
  procedure AnObj.proc;
  begin
  end;

  var
    T: TObject;
    A: AnObj;
begin
  new(T);
  a := AnObj(T);
  a.proc;                      { ERROR }
end.

```

Method not inherited by this object

You used the **inherited** keyword in front of a method name, but the object you're referencing doesn't inherit that method.

```

program test;
  uses
    ObjIntf;
  type
    AnObj = object (TObject)
      end;
    AnotherObj = object (AnObj)
      function func: boolean;
    end;
  function AnotherObj.func: boolean;
  begin
    func := true;
  end;

  var
    x: AnotherObj;
    b: boolean;
begin
  new(x);
  b := x.func;                  { Correct }
  with x do
    b := func;                  { Correct }
  with x do
    b := inherited func;       { ERROR }
end.

```

Missing or Invalid unit name in this USES.

See 8.1 of Chapter 17. Example:

```
program WithBadUSES;
  uses
    4;          { Error: invalid unit name }
begin
end.
```

NIL dereference.

The Range compile option inserts code into your program which checks for nil pointer dereferences (see Chapter 15). This runtime error occurs when this check has failed. Sometimes the actual cause of this error is long gone from the scene, but often the culprit is close at hand. You may be able to find the problem using Observe and LightsBug.

Note: If the pointer is not initialized before assignment, then you probably will not get this error. Instead, some piece of memory may be corrupted. Later, an Odd Address or an Illegal Instruction may occur.

Example:

```
{ Assumes the Range compile option is On }
var
  IntPtr : ^integer;
begin
  IntPtr := nil;          { It's important to initialize Pointer Variables! }
  IntPtr^ := 13;          { Wrong: IntPtr is a NIL pointer }
  IntPtr := @anInt;       { This makes sure that IntPtr points to something }
  new(IntPtr);            { Also make sure that IntPtr points to something }
  IntPtr^ := 13;          { Right: IntPtr now points somewhere }
end;
```

No context

This message appears in the value cells of the Observe window when your program is not running, paused, or halted. Because there is no program context, expressions (even constants) cannot be evaluated.

No windows are available for opening your file.

You can open up to eight edit windows to edit files. This number does not include the Instant and Observe windows. If you get this message, you'll need to close some files. Remember that THINK Pascal doesn't close files when you click on a window's close box. You need to use the **Close** command in the **File** menu to close a file. You can also hold down the Command key as you click in the window's close box.

Not previously declared as a method

You're trying to define a method that you didn't include in the object's declaration.

```

unit unit1;
interface
  uses
    ObjIntf;
  type
    TestObj = object(TObject)      { Declare object type "TestObj"      }
      function func: boolean;      { Declare method "TestObj.func"    }
    end;
implementation
  function TestObj.func: boolean;   { Implementation of method        }
  { "TestObj.func"                  }
  begin
    func := true
  end;

  procedure TestObj.proc;           { ERROR: this method wasn't declared }
  {                               { in the declaration of TestObj }
  begin
  end;
end.

```

Objects and Methods may not be declared in nested scopes

Objects and methods must be defined in the outermost scope. Procedures and functions cannot define objects or methods.

```

program test;
  uses
    ObjIntf;
  procedure proc;
    type
      TestObj = object(TObject) { ERROR: trying to declare object }
      and;                       { within scope of "proc" }
    begin
    end;
begin
end.

```

```

unit unit1;
interface
  uses
    ObjIntf;
  procedure proc;
  type
    TestObj = object(TObject)      { Object "TestObj"      }
      function func: boolean;      { Method "TestObj.func" }
    end;

implementation
  procedure proc;
    function TestObj.func: boolean; { ERROR: can't declare method  }
    begin                           { within scope of "proc"  }
    end;
  begin
  end;
end.

```

Operand type Incompatibility.

Some operations aren't allowed, though it seems they ought to be. For example, record assignments are legal because it is easy to do a blind byte copy. However, record comparisons are forbidden, because a blind byte comparison will not work, and anything else is too hard. In particular, two *equivalent* structures can have *different* bit patterns because of byte padding or different garbage in inactive variants. The solution is to write your own comparison functions for records and arrays. (Note: it is permitted to compare packed arrays of characters.)

The following examples show common errors and methods to break the rules. Use the methods at your own risk. See 3.5 and 5 of Chapter 17.

```

var
  AC1, AC2 : array[1..10] of char;
  PAC1, PAC2 : packed array[1..10] of char;
  PAI1, PAI2 : packed array[1..10] of integer;
  Rec1, Rec2 : packed record
    IntField1 : integer;
    CharField : char; { byte of pad between IntField1 and IntField2 }
    IntField2 : integer;
  end;
  ptr1, ptr2 : ptr;
  string1, string2, string3 : string;

begin
  string1 := string2 + string3;      { Wrong }
  string1 := concat(string2, string3); { Right: see 10.5.3 of Chapter 17 }

  aBool := (aReal <> anInt);         { OK to compare real with integer }
  aBool := (aString <> AC1);          { Wrong: can't compare strings with }
                                      { UNpacked char array }
  aBool := (aString <> PAC1);         { Right: OK to compare strings with }
                                      { packed char array }
  anInt := 3.5 mod 4.5;              { Wrong: MOD can't take real arguments }

```

```

anInt := 3 mod 4;           { Right }
aColor := red + blue;       { Wrong: can't use + on enumerated types }
aColor := ColorType(ord(red)+ord(blue)); { Right: BUT is this }
                                   { really meaningful? }
aBool := (ptr1 > ptr2);     { Wrong: ptrs have no ordering }
aBool := (longint(ptr1)>longint(ptr2)); { Right: BUT is this }
                                   { really meaningful? }
aBool := (ptr1 <> ptr2);    { OK: can only use = and <> on ptrs }
AC1 := AC2;                 { OK: it's legal to assign unpacked arrays }
aBool := (AC1 <> AC2);      { Wrong: can't use =, <>, etc. }
                                   { on unpacked char arrays }
aBool := (PAC1 <> PAC2);    { Right: OK to compare packed CHAR array }
aBool := (PAC1 > AC1);      { Error: can't compare packed and }
                                   { unpacked CHAR array }
aBool := (PAI1 <> PAI2);    { Error: can't compare packed arrays }
                                   { of NonChars }
Rec1 := Rec2;               { OK to assign records }
aBool := (Rec1 = Rec2);     { Error: can't compare records this way }
end;

```

OVERRIDE expected

If you want to redefine a method that belongs to an ancestor object, you need to use the override directive.

```

program test;
  uses
    ObjIntf;
  type
    AnObj = object(TObject)
      function func: boolean; { OK method "AnObj.func" }
    end;
    AnotherObj = object(AnObj)
      function func: boolean; { Error: need "override" if re- }
                                { defining method }
    end;
  function AnObj.func: boolean;
  begin
    func := true
  end;
  function AnotherObj.func: boolean;
  begin
    func := false
  end;
begin
end.

```

Parameter list doesn't match previous declaration

THINK Pascal lets you repeat the parameter list for procedures and functions defined in the interface section or for routines declared forward. The parameter lists must match exactly. Note that parameters of type **string** do not match. You must create a type for them.

```
unit unit1;
interface
    procedure proc (x, y, z: integer);
implementation
    procedure proc (x, y, z: longint); { ERROR: type should be "integer" }
    begin
    end;
end.
```

Period (.) required following the last END of the file.

See 8.1 of Chapter 17. Example:

```
program MissingPeriod;
begin
end          { Error: Missing period }
```

Predefined and inline procedure names can't be used as procedural parameters.

Predefined, Toolbox, or inline calls (which aren't genuine subroutines) cannot be passed as procedural or functional parameters. However, you can write a wrapper routine (which wraps itself around the desired subroutine) that you can pass as a procedural or functional parameter. See 7.3.3 of Chapter 17. Example:

```
program ProceduralAndFunctionalParameters;
    procedure CallF (function F (e : Extended) : integer);
    begin
    end;
    procedure CallP ( procedure P);
    begin
    end;
    procedure NOP;
    inline
    $4e71;                                { M68000 NOP instruction }
    procedure NOPWrap;
    begin
    NOP;
    end;
    function TruncWrap (e : Extended) : integer;
    begin
    TruncWrap := Trunc(e);                { Predefined function }
    end;
    procedure OpenRgnWrap;
    begin
    OpenRgn;
    end;
begin
    CallF(Trunc);                          { Wrong: Trunc is predefined function }
    CallF(TruncWrap);                      { Right: Trunc is called from TruncWrap }
```

```

CallP (NOP);           { Wrong: NOP is an inline procedure      }
CallP (NOPWrap);       { Right: NOP is called from NOPWrap          }
CallP (OpenRgn);       { Wrong: OpenRgn is a Toolbox procedure   }
CallP (OpenRgnWrap);   { Right: OpenRgn is called from OpenRgnWrap  }
end.

```

Printer port in use by AppleTalk.

You will have to use the modem port instead. If you really want to use AppleTalk, see *Inside Macintosh* and Chapter 11.

Procedure name required as an actual parameter to a formal procedural parameter.

See 7.3.3 of Chapter 17. Example:

```

function Func (i : integer) : integer;
begin
end;

procedure Proc (i : integer);
begin
end;

procedure CallP ( procedure aProc (i : integer));
begin
    aProc(4);
end;

procedure RecallP ( procedure aProcParam (i : integer));
begin
    CallP(Proc);           { OK: CallP is passed a procedure      }
    CallP(aProcParam);     { OK: CallP is passed a procedural parameter name }
    CallP(4);              { Error: an integer is passed          }
    CallP(Func);           { Error: a function is passed          }
end;

```

Procedure or function has a FORWARD declaration but was never defined.

The subroutine declarations that use the forward directive allow you to defer the subroutine's definition. However, the definition must appear later in the file. See 7.1.1 and 8.3 of Chapter 17. Example:

```

unit aUnit;
interface
implementation
    procedure aForward;    { Error: never defined later in this unit. }
    forward;
end.    { end of unit }

```

PROGRAM or UNIT is missing from the beginning of this file.

You must have the keyword **program** or **unit** at the beginning of your file.

Project has no main program.

In order for your program to run, a file which contains a main program needs to be added to your project. The file should contain the **program** keyword.

Quitting will Reset your program. Continue anyway?

See “*action* will Reset your program. Continue anyway?”

READLN and WRITELN can only be used with text files.

The predefined procedures `readln` and `writeln` only work with text files. Use `read`, `write`, `put` or `get` for all other types of files. See 9.3 and 9.4 of Chapter 17. Example:

```

procedure Proc;
  var
    FileOfChar : file of char;           { not the same thing as text }
    PackedFileOfChar : packed file of char; { not the same thing as text }
    FileOfInt : file of integer;         { not at all the same as text }
begin
  open(TextFile, 'Text File');
  writeln(TextFile, 4);                   { OK }
  readln(TextFile, anInt);                 { OK }
  open(PackedFileOfChar, 'packed file of CHAR');
  writeln(PackedFileOfChar, 'a')           { Wrong }
  write(PackedFileOfChar, 'a');            { Right }
  readln(PackedFileOfChar, aChar);         { Wrong }
  read(PackedFileOfChar, aChar);           { Right }
  open(FileOfChar, 'file of CHAR');
  writeln(FileOfChar, 'a')                 { Wrong }
  write(FileOfChar, 'a');                  { Right }
  open(FileOfInt, 'file of INTEGERS');
  writeln(FileOfInt, 4);                   { Wrong }
  write(FileOfInt, 4);                     { Right }
end;

```

Record contains no fields.

Records must contain at least one field. Example:

```

type
  RecordType = record
    end;           { Error: no fields }

```

Record name expected. “*symbol*” Isn't a record.

Symbol is not properly used in a record reference. See 6.2.4 of Chapter 17.

```

type
  RecordType = record
    aField : integer;
  end;
  RecordPtrType = ^RecordType;
var
  aRecord : RecordType;
  aRecordPtr : RecordPtrType;
function aFunc : RecordPtrType;
begin
end;

```

```

begin
  anInt.aField := 3;           { Wrong: anInt is not a record      }
  aRecord.aField := 3;        { Right }
  with anInt do                { Wrong: anInt is not a record      }
    aField := 3;
  with aRecord do              { Right }
    aField := 3;
  new(aRecordPtr);
  aRecordPtr.aField:=3;        { Wrong: need to dereference pointer  }
  aRecordPtr^.aField:=3;      { Right: the ^ makes all the difference }
end.

```

Replacing will Reset your program. Continue anyway?

See “*action* will Reset your program. Continue anyway?”

Replacing a file in the project will Reset your program. Continue anyway?

See “*action* will Reset your program. Continue anyway?”

Result can't be assigned to the function named “*symbol*” outside of its declaration.

See 7.2 of Chapter 17. Example:

```

program AssigningToFunctions;
  type
    IntPtrType = ^integer;
  var
    IntPtr : IntPtrType;
  function aFunction : integer;
  begin
    aFunction := 42; { Right }
  end;
begin
  aFunction := 42;    { Error: outside of function declaration }
end.

```

Result type doesn't match previous declaration

You can repeat the result type of a function you previously declared forward, but the result types must match exactly. Example:

```

function GregHowe : char;
forward;
function GregHowe : integer;      { ERROR: Result type must be char      }
function GregHowe : char;         { OK: Result type matches          }
function GregHowe;                { OK: OK to leave out result type    }
                                   { if previously declared forward      }

```

```

unit unit1;
interface
    function PaulVetri: integer;
implementation
    function PaulVetri: longint;    { ERROR: type should be "integer"    }
    begin
        PaulVetri:= 13;
    end;
end.

```

SANE Floating Point Error

A SANE (Standard Apple Numerics Environment) call has resulted in an error. Note: a Floating point division by zero results in this error, while an integer-type division by zero results in the Divide by zero error. Example:

```

var
    r: real;
    i: integer;
begin
    r := 10.2;
    for i := 1 to 11000 do
        r := r * 10;
    end;
end;

```

SEEK and FILEPOS are only allowed for disk files.

You cannot use the SEEK and FILEPOS routines on devices like the modem port.

```

program test;
var
    textfile: text;
    position: longint;
begin
    open(textfile, 'old file');
    seek(textfile, 0);                { Correct }
    position := filepos(textfile);    { Correct }
    close(textfile);
    open(textfile, 'MODEM:');
    position := filepos(textfile);    { ERROR: textfile is the MODEM: }
    seek(textfile, 0);                { ERROR: textfile is the MODEM: }
end.

```

SEEK of a negative component number not allowed.

See 9.2.8 of Chapter 17. Example:

```

open(FileOfInt, 'old file');
seek(FileOfInt, -1);                { Error }
seek(FileOfInt, 0);                 { OK: seeks to first component in file }
seek(FileOfInt,maxlongint);         { OK: seeks to last component in file }

```


Segment Loader error (not enough memory).

When a subroutine in an unloaded segment is called, the segment loader automatically loads it into the heap. This error will occur when there is not enough memory to load the segment. You can either increase your Heap size with the **Run Options Command...**, or use **UnloadSeg** to unload unneeded segments. See *Inside Macintosh*. If you're running with MultiFinder, increase the size of THINK Pascal's partition.

Selector expression of CASE statement Isn't of ordinal type.

See 3.1.1 and 6.2.2.2 of Chapter 17. Example:

```
begin
  case aReal of      { Error: aReal is not ordinal      }
    3.14159 :
      ;
  end
  case ColorSet of { Error: ColorSet is not ordinal }
    [red] :
      ;
  end
  case anInt of      { OK: anInt is ordinal      }
    0 :
      ;
  end;

  case aChar of      { OK: aChar is ordinal      }
    'c' :
      ;
  end;

  case aColor of      { OK: aColor is ordinal      }
    red :
      ;
  end;

  case aLong of      { OK: aLong is ordinal      }
    $40000 :
      ;
  end;
end;
```

Semicolon (;) or END expected after the previous statement.

See 6.2.1 of Chapter 17. Example:

```
begin
  writeln('Error: Missing SemiColon at end of this line -->')
  writeln('OK: SemiColon not needed at end of this line')
end;
```

Semicolon (;) or UNTIL expected after the previous statement.

See 6.2.3.1 of Chapter 17. Example:

```
begin
  repeat
    writeln('OK');
  until false;
  repeat
    writeln('Error: missing semicolon on this line -->')
    writeln
  until false;
  repeat
    writeln('Error: Missing UNTIL');
end;
```

Semicolon (;) required on this line or above.

See 2.1, 1.7, 3, 4.1, 7.1, 7.2 of Chapter 17.

```
label
  0                                { Error: missing semicolon at end of line }
const
  BadConstant = 0                 { Error: missing semicolon at end of line }
type
  BadType = char                  { Error: missing semicolon at end of line }
var
  BadVar : integer                { Error: missing semicolon at end of line }

procedure BadProc : CHAR;        { Error: procs do not return values }
begin
end;
function BadFunc : char          { Error: missing semicolon at end of line }
begin
end;
```

Set constant out of range

Set value out of range.

When assigning a value to a set, the range of the resulting set expression must be within the range of variable's set type. Example:

```
var
  DigitSet : set of 0..9;
  WarmSet : set of red..yellow;
begin
  DigitSet := [-1, 1];             { Error: -1 not in range 0..9 }
  DigitSet := [5..10];            { Error: 10 not in range 0..9 }
  DigitSet := [];                 { OK }
  DigitSet := [0..9];             { OK }
  DigitSet := [1, 2, 3, 5, 7];    { OK }
  if 1000 in DigitSet then        { OK: even if 1000 isn't in range 0..9 }
  ;
```

```

DigitSet := DigitSet - [10..15];    { OK: expression is in range    }
WarmSet := [blue];                  { Error: blue not in range red..yellow }
WarmSet := [];                      { OK }
WarmSet := [red..yellow];           { OK }
end;

```

Set elements must be Integer, char or enumerated.

Set element types are limited. However, in THINK Pascal the ranges are not as limited as in other Pascal implementations. See 3.2.2 of Chapter 17. Example:

```

type
  ArrayType = array[1..3] of integer;
  RecordType = record
    aField : integer;
  end;
  SetType = set of char;
var
  SetOfArray : set of ArrayType;    { Error }
  SetOfLongint : set of longint;    { Error }
  SetOfRecord : set of RecordType;  { Error }
  SetOfReal : set of real;          { Error }
  SetOfSet : set of SetType;        { Error }
  SetOfColor : set of ColorType;    { OK }
  SetOfChar : set of char;          { OK }
  SetOfSmallInts : set of 1..10;    { OK }
  SetOfIntegers : set of integer;   { OK }

```

Stack has moved into application heap.

The stack has overflowed into the heap; the heap is probably damaged. This error, which may be more familiar as System Error ID 28, is not always detected. (See *Inside Macintosh*.) In hindsight, you should have compiled with the Debug option, which probably would have caught this error safely and in time. (See Chapters 7 and 15.)

String constant must have length 1 to be compatible with Char.

See 3.5.3 of Chapter 17.

```

const
  aStringConstant = 'Hi';
  NullStringConstant = '';
  CouldBeEither = '?';
begin
  aChar := '';                { Error }
  aChar := NullStringConstant; { Error }
  aChar := 'Hi';              { Error }
  aChar := 'C';               { OK }
  aChar := CouldBeEither;     { OK }
end;

```

String constant too large or too small for destination

When assigning a string constant to a string variable, the string constant must fit. When assigning a string constant to a packed array of characters, the length of the string constant must be the same as the number of characters in the array. Example:

```
var
  s5: string[5];
  PAC5 : packed array[1..5] of char;
begin
  s5 := 'twelve chars';      { Error: length of string constant > 5  }
  s5 := 'five!';             { OK }
  PAC5 := 'twelve chars';    { Error: length of string constant <> 5  }
  PAC5 := 'four';            { Error: length of string constant <> 5  }
  PAC5 := 'five!';          { OK }
end;
```

String range error.

Do not confuse a string's *size* with its *length*. This runtime error is caused by accessing a character element of a string with an index outside the range 1..length(aString). If you need to lengthen a string, use the insert procedure or include function. See 3.3, 4.3.1, and 10.5 of Chapter 17. Example:

```
var
  a: integer;
  s: string[8];
begin
  s := 'abcdefgh';
  for a := 1 to 9 do
    writeln(s[a]);
end;

aString := 'foo';
aString[3] := 'u';      { OK: current length of aString is 3      }
aString[4] := 'r';      { Wrong: can't append to string this way }
insert('r', aString, 4); { Right: aString = 'four'                }
```

String size must be a number between 1 and 255.

See 3.3 of Chapter 17. Example:

```
var
  aStringFloating : string[3.4];      { Wrong: string size not an integer }
  aString256 : string[256];           { Wrong: string size > 255          }
  aStringMinusOne : string[-1];       { Wrong: string size < 1            }
  aString5 : string[5];               { OK }
```

String too large.

If a string is being read into by a `read`, `readln`, or `readstring` statement, the destination string must be big enough to accommodate the result. See 9.4.1.4 of Chapter 17.

```

var
    aString5 : string[5];
begin
    readstring('The input string has more than 5 chars', aString5);
                                                { Error }
end;
```

Subrange boundaries are not of the same type.

Two constants in a subrange definition must have compatible types. See 3.1.1.3 of Chapter 17.

Example:

```

var
    SubrangeOfDifferentTypes : red..HEAVY;   { Wrong }
    GoodSubRange : red..yellow;              { Right }
    BadSet : set of red..HEAVY;              { Wrong }
    GoodSet : set of red..blue;              { Right }
    AlsoGoodSet : set of ColorType;          { Also Right }
```

Subrange boundary is not integer, char, or enumerated.

See 3.1.1.3 of Chapter 17.

```

type
    BadRealType = 2.5..3.1;   { Error: real in subrange }
                           { is not allowed }
var
    RealSubrange : 2.5..3.1; { Error: real in subrange }
                           { is not allowed }
```

Subrange error.

The Range compile option inserts code which checks for out of range errors in your program (see Chapter 15). This runtime error occurs when this check has failed. Example:

```

var
    Small : 1..5;
    Big : -20000..20000;
begin
    anInt := 6;
    Small := anInt;      { Error }
    Small := anInt - 3;  { OK }
    anInt := 30000;
    Big := anInt;        { Error }
end;
```

Superclass type must be object

The heritage of an object must be another object. If you declare objects whose superclass is TObject, don't forget to add ObjIntf.p to your project.

```
program test;
  uses
    ObjIntf;
  type
    TestObj = object(TObject)      { CORRECT }
  end;
    BadTestObj = object(integer)   { ERROR: integer isn't an object }
  end;
begin
end.
```

Tag type must be ordinal.

See 3.1.1 and 3.2.2 of Chapter 17.

```
type
  BadVariantRecordType = record
    case FloatingPointTag : Real of
      3.1416 : (                { Wrong: tag type can't be Real }
        Pi : string[2]
      );
  end;

  GoodVariantRecordType = record
    case i : integer of
      3 : (                    { Right }
        Pi : string[2]
      );
  end;

  OKVariantRecordType = record
    case longTag : longint of
      $12345 : (               { OK: tag type can be Longint }
        L : string[8]
      );
  end;
```

The entire read will be re-executed when you continue.

If you click on the Bug Spray Can while reading from the Text window with read or readln, the read will stop and all pending input will be thrown away. When you continue the program, the entire read will be started over again.

The jump table is too large (actual size x bytes, limit is y bytes)

The jump table is too large. If you didn't turn on the "Far Code" option in the Set Project Type... dialog, try turning it on.

If "Far Code" is...	and you are	The jump table limit is
off	running under THINK Pascal	65,534 bytes
off	building an application	32,766 bytes
on	running under THINK Pascal	262,144 bytes
on	building an application	262,144 bytes

For more information on the "Far Code" option, see "Building applications with large jump tables" in Chapter 12, "Building Projects."

The requested document couldn't be opened, because there is already a project open.**The requested document couldn't be opened, because all available windows are in use.**

When you're running under System 7.0, THINK Pascal displays these errors when it receives an AppleEvent it can't honor. THINK Pascal can't open a project document if one is already open, and it can't open more than 16 windows.

This declaration or statement doesn't belong here.

See Section 8 of Chapter 17. Example:

```
unit BadUnit;
interface
    label          { Error: Can't have a label here }
    0;
implementation
    label          { Error: Can't have a label here }
    1;
end.
```

This doesn't make sense as a statement.

Everything between a **begin** and an **end** must be a statement. See 2.1 and 6 of Chapter 17. Example:

```
program NonStatements;
begin
    { The following are illegal because they aren't statements. }
    inline
    string;
        aBool = false;    { This is an expression, not an assignment }
    const
    unit bad;
    var
    type
    const
end.
```

This doesn't make sense.

This error means that THINK Pascal finds something that doesn't fit the syntax diagrams. See Chapter 17. Example:

```

program;          { Error: missing program name. }
type
  StringType = string;
var
  BadArray : array[5] of char;      { Wrong }
  GoodArray : array[1..5] of char;  { Right }
  PointerToString : ^string;        { Wrong: Can't use STRING in      }
                                      { an anonymous type          }
  PointerToString : ^StringType;    { Right }
  π : Extended;                    { Wrong: can't use special characters }
                                      { in identifiers           }
  Pi : Extended;                   { Right }
procedure ProcWithoutArgs;
begin
  aChar := 'π';                    { OK: can use these set for strings & comments }
end;

begin
  anInt :=;                        { Error: missing value }
  ProcWithoutArgs();              { Wrong: Procs and Funcs without }
                                      { args mustn't have () }
  ProcWithoutArgs;                { Right: It does look funny if you are }
                                      { used to the C Language }
  if true then
    writeln('Error: semicolon doesn't belong at end of this line ->');
  else { See 6.2.2.1 of Chapter 17 }
    writeln('OK: the problem is with the previous writeln');
  writeln('Error: Missing close quote -> ');
  Error: missing open comment }
  { Error: missing close comment
end.

```

This file is marked "read-only" by Projector. Choose "Save As..." to make an editable copy.

The **Projector-Aware** option is on, and you just tried to edit a file that Projector marked read-only. To edit the file, you must make a copy of it. Choose **Save As...** and enter a new file name in the standard file dialog. THINK Pascal copies the file's text, but not its Projector resource. For more information on the **Projector Aware** option, see "Using MPW PROjector with THINK Pascal" in Chapter 6, "Editing."

This is not allowed in the Instant window.

The THINK Pascal Instant window is special. It can execute any Pascal *statement* that would be legal at the point where your program is currently halted, except for **gotos**. See Section 6 of Chapter 17.

New declarations are not allowed. You might want to declare some scratch variables in your main program for use in the Instant window.

This statement or keyword doesn't belong here.

Possible misplaced or missing keywords. See 8.1, 2.1, and 8.3 of Chapter 17. Example:

```

program BadProgram;
  writeln('BAD');    { Error: statements don't go in declaration part }
  i : integer;      { Error: Need VAR before declaring variables }
  type
    t = integer;
  and;             { Error: END doesn't go after TYPE declaration }
begin
end. { End of BadProgram }

```

Another example:

```

unit BadUnit;
interface
  label
    13;              { Error: labels can't be made visible outside the unit }
implementation
  label
    13;              { Error: labels can't be shared inside the unit }

  procedure empty;
  begin
  end;
end. { End of BadUnit }

```

Too few parameters used in procedure or function call.

See 7.3 of Chapter 17.

Too many constants in enumerated type.

For efficiency, enumerated types are implemented as unsigned byte values ranging from 0 to 255. Therefore, an enumerated type can have a maximum of 256 enumerated constants.

Too many indices are being applied to a variable or expression.

Array elements have to be accessed the same way in which they are declared. See 4.3 and 4.3.1 of Chapter 17. Example:

```

var
  OneDim : array[1..10] of integer;
  TwoDim : array[1..10, 1..10] of integer;
begin
  OneDim[1][2] := 3;      { Error }
  OneDim[1, 2, 3, 4] := 4; { Error }
  OneDim[3] := 5;         { OK }
  TwoDim[1][2] := 3;      { OK }
  TwoDim[1, 2] := 4;      { OK: Pascal allows either form }
end;

```

Too many NEAR-placed modules.

You're compiling a project with the "Far Code" option on, and it contains too many libraries compiled with the "Far Code" option off. Try recompiling the libraries with the "Far Code" option on, or put all the libraries with the "Far Code" option off together in their own segment. For more information on the "Far Code" option, see "Building applications with large jump tables" in Chapter 12, "Building Projects."

Too many nested {\$IFC} or {\$PUSH} directives

You can nest {\$IFC} directives only 16 levels deep and {\$PUSH} directives only 8 levels deep.

```

procedure test;
begin
  {$IFC TRUE}
  {$IFC TRUE}
  {$IFC TRUE}
  {$IFC TRUE}
  {$IFC TRUE}
  {$IFC TRUE}
  {$IFC TRUE}
  {$IFC TRUE}
  {$IFC TRUE}
  {$IFC TRUE}
  {$IFC TRUE}
  {$IFC TRUE}
  {$IFC TRUE}
  {$IFC TRUE}
  {$IFC TRUE}
  {$IFC TRUE} { ERROR: limit of 16 nested $IFC directives }
  writeln;
end;

  {$PUSH}
  {$PUSH}
  {$PUSH}
  {$PUSH}
  {$PUSH}
  {$PUSH}
  {$PUSH}
  {$PUSH}
  {$PUSH}
  {$PUSH} { ERROR: limit of 8 nested $PUSH directives }
  {$R-}
procedure NullifyString (s:string);
...

```

Too many parameters used in procedure or function call.

See 7.3 of Chapter 17.

Too many up-arrows (^) are being applied to a variable or expression.

Too many up arrows result in too many dereferences. See 4.3 and 4.3.4 of Chapter 17.

```

type
  pType = ^integer;
  ppType = ^pType;
var
  p : pType;
  pp : ppType;
  i : integer;
begin
  anInt := p^^;           { Wrong }
  anInt := p^;            { Right }
  anInt := pp^^^;         { Wrong }
  anInt := pp^^;          { Right }
end;

```

Type expected. "symbol" isn't a type.

See sections 3 and 4 of Chapter 17. Example:

```

var
  NotAType : integer;
  s : set of NotAType; { Error: NotAType is not a type }
  i : NotAType;       { Error: NotAType is not a type }
  BadArray : array[aConstant] of char; { Wrong: needs a type }
                                           { or subrange }
  GoodArray : array[1..aConstant] of char; { Right }
type
  aBadType = NotAType; { Wrong }
  aGoodType = integer; { Right }
begin
end;

```

Type Incompatibility between an actual and formal procedural or functional parameter.

The functional or procedural parameter list must match the parameter list declared in the called subroutine. See 7.3.5 of Chapter 17. Example:

```

procedure P1 (r : real);
begin
end;
procedure P2 (x, y : integer);
begin
end;
procedure CallP (procedure aProc (i : integer));
begin
  aProc(4);
end;
function F (i : integer) : real;
begin
end;

```

```

procedure CallF (function aFunc (i : integer) : integer);
begin
    writeln(aFunc(4));
end;
begin
    CallP(P1);    { Error: P1 doesn't have same parameter types as aProc }
    CallP(P2);    { Error: P2 doesn't have same number of args as aProc }
    CallF(F);     { Error: F doesn't have same result type as aFunc }
end;

```

Type Incompatibility between an actual and formal value parameter.

Pascal requires assignment compatibility between *actual* value parameters (the actual expressions passed to a subroutine) and *formal* value parameters (the arguments in the subroutine definition). Assignment compatibility of types can be subtle. See 7.3.1 and 3.5.3 of Chapter 17. Example:

```

type
    Canonical = array[1..4] of char;    { a type }
    Identical = Canonical;              { an identical type with Canonical }
    AnotherType = array[1..4] of char; { type different from Canonical }
var
    AnonArray : array[1..4] of char;    { an array of an anonymous type }
    CanonicalArray : Canonical;          { a variable of a named type }
    IdenticalArray : Identical;          { a variable of a same named type }
    AnotherArray : AnotherType;         { a variable of a different named type }
    PAC : packed array[1..6] of char;
procedure RealValue (r : real);
begin
end;
procedure IntValue (i : integer);
begin
end;
procedure ArrayValue (a : Canonical ); { type of a must be named }
begin
end;
begin
    RealValue(aReal);    { OK }
    RealValue(anExt);    { OK: anExt gets converted to real }
    RealValue(anInt);    { OK: anInt gets converted to real }
    RealValue(aLong);    { OK: aLong gets converted to real }
    IntValue(aLong);     { OK: aLong gets converted to integer }
    IntValue(aColor);    { Error: enums don't get converted to integer }
    IntValue(aReal);     { Wrong: reals don't get converted to integer }
    IntValue(round(aReal)); { Right }
    IntValue(aPtr);      { Error }
    anInt := length('string'); { OK }
    anInt := length(aChar); { OK: aChar is treated as a string }
    anInt := length(PAC);  { OK: PAC [1..N] is treated as a string }
    anInt := length(AnonArray); { Error: unpacked arrays don't }
                                { get converted }

```

```

{ These examples are subtle - see 3.5.1 of Chapter 17 }
ArrayValue(CanonicalArray);    { OK }
ArrayValue(IdenticalArray);    { OK: the types are identical      }
ArrayValue(AnonArray);         { Error: anonymous types aren't    }
                                { identical with anything         }
ArrayValue(AnotherArray);      { Error: the named types are different }
end;

```

Do not redeclare a type that is in one of the built-in interfaces. If you redeclare one of those types and try to call a Toolbox routine that requires an argument of that type, you get this error. Example:

```

program test;
type
  rect = record
    top, left, bottom, right: integer
  end;
var
  r: rect;
begin
  SetRect(r, 10, 5, 200, 100);
  ...
end.

```

Type Incompatibility between an actual and formal VAR parameter.

Pascal requires the types of actual **var** parameters (the variable passed to a subroutine) and the types of formal **var** parameters (the arguments in the subroutine definition) to be *identical*. When a parameter is passed to a formal **var** parameter, the called subroutine can change the actual parameter passed, not just a copy of it. Therefore, the data representation of the actual parameter must exactly match the data representation of the formal parameter. Type identity can be subtle. See 7.3.2 and 3.5.1 of Chapter 17.

```

type
  Canonical = array[1..4] of char;    { a type }
  Identical = Canonical;              { an identical type with Canonical }
  SomeType = array[1..4] of char;     { type DIFFERENT from Canonical }
var
  AnonArray : array[1..4] of char;    { an array of an anonymous type }
  CanonicalArray : Canonical;          { a variable of a named type }
  IdenticalArray : Identical;          { a variable of a same named type }
  SomeArray : SomeType;                { a variable of a different named type }
  PAC : packed array[1..6] of char;

```

```

procedure RealVar (var r : real);
begin
end;
procedure IntVar (var i : integer);
begin
end;
procedure StringVar (var s : string);
begin
end;
procedure ArrayVar (var a : Canonical );      { type of a must be named }
begin
end;

begin
    RealVar(aReal);           { OK }
    RealVar(anExt);           { Error: types must match exactly for VAR }
    RealVar(anInt);           { Error }
    RealVar(aLong);           { Error }
    IntVar(anInt);            { OK }
    IntVar(aColor);           { Error }
    IntVar(aLong);            { Error }
    IntVar(aReal);            { Error }
    IntVar(aString);          { Error }
    IntVar(aPtr);             { Error }
    StringVar(aString);       { OK }
    StringVar(aString25);     { OK: any string type is identical to }
                                { type VAR s:STRING }
    StringVar(aChar);         { Error }
    StringVar(PAC);           { Error }
    StringVar(AnonArray);     { Error: unpacked char arrays }
                                { don't match strings }
    { These examples are subtle - see 3.5.1 of Chapter 17 }
    ArrayVar(CanonicalArray); { OK }
    ArrayVar(IdenticalArray); { OK: the types are identical }
    ArrayVar(AnonArray);     { Error: anonymous types aren't }
                                { identical with anything }
    ArrayVar(AnotherArray);   { Error: the named types are different }
end;

```

Type Incompatibility.

You can't add apples and oranges. Pascal catches illogical statements that try to use incompatible types. On occasion, this rule needs to be broken. The following examples show common errors and methods to break the rules. Use the methods at your own risk. Type compatibility can be subtle. See 3.5.2 and 5 of Chapter 17. Example:

```

var
    PtrToInt : ^integer;
    PtrToChar : ^char;
begin
    aBool := (aColor = 1);           { Wrong: can't compare }
                                      { incompatible types }
    aBool := (Ord(aColor) = 1);      { Right: but what do you }
                                      { really want to do? }

```

```

aBool := (aColor = ColorType(1)); { Also Right: but what do you      }
                                   { really want to do?                }
aBool := (aString = aChar);       { OK to compare strings to char  }
aBool := (aString = anInt);       { Wrong: strings and integers    }
                                   { aren't compatible                }
aBool := (aString = stringof(anInt : 1)); { Right: anInt is converted }
                                   { to a string                      }
aBool := (red in [red..violet]);   { Right }
aBool := (2 in [red..violet]);     { Wrong: 2 isn't in          }
                                   { set of ColorType                }
aBool := (red in [red..violet]);   { Right }
aBool := ('a' in [$OD..'z']);      { Wrong: $OD is a different    }
                                   { type from 'z'                    }
aBool := ('a' in [chr($OD)..'z']); { Right: chr($OD) is          }
                                   { a carriage return                }
aBool := (PtrToInt = PtrToChar);   { Wrong: pointers are of      }
                                   { different types                  }
aBool := (pointer(PtrToInt) = PtrToChar); { Right: but what do you }
                                   { really want to do?                }

```

end;

Type or procedure name used where a variable, field name, or value is required.

See Sections 5 and 6 of Chapter 17. Example:

```

type
  aType = integer;
  aName = integer;
var
  aRecord : record
    aName : integer
  end;

  procedure aProc;
  begin
  end;
begin
  anInt := aName;      { Error: aName refers to a type in this statement }
  if anInt = aType then { Error: aType is not a value                    }
  ;
  with aRecord do
    begin
      anInt := aName; { OK: aName refers to field in          }
                      { aRecord in this statement            }
      anInt := aProc; { Error: aProcedure is not a value      }
      anInt := aType; { Error: aType is not a value           }
    end;
end;

```

Unexpected end of file.

EOF (End Of File) was reached while doing a Read. You should always check for EOF before reading. See 9.9 of Chapter 17. Example:

```
RESET(FileOfText, 'input file');
while true do      { Error: eventually tries to read past end of file}
    readln(FileOfText, aString);
close(FileOfText);
RESET(FileOfText, 'input file');
while not eof(FileOfText) do { Right: checks for EOF }
    readln(FileOfText, aString);
```

USES only allowed Immediately after PROGRAM heading , INTERFACE, or IMPLEMENTATION.

See 8.1 and 8.3 of Chapter 17. Example:

```
program WhichUsesUnits;
uses
    aUnit;          { Right }
var
    i : integer;
uses
    AnotherUnit; { Wrong }
begin
end.
```

Variable or function name expected. "symbol" Isn't a variable or function.

The target of an assignment must be a variable or a function name when a return result is being set. See 6.1.1 and 7.2 of Chapter 17.

```
type
    NotAVariable = integer;
var
    aVariable : integer;
    aPtrToInteger : ^integer;
procedure aProcedure;
begin
end;

function aFunction : integer;
begin
    NotAVariable := 4;      { Wrong: NotAVariable is not a variable      }
    aVariable := 4;        { Right }
    aPtrToInteger^ := 4;    { Also Right }
    aProcedure := 4;        { Wrong: aProcedure is not a function name    }
    aFunction := 4;         { Right: this is how functions return values }
end;
```

Variables of a file type or a type which contains a file type can't be assigned.

For every open file, there must only be one copy of the file-variable. The operating system needs file-variables for file input/output. If a copy of the file-variable could be made, and both were used

for file I/O, then the file would be inconsistent. Therefore, making a new copy of a file-variable is forbidden. Example:

```
var
  file_1, file_2 : file of integer;
  array_1, array_2 : array[1..3] of file of integer;
  record_1, record_2 : record
    FileName : string;
    FileParameter : file of real;
  end;
begin
  open(file_1, 'My file');
  file_1 := file_2;           { Error: could cause havoc with filesystem }
  close(file_2);              {      especially, if you did this      }
  array_1 := array_2;         { Error: these arrays contain filetypes }
  record_1 := record_2;       { Error: these records contain filetypes }
end;
```

Variables of this type would be too large.

Because THINK Pascal use 16 bit offsets (the MC68000 architecture doesn't allow 32 bit offsets), a data structure cannot exceed 32766 bytes.

```
type
  rec = record
    ar : packed array [0..40000] of char;
    sp : integer;
  end;
```

Variant-part of record contains no variants.

See 3.2.2 of Chapter 17. Example:

```
type
  BadVariantRecordType = record
    case tag : integer of
      { Error: must have at least one variant here }
    end;
```

When using FAR CODE, the “%_MethTables” virtual segment must be in a physical segment by itself.

When the “Far Code” option is on, the entry %_MethTables must be in a segment by itself. For more information, see “Segmenting a Project” in Chapter 7, “Working with Projects.”

While your program is halted, you may not manipulate your program's windows.

While your program is halted, your code, which handle events in your program's windows, is not running, so you can't manipulate your program's windows.

Writing over a project or library is not allowed.

This is a safety feature that prevents you from overwriting a project or library by accident. You can delete the project or library with the **Delete** command in the **File** menu.

You already have a window named "Name1". Would you like to open your file as "Untitled N"?

You may have tried to open a file twice. You may have two different files with the same name but in different volumes or folders. At this point, you can either open the file and see what's in it, or you can cancel.

```
program InputAndOutput (input, output);
begin
    reset(output);           { Wrong }
    reset(input);            { Right }
    rewrite(input);          { Wrong }
    rewrite(output);         { Right }
end.
```

You can't CLOSE an anonymous file.

Unnamed files are sometime also referred to as anonymous files. See 9.1 and 9.2.4 of Chapter 17. Example:

```
var
    f : Text;
begin
    rewrite(f); { Associate an unnamed file with f }
    close(f);   { Wrong: can't close an unnamed file }
end;           { When the scope the file variable is exited, f will be closed }
```

{\$ELSEC} or {\$ENDC} (or {\$POP}) has no matching {\$IFC} (or {\$PUSH})

Every {\$ELSEC} or {\$ENDC} must have a matching {\$IFC}, and every {\$POP} must have a matching {\$PUSH}.

```
procedure test;
begin
    {$SETC compiler_var = false}
    {$IFC compiler_var}
        writeln('This line will not be compiled');
    {$ELSEC}
        writeln('This line will be compiled');
    {$ENDC}

    {$ELSEC}      { ERROR }
    {$ENDC}      { ERROR }
end;

{$PUSH}
{$R-}
procedure NullifyLength (s:string);
begin
    s[0]:=0;
end;
{$POP}

{$POP}      { ERROR }
```

Index

Entries in **bold face** refer to menu commands. Entries in typewriter face refer to functions, methods, variables, keywords, or files.

Symbols

#, Pascal Source Converter 397
\$, Pascal Source Converter 397
* 292, 294
** 486
+ 292, 294
- 292, 294
/ 292
< 295
<= 295
<> 295
= 295
> 295
>= 295
?, Pascal Source Converter 397
@ operator 279, 296
≥ 450
≥≥ 450

A

A5 world 175
About THINK Pascal... 223
ABPackage.lib 144
abs 357
absolute value 357
accessing resource data 426
activations 264, 265-266
ADBS resource 164
Add file 237
Add Current Directory button
 SADeRez 457
Add File... 101, 136, 184, 237
Add Window 101
addition 292
aerosol can icon (see bug spray can icon)
align types, in resource descriptions 417
All Text Files option 443
 SADeRez 456
alternate input file
 SARez 449
and 293
anonymous file 332
ANS Pascal 475-479
APDA 11

apostrophe, in a string 260
appheap attribute 411
Apple Computer 11
Apple Numerics Manual, Second Edition 271
Apple Programmer's and Developer's Association 11
AppleTalk 144
AppleTalk.p 144
application
 building 150-151, 171
 globals 175, 176
 heap 174
 parameters 175
arccos 218
arcsin 218
arctan 218, 359
arctanh 218
arguments
 on stack 181
arithmetic functions 356
arithmetic operations
 checking for overflow 209
arithmetic operators 268, 292
Arrange... 77, 252
arranging windows 77
array data, in resource descriptions 423
array type, in resource descriptions 418
array-type 273
\$\$ArrayIndex() 427
arrays 273, 284
 checking index bounds 210
 examining in LightsBug 192
 over 32K 484
 over 32K elements, in LightsBug 193
 referring to elements 285
 representation 178
arrow keys 81
assembly code, examining 129
assembly language 139, 173-184
assignment
 checking bounds 210
assignment compatibility 281
assignment statements 302
\$\$Attributes 411, 437
attributes
 in SARez 411

Auto-Reformat 79, 232
Auto-Reopen 77, 253
Auto-Save 117, 248
Auto-Show Finger 123, 251
 automatic execution commands 127

B

backing up files 386
BAND 366
 base-type 276, 279
BCLR 367
 Beekman, George 9
\$BEEP 400
BEGIN starts a new line option 91
 binary operations 287
BInlineF 373
 bit operations 365-368
BitAnd 365
\$\$BitFieldsd() 426, 437
BitNot 365
BitOr 365
 bitstring type, in resource descriptions 415
BitXor 365
 blocks 262
 activation 265
BNDL resource 148
BNOT 366
 boolean 268
 definition 268
 operators 293
 representation 177
 short circuit 492
 Boolean types, in resource descriptions 415
BOR 366
 bounds checking (see Range compiler directive)
%_BP 208
Break at A-Traps 201, 251
 breakpoints (see Stop Signs)
BROTL 367
BROTR 367
Browser 253
BSET 368
BSL 366
BSR 367
BTST 367
 bug icon 114
 bug spray can icon 115, 122
Build 117, 246
Build Application... 171, 238
Build Code Resource... 171, 238
Build Desk Accessory... 171, 238
Build Driver... 171, 238
Build Library... 137, 238
 build order 102

building programs 117
 built-in functions, resource descriptions 426, 436
Bundle bit 148
BXOR 366
 byte type, in resource descriptions 415

C

callback routines, in code resources 170
 callback routines, in drivers 156
 calling conventions 180-182
 calling sequence 180
 can, bug spray (see bug spray can icon)
case label subranges 491
case statements 308
 case-constants 308
 cash register icon (see LightsBug, register display)
CDEF resource 164
 cdev resource 164
 change statement 422
 changed attribute 412
 char 269
 definition 269
 representation 177
 subrange 270
 character string 260
 character types, in resource descriptions 415
 character-pairs 258
 chars
 comparing 295
Check Link 115, 117, 246
Check Syntax 117, 246
 Chernicoff, Stephen 11
 choosing types to decompile 459
 chr 359
CKID resource 93
 Clancy, Michael 9
Class Browser 253
 classes 277-278
 in libraries 139
 public 139
Clear 229
 clicking
 Command-click in close box 77
 Command-click in title bar 82
 Command-click on compiler option 109
 Command-Option-click in title bar 82
 Option-click in close box 87
 Option-click in project window 102
 Option-click in title bar 76
 Option-click on compiler option 109
 Option-click on definition 82
 triple-click on line 81

- Close** 87, 224, 335
 - SADeRez 462
 - SARez 452
- Close All** 86, 87, 224
- close box 87
 - Command-clicking 77
 - Option-clicking 87
- Close Project** 100, 236
- closing
 - files 87
 - projects 100
- cmnu resource 463
- code resources
 - building 163, 171
 - global data in 166
 - headers 165, 168
 - ID number 165
 - locking 167
 - main function 165
 - multi-segment 168
 - naming 165
 - reentrant 167
 - setting attributes 165
 - setting the project type 164
 - type field 165
 - using callback routines 170
 - using trap intercept routines 170
 - writing 165
- coercion (see type casting)
- collected views 187, 194
- collections (see LightsBug, collected views)
- Command Line box 442
 - SADeRez 454
- Command-clicking
 - in close box 77
 - in title bar 82
 - on compiler options 109
- Command-Option-clicking
 - in title bar 82
- Command-Shift-Period 115, 122
- comments 262
 - nested 399
- Compare, MPW 400
- compatibility
 - assignment 281
 - type 281
- compatibility with earlier versions 467
- compdate 375
- compilation, changing order of files 102
- compilation time routines 375
- Compile Options...** 215, 242
- compile-time variables 214
- compiled code, examining 129
- compiler directives 203-219, 243
 - \$ELSEC 214
 - \$ENDC 214
 - \$IFC 214
 - \$SETC 214
 - Debug 207
 - External Routine 213
 - External Variable 212
 - Initialization 211
 - MPW Pascal 399, 400-404
 - Names 207
 - Overflow 209
 - page break 214
 - Pascal Source Converter 400-404
 - placing 206
 - Pop 213
 - Push 213
 - Range 210
 - segmentation 105, 213
 - Tracing 208
- compiler options 108, 203, 262
 - Command-clicking 109
 - Option-clicking 109
 - setting for all files 109
- compiler variables 216
- compiling programs 117
- component-type 273, 277
- compound statements 306
- comptime 375
- CompuServe 12
- computational 271
 - representation 177
- concat 361
- conditional compilation 214
 - Pascal Source Converter 397
- conditional statements 306-309
- Confirm Saves** 117, 248
- conjunction 293
- constant declarations 261
- constant expressions 490
- constant-declaration-part 263
- control entry, in drivers 154
- control procedures 368
- control-variable 311
- conversion routines 355-356
 - from strings 371
 - integer to char 359
 - integer to pointer 356
 - ordinal or pointer to longint 356, 359
 - real to longint 355
 - to strings 370
- Converter, Pascal Source 395, (see also Pascal Source Converter)
- Cooper, Doug 9
- Copy** 228, 361

cos 218, 358
 cosh 219
 \$\$countof () 418
 creating files 75
 creating projects 98
 \$CREATOR 400
 creator 148
 cstring type, in resource descriptions 416
 Cut 228
 Cycle 369, 490

D

\$D 207
 DA Shell 162
 data statement 412
 data type representation 176
 data types 176
 data, in resource descriptions 423
 array 423
 switch 423
 data-type, in resource description (see types)
 \$\$Date 436
 \$\$Day 437
 Debug compiler directive 207
 Debug option 123
 Debugger 375
 DebugStr 375
 declaration-part 262
 declarations
 constants 261
 functions 318
 methods 327
 procedures 315
 scope 264
 variables 283
 Decompile box 459
 defining-declaration 316
 definitions, finding 82, 190
 delete 362
 delete statement 421
 Delete... 227
 \$DEPEND 401
 Description Files... button
 SARez 443
 desk accessories (see also device drivers) 151-163
 debugging 162
 device drivers (see also desk accessories) 151-163
 building 151-163, 171
 closing 159, 160
 control entry 154
 dCtlDelay 158, 159
 dCtlEMask 158, 159
 dCtlFlags 158, 159
 dCtlMenu 158, 159

drvREMask 158, 158
 drvRFlags 158, 158
 event mask 158
 event record 155
 flags 158
 global data in 155
 headers in 157
 how they work 153
 I/O parameter block 154
 jlODone 160-161
 libraries, imported 157, 167, 183
 main function 154
 multisection 153, 161
 naming 152, 153
 opening 159
 preloading segments 161
 returning from 159, 160
 segmentation 163
 setting project type 152, 153
 unloading segments 162
 using callback routines 156
 using Object Pascal 153
 using QuickDraw globals 157, 166
 using trap intercept routines 156
 writing 154
 Device Manager 153
 devices 351-352
 difference 294
 .diff files 400
 \$DIFFs 401
 \$DIR 401
 directive entry 107
 directives 259
 directives, compiler (see compiler directives)
 disjunction 293
 dispose 354
 div 292, 293
 division 293
 Don't escape characters option 459
 Don't Find button 84
 Don't Save 117, 248
 double 271
 range 271
 representation 177
 drawer icon (see LightsBug, collected views)
 Drawing 254
 Drawing window
 routines 363-365, 370
 using in applications 150
 drivers (see desk accessories and device drivers)
 DRVRRuntime.lib 141, 153, 156
 DumpProfile 393
 DumpProfileToFile 393
 dynamic allocation routines 353, 355

dynamic-variables 279, 287

E

Echo to file 119
 Echo to printer 119
 editing files 78-83
 editing values, in LightsBug 188, 195
 Elems881 216, 217
else 307
 \$ELSEC 214, 401
 empty set 277
 End key 82
 \$ENDC 214, 401
Enter Selection 84, 235
 Entire Document 88
 enumerated types 269
 comparing 295
 representation 177
 subrange 270
 eof 336
 eoln 347
 %_EP 208
 equal to 295
 error file
 SADeRez 457
 SARez 449, 452
 error messages 493
 errors
 input/output 352
 moving to 81
 SADeRez 461
 SAREZ 451
 syntax 79
 escape characters, resource descriptions 439
 event mask, drivers 158
 event record, in drivers 155
 %_EX 208
 examples
 labels in SARez 429
 sample resource description file 409, 423
 sample resource type statement 420
 execution commands 127
 execution finger 116, 122, 123
 with "A" 201
 execution position (see execution finger)
 Exit 369, 490
 exp 219, 358
 exp1 219
 exp10 219
 exp2 219
 expanded view 187, 192
 exponentiation 486
 exporting routines, types, variables 132

expressions 287-301
 constant 490
 in constants 261
 simple 290
 syntax 291
 expressions, in resource descriptions 435
 extended 271
 range 271
 representation 177
 external 139
 external declarations 317
 External Routine compiler directive 213
 External Variable compiler directive 212
 eye on folder icon (see LightsBug, watchpoints)

F

factor 289
 fake nose and glasses icon (see LightsBug, type cast value)
 Far Code option 180
 fields 274, 286
 file cabinet icon (see LightsBug, collected views)
 file entry 104, 107
 File To Decompile button 455
 file type 148
 File Windows 254
 file-buffer 286, 333
 file-type 277
 file-variable 286
 filepos 337
 files (see also units), 277, 332
 accessing randomly 333
 accessing sequentially 333
 adding to projects 101
 anonymous 332
 backing up 88, 386
 buffer 333
 changing compilation order 102
 closing 87
 components 333
 creating 75, 332
 creating function 372
 declaration 277
 editing 78-83
 file-type 277
 file-variable 286
 moving in 81, 82
 moving in projects 102
 moving in segments 104
 non-textfile routines 337-339
 opening 75, 76, 77, 332
 opening function 371
 options, SADeRez 460
 options, SARez 450

- organizing 96
- position 333
- printing several 86, 384
- referring to components 286
- removing from projects 102
- replacing files in projects 102
- representation 179
- routines for all 333-337
- saving 87, 117
- saving with new name 87
- text file routines 339, 351
- without external file 332
- files, standard type declaration 407
- fill types, in resource descriptions 417
- Find Again** 83, 85, 234
- Find button 84
- Find in All Files** 86
- Find in Next File** 86, 234
- find options 84
 - setting up for later 84
- finding definitions 82
 - LightsBug 190
- finding selection 81
- Find...** 83, 233
- finger icon (see execution finger)
- fixed-part 274
- FKEY resource 164
- folder with eye icon (see LightsBug, watchpoints)
- Folsom, Rachel 9
- font, changing 89
- for** statements 311
 - exiting 368
- formal parameter list 320
 - compatibility 325
- \$\$Format () 436
- formatting (see pretty-printing)
- forward declarations 316
- FPU
 - initializing 212
 - using 217-219
- free memory* 174
- function-declaration 318
- function-heading 319
- functional parameters 321, 325
- functions 315
 - address of 298
 - arithmetic 356
 - as l-values 299
 - calling 174, 298-300
 - calling conventions 180-182
 - calling sequence 180
 - declarations 318
 - entry 180
 - examining in LightsBug 188

- exit 181
- external 139
- external declarations 317
- forward declarations 316
- in libraries 137
- inline declarations 318
- ordinal 359
- parameters 320
- passing parameters 182
- public 137
- recursive 265
- results, generalized 491
- return values 182
- scope 264
- stack, on entry 181
- standard 141, 353-375
- stepping into 124
- stepping over 124
- timing (see profiler)
- Toolbox (see Toolbox)
- using @ 298
- using as l-values 491
- using in other units 132

G

- Generic 374
- get 336
- Get Info...** 109
- GetDrawingRect 364
- GetTextRect 364
- global data
 - amount in project 109
 - application 175, 176
 - desk accessories 155
 - device drivers 155
 - in code resources 166
 - QuickDraw 175
 - QuickDraw, in drivers 157, 166
- glue code 142
- Go** 113, 127, 247
- Go-Go** 124, 127, 247
- goto** statements 303
- greater than 295
- greater than or equal to 295
- Groucho Marx icon (see LightsBug, type cast value)

H

- halt 369, 490
- hand icon (see execution finger)
- handles
 - checking for nil handle 210
 - dereferencing in LightsBug 192, 199

- headers
 - code resources 168
 - in code resources 165
 - in drivers 157
- heap 174
- heap display 187
- heap icon (see LightsBug, heap display)
- heap size 119
- heap zones
 - examining in LightsBug 197
- HeapCheck 355
- HeapResult 355
- Help box 442
 - SADeRez 454
- hexadecimal numbers 259
- HideAll 363
- hiding project window 100
- hiding windows 77
- HiWord 368
- HiWrd 368
- Home key 82
- host-type 270
- \$\$Hour 437
- How to Write Macintosh Software* 11
- I**
 - \$I 211, 398
 - I/O parameter block, in drivers 154
 - icons
 - LightsBug 187
 - \$\$ID 411, 437
 - identical types 280
 - identifiers 258
 - scope 264
 - identity 293
 - IEEE 218
 - if statements 307
 - if-then-else directives 432
 - \$IFC 214, 397, 401
 - implementation 132
 - Implementation uses 135
 - implicit parameters 326
 - importing routines, types, variables 132
 - in 295
 - include 363
 - include file directive 398
 - Include Paths... button 447
 - \$INCLUDES 402
 - include statement 410
 - Indent BEGIN/END within statements option 91
 - Indent statements within BEGIN/END option 92
 - indentation, changing 91
 - index-type 273
 - indices 284
 - INIT resource 164
 - InitFPState 212
 - initialization, automatic 211
 - Initialization compiler directive 211
 - InitProfiler 393
 - inline declaration 318
 - InlineP 373
 - \$INPUT 396, 402
 - input parameter 328, 333
 - input/output 331
 - error handling 352
 - lazy 348, 351
 - modem 351-352
 - printer 351-352
 - insert 363
 - insertion point
 - moving 81, 82
 - moving to 81
 - Inside Macintosh* 10
 - Instant 251
 - Instant Project option 99
 - Instant window 122, 129
 - saving 129
 - integer type, in resource descriptions 415
 - integers 268
 - comparing 295
 - converting 356, 359
 - range 268
 - representation 176
 - subrange 270
 - syntax 259
 - interactive I/O 348, 351
 - interface 132
 - interface file 136, 137
 - Interface.lib 98, 142
 - internationalization of resources 421
 - intersection 294
 - IOCheck 352
 - IOResult 352
 - J**
 - \$J 212
 - Jensen, Kathleen 9
 - jIODone, in drivers 160
 - Johnson, Michael 9
 - jump table 175, 176
 - K**
 - keywords
 - appearance in files 90
 - Knaster, Scott 11
 - Kronick, Scott 9

L

L-values

- function calls 299
- function results as 491
- type casting 492

label-declaration-part 263

labels

- declaring 263
- scope 264
- syntax 260

labels, in resource descriptions 425

- declaring in arrays 427
- examples 429
- limitations 427

Large sets option 219

lazy I/O 348, 351

LDEF resource 164

Leave 369, 490

Leave behind "CMNU" resources 464

length 360

- of strings 278

less than 295

less than or equal to 295

libraries 136

- adding to projects 101
- imported 182
- imported, in desk accessories 183
- imported, in device drivers 183
- imported, in drivers 157, 167
- in drivers 157, 166
- interface files 137
- standard 141, 353-375
- using 136
- writing 137

LightsBug 185, 250

- collected views 187, 194
- dereferencing handles and pointers 192, 199
- edit values 188, 195
- editing memory 200
- examining arrays 192
- examining many variables 194
- examining objects 192
- examining records 192
- examining sets 192
- examining subroutines 188
- examining variables 191
- expanded view 187, 192
- finding subroutine definitions 190
- heap display 187, 197
- icons 187
- memory display 198
- opening several 186
- panes 186

register display 187, 196

scope 191

subroutine call chain 188

Toolbox routines 201

trash 188

type casting 188, 195

variable display 187, 191

watchpoints 187, 194

\$LINE 402

lines

- selecting 81

Link Errors window 115

linking programs 115, 117

LInLineF 373

literals, in resource descriptions 434

ln 219, 358

ln1 219

locked attribute 412

locking code resources 167

log10 219

log2 219

\$\$Long () 426, 437

Long names option 208, 219

longint 268

- range 268

- representation 176

longint type, in resource descriptions 415

LoWord 368

LoWrd 368

M

MacApp

- memory requirements 5

MacApp debugger 208

MACDEV 12

machine code, examining 129

Macintosh Pascal 9

Macintosh Pascal Illustrated: The Fear and Loathing Guide 9

Macintosh Pascal Programming Primer 11

Macintosh Programmer's Workshop 102, 184, 238

Macintosh Programming Secrets 11

Macintosh Revealed 11

Macintosh Toolbox (see Toolbox)

macro variables

- SADeRez 460

- SARez 448

Macsbug 129, 375

MacTutor 11

magnifying glass icon (see LightsBug, expanded view)

main function

- code resources 165
- desk accessories 154
- device drivers 154

Main segment 106
 Make Resource File Read-only option 445
 Mark, Dave 11
 Marx, Groucho icon (see LightsBug, type casting)
 Match Case option 84
 maximum file size 80
 maxint 268
 maxlongint 268
 MBDF resource 164
 MC68020 and MC68030, using 217
 MC68881 and MC68882
 initializing 212
 using 217-219
 MDEF resource 164, 167
 Member 375
 memory
 editing 200
 examining, in LightsBug 198
 memory allocation routines 353-355
 memory options 119
 memory requirements 5
 memory, in applications 174-175, 176
 MENU resource 463
 Merge Resources into Resource File option 445
 message window
 SADeRez 461
 SARez 451
 method-designator 303
 methods 277
 declarations 327
 «%_MethTables» 108
 \$\$Minute 437
 mntb resource 463
 mod 292, 293
 modem 351
 Modification date option 446
 Moll, Robert 9
Monitor 252
 \$\$Month 437
 moving to error 81
 moving to selection 81
 MPW 102, 184, 238
 porting from (see also Pascal Source Converter),
 481
 MPW .o files, using 492
 MPW Projector 93
 multi-file search 85, 234
 MultiFinder
 running under 119
 multiplication 292
 music playing routine 372

N

\$N 207

\$N++ 208
 \$\$Name 411, 436
 Names compiler directive 207
 nAppleTalk.lib 144
 negation 293
 nested uses 135
New 75, 224, 279, 353
 SADeRez 461
 SARez 451
New LightsBug 186
New Project... 98, 236
 NewFileName 372
nil 280
 No warnings for redeclared types option 459
 non-relocatable memory 174
 nonpreload attribute 412
 nonpurgeable attribute 411
not 293
 not equal to 295
 Note 372
 \$NOUSES 402
 null-string 279
 numbers 259
 hexadecimal 259
 numbers, in resource descriptions 434
 numeric types, in resource descriptions 414

O

Object Pascal
 in desk accessories 153
 in device drivers 153
Object-Oriented Programming for the Macintosh 10
 object-type 277
 objects 277-278
 examining in LightsBug 192
 routines 375
 scope of fields 265
Observe 251
 Observe window 122, 127-128
 saving 128
 odd 356
 .o files 102, 238
 using 492
Oh! Pascal 9
Oh! THINK's Lightspeed Pascal 9
 OK to Replace Protected Resources option 445
 OldFileName 371
 omit 362
 Only Files Ending in .r option 443
 SADeRez 456
 open 335
Open Project... 100, 236
 opening files 75, 76, 77
 opening projects 100

- opening units 76
- opening windows 77
- Open...** 75, 128, 129, 224
 - SADeRez 461
 - SARez 451
- operands 287
- operators 287, 291-298
 - @ 296
 - arithmetic 268, 292
 - boolean 293, 492
 - order of evaluation 291
 - precedence 288
 - set 294
 - in resource descriptions 435
- Option-click
 - in title bar 76
- Option-clicking
 - in close box 87
 - in project window 102
 - on compiler options 109
 - on definition 82
- options
 - 68020/68030 217
 - 68881/68882 217
 - compiler 108, 215
 - compiler variables 216
 - large sets 219
 - Long names 208, 219
 - memory 119
 - Profile 219
 - run 118
 - save 88, 117
 - search 84
 - source 89
 - Text window 119
 - uses features 217
- Options column (see compiler options)
- options file
 - SADeRez 460
 - SARez 450
- options, compiler (see compiler options)
- or** 293
- ord 359
- ord4 356
- order of evaluation 291
- ordinal functions 359
- ordinal types 267
 - comparing 295
 - converting 356, 359
 - ranges 268
 - standard 268
- ordinal-types 270
- ordinality 267
- organizing files 96

- otherwise** 308
- Outdent declarations option 92
- \$OUTPUT 396, 403
- output file menu buttons, SADeRez 458
- output file menus, SADeRez 457
- output parameter 328, 333
- Overflow compiler directive 209
- overlays (see segmentation)
- override 277
- P**
- \$P 214
- pack 476, 488
- packed arrays 273
 - representation 178
- packed records
 - portable 485
 - representation 178, 485
- packed-string-type 273
- \$\$PackedSize () 437
- page 348
- Page Down key 82
- Page Setup...** 86, 226
 - SADeRez 462
 - SARez 452
- Page Up key 82
- panes, LightsBug 186
- paper icon (see LightsBug, variable display)
- parameter lists, changing format 93
- parameter passing 182
- parameters 320
 - functional 321, 325
 - implicit 326
 - procedural 182, 321, 322
 - univ** 326
 - value 182, 321, 322
 - var** 182
 - variable 321, 322
- Pascal on the Macintosh: A Graphical Approach* 9
- Pascal Source Converter 395
 - # 397
 - \$ 397
 - \$input 396
 - \$output 396
 - ? 397
 - comments 399
 - compiler directives 399, 400-404
 - converting files 397
 - prompting 397
 - sample script 404
 - scripts 395
 - setting directory and file names 396

- setting input and output 396
 - using diff(ference) files 400
 - using with MPW Compare 400
- Pascal User Manual and Report* 9
- Pascal, ANS 475-479
- Paste** 228
- pencil icon 93, (see also LightsBug, edit value), 232
- pile of sand icon (see LightsBug, heap display)
- point type, in resource descriptions 417
- pointer types 279
- pointers 279-280, 287, 296, 356
 - checking for nil pointer 210
 - converting 356, 359
 - creating 279
 - dereferencing in LightsBug 192, 199
 - nil 280
 - representation 177
- pointing finger icon (see execution finger)
- \$POP 213
- Pop compiler directive 213
- pop-up menus
 - routine names 82
 - unit names 76
- \$PORTABLE± 403
- porting 481
 - from MPW (see Pascal Source Converter)
- pos 361
- ppat resource
 - defined in SAREz 429
- precedence rules 287-288
- precision
 - of real types 271
- pred 360
- predecessor 267
- predefined routines 141
- preload attribute 412
- Preprocessor box, SAREz 460
- preprocessor directives 431
 - variable definitions 432
- Preprocessor... button 448
- pretty-printing 79
 - automatic 79
 - options 89
- PrimeTime 145
- Print All Files** 86, 226
- PrintCalls.lib 145
- printer 351
- printing 86
 - all files 86
 - several several files 384
 - Text and Drawing Windows 118
 - Text window 119
- Printing Manager 145
- Printing.p 145
- PrintTraps.p 145
- Print...** 86, 226
 - SAREz 462
 - SAREz 452
- procedural parameters 182, 321, 322
- procedure statment 302
- procedure-and-function-declaration-part 263
- procedures 315
 - address of 298
 - calling 174
 - calling conventions 180-182
 - calling sequence 180
 - declarations 315
 - entry 180
 - examining in LightsBug 188
 - exit 181
 - external 139
 - external declarations 317
 - forward declarations 316
 - in libraries 137
 - inline declarations 318
 - parameters 320
 - passing parameters 182
 - predefined 141
 - public 137
 - recursive 265
 - scope 264
 - stack, on entry 181
 - standard 141, 353-375
 - stepping into 124
 - stepping over 124
 - timing (see profiler)
 - Toolbox (see Toolbox)
 - using @ 298
 - using in other units 132
- profiler 391-393, 394
 - option 219
- program parameters 328
- program** syntax 327
- programs (see also projects), 327-331
 - building 117
 - compiling 117
 - halting 369
 - linking 115, 117
 - position in (see execution finger)
 - resetting 126
 - restarting 126
 - running 113-119
 - stepping through 124
 - stopping 115-116, 122, 124
 - tracing 124
- Progress information option 446, 459
- Project name 252
- project tree 96

- project type 148
- Project Utilities 379
 - backing up files 386
 - printing files 384
 - printing window 382
 - reading display 381
 - saving file pathnames 382
 - selecting files 383
- project window 96-97
 - customizing 110
 - hiding 100
 - option-clicking 102
 - Options column (see compiler options), 204
 - segment view 104, 107
- Projector Aware** option 232, 93
- projects 95-111, (see also programs)
 - adding files 101
 - arranging files 102
 - backing up files 386
 - changing compilation order 102
 - closing 100
 - corrupted 111
 - creating 98
 - moving files 102
 - opening 100
 - printing files 384
 - removing files 102
 - replacing files 102
 - saving file pathnames 382
- Propagated **uses** 135
- protected attribute 412
- protected bit, in resource descriptions 422
- pstring type, in resource descriptions 416
- %_PTrace 394
- public
 - classes 139
 - functions and procedures 137
 - types 137
 - variables 139
- Pull All Stops** 126
- Pull Stops** 126, 251
- purgeable attribute 411
- \$PUSH 213
- Push compiler directive 213
- put 337
- qualifiers 284

Q

- QuickDraw 143
 - alternative parameter list 144
 - globals 175
 - globals, in drivers 157, 166
- Quietly Auto-Reset** 126

- Quit** 227
 - SADeRez 454
 - SARez 442

R

- \$R 210
- range
 - of ordinal types 268
 - of real types 271
- Range compiler directive 210
- read 338, 340
- read statement 412
- readln 343
- ReadString 371
- real 271
 - range 271
 - representation 177
- real numbers 271
 - comparing 295
 - converting 355
 - ranges 271
 - syntax 259
- rearranging windows 77
- record types 274
- records 274-276
 - declaration 274
 - examining in LightsBug 192
 - referring to fields 286
 - representation 178
 - variant 275-276
- rectangle type, in resource descriptions 417
- recursion 265
- redeclared type 459
- Redeclared types OK option 446
- Redirection... button 449
- reference variable 277, 287
- reference-type 277
- references 277
- register display 187
- register icon (see LightsBug, register display)
- register saving 182
- registers
 - examining, in LightsBug 196
- relational operators 295
- relocatable memory 174
- RememberA4 156, 169
- Remove** 102, 108, 137
- Remove Objects** 111, 239
- repeat statements 309
 - exiting 368
- repetitive statements 309-313
- Replace** 85, 235
- Replace All** 85, 235
- Replace and Find Again** 85, 235

- replacing 85
- representation (see types)
- ResEdit 2.1 Reference* 11
- reserved words 258
 - appearance in files 90
- Reset** 116, 126, 247, 334
- ResetProfile** 393
- resetting programs 126
- Resource Alignment box 445
- resource attributes 411
- resource declarations 407
- resource description file 406
 - comments 408
 - example 423
 - preprocessor directives 408
 - SADeRez 455, 457
 - SAREz 443
 - structure of 407
 - type declarations 408
- resource description statements 409
 - special terms 410
 - syntax 409
- resource files 118
 - making 150
 - using 149
 - SADeRez 455
- Resource Output File box, SAREz 444
- `$$Resource()` 436
- resource statement 422
- resource types
 - defining 408
- resources (see code resources)
- `$$ResourceSize` 437
- restarting programs 126
- RestoreA4** 156, 170
- return values 182
- Revert** 81, 226
- rewrite 334
- Rewrite Resource File option 445
- Rez 1.0 compatibility 459
- round 219, 355
- routines (see also functions and procedures)
 - entry 180
 - examining in LightsBug 188
 - exit 181
 - predefined 141
- `RSRCRuntime.lib` 170
- Run Options...** 118, 149, 249
- running programs 113-119
- runtime environment 174-175, 176
- `Runtime.lib` 52, 98, 141, 218
- `μRuntime.lib` 141

S

- `$$S` 105, 213
- SADeRez 406
 - choosing types to decompile 459
 - Close** 462
 - error file 457
 - errors 461
 - input files 455
 - macro variables 460
 - message window 461
 - New** 461
 - only option 459
 - Open...** 461
 - options 458
 - options file 460
 - output files 457
 - Page Setup...** 462
 - Print...** 462
 - resource description files 455, 457
 - resource file 455
 - Save** 461, 462
 - Save As...** 461, 462
 - skip option 459
- Saderez button 454
- SANE 145, 217, 218
- `SANE.p` 145
- `SANELib.lib` 145, 218
- `SANELib881.lib` 145, 218
- SAPostRez 463-464
- SAREz 406
 - alternate input file 449
 - change resource data 422
 - Close** 452
 - data statements 423
 - delete a resource 421
 - error file 449, 452
 - errors 451
 - input files 443
 - macro variables 448
 - message window 451
 - New** 451
 - options 446
 - options file 450
 - Page Setup...** 452
 - Print...** 452
 - resource description files 443
 - Save** 451, 452
 - Save As...** 451, 452
 - specify actual resource data 422
 - symbolic definitions 421
 - symbolic names 425

- Save** 87, 225
 - SADeRez 461, 462
 - SARez 451, 452
- Save a Copy As...** 88, 226
- Save All** 86, 87, 225
- Save As...** 87, 128, 129, 225
 - SADeRez 461, 462
 - SARez 451, 452
- save options 117
- Save Positions** 77, 253
- SaveDrawing 364
- saving
 - all files 87, 117
 - files 87
 - Instant window 129
 - Observe window 128
 - options 88
 - Text and Drawing Windows 118
 - Text window 119
- Schmucker, Kurt 10
- scope 264-265
 - declarations of identifiers 264
 - in LightsBug 191
 - of a declaration 264
 - of field 265
 - of interface identifiers 265
 - of standard identifiers 265
 - redeclarations 264
- scripts, Pascal Source Converter 395
- search options 84
 - setting up for later 84
- searching 83-86
 - multi-file 85, 234
- \$\$Second 438
- seek 337
- Segment Loader 175, 176
- segmentation 103-108
 - changing names 105
 - choosing how 106
 - combining segments 107
 - creating segments 104
 - desk accessories 163
 - directive 105, 213
 - editing attributes 105
 - in desk accessories 153
 - in device drivers 153
 - in drivers 161
 - in project window 104, 107
 - moving items 107
 - moving segments 104
 - Object Pascal 108
 - options 106
 - removing directive entries 108
 - size limit 103
 - segments 103, (see also segmentation)
 - in code resources 168
 - unloading 169
- Select All** 229
- selecting lines 81
- selection
 - moving to 81
- selector 308
- self 326
- «%_SelProcs» 108
- separators 257
- \$SETC 214, 404
- set membership 296
- set operators 294
- Set Project Type...** 148, 239
 - Attributes 165
 - code resources 164
 - Custom Header 165
 - desk accessories 152
 - device drivers 152
 - Flag and Mask menus 158
- set-constructors 300
- set-type 276
- SetDrawingRect 364
- sets 276-277
 - comparing 296
 - declaration 276
 - examining in LightsBug 192
 - making 300
 - representation 179
- SetTextRect 364
- setting protected bit on code resources 422
- SetUpA4 156, 169, 170
- \$\$Shell () 436
- short circuit booleans 492
- Show ".Rsrc" files only 464
- Show Clipboard** 229
- Show Error** 81, 235
- Show Finger** 123, 251
- Show Selection** 81, 235
- ShowDrawing 364
- ShowText 363
- sign-negation 293
- simple expression 290
- simple statements 302
- simple types 267
- sin 219, 358
- sinh 219
- 68020/68030 option 181
- 68881/68882 option 217

- size
 - maximum file 80
 - of strings 278
 - program heap 119
 - program stack 119
 - program zone 119
- sizeof 369
- \$SKIP 404
- Skip box 460
- smart linking 171
- Source Converter, Pascal
- Source Options...** 79, 89, 229
- Special-symbols 257
- SPLash 12
- spray can icon (see bug spray can icon)
- sqr 357
- sqrt 219, 357
- square 357
- square root 357
- stack 174, 181
- stack frame 174
- stack size 119
- standard routines 141, 353-375
- statement-part 262, 264
- statements 301-315
 - assignment 302
 - case** 308
 - compound 306
 - conditional 306-309
 - current (see execution finger)
 - for** 311
 - goto** 303
 - if** 307
 - next (see execution finger)
 - procedure 302
 - repeat** 309
 - repetitive 309-313
 - simple 302
 - structured 305
 - syntax 301
 - while** 310
 - with** 313
- Step Into** 124, 127, 247
- Step Out** 124, 127, 248
- Step Over** 124, 127, 247
- Step-Step** 124, 127, 247
- Stop Signs 124-126
 - from Instant window 129
 - ignoring all 126
 - putting in 125
 - removing 126
- stopping programs 115-116, 122
- Stops In** 125, 251
- string types 273, 278
- string types, in resource descriptions 416
- stringof 128, 370
- strings 278-279, 284
 - checking index bounds 210
 - comparing 296
 - in resource descriptions 438
 - length attribute 278
 - null 279
 - ordering 279
 - referring to characters 285
 - representation 177
 - routines 360-363, 370, 371
 - size attribute 278
 - syntax 260
- StripAddress 177
- structured statements 305
- structured-type 272
- subrange types 270
- subroutine call chain 188
- subset of 295
- subtraction 292
- succ 360
- successor 267
- superset of 295
- switch data, in resource descriptions 423
- switch types, in resource descriptions 419
- Symantec Programming Languages Association 12
- symbolic names
 - in resource description statements 425
- symbolic names, in SAREz 421
- Synch 372
- syntax errors 79
- syntax, of resource description statements 409
- syntax, checking 117
- sysheap attribute 411
- System 7.0 467
 - running under 119
- T**
- tag-field 275
- tan 219
- tanh 219
- term 290
- TerminateProfile 393
- Text** 254, 277
- Text Only 88
- Text Window 117
 - options 119
 - printing 119
 - routines 363
 - saving to file 119
 - using in applications 150
- then** 307
- THINK C 102, 183, 238

- THINK Class Library
 - memory requirements 5
 - THINK Pascal tree 96
 - THINK Reference 10
 - THINK_Pascal 216, 398
 - THINK_Pascal_Version_4 216
 - thumbs down icon 114
 - \$Time 437
 - Time Manager 145
 - timing routines (see profiler)
 - title bar
 - Command-clicking 82
 - Command-Option-clicking 82
 - Option-clicking 76
 - TMON 129
 - tokens 257
 - Toolbox
 - calling routines 142
 - debugging routines 201
 - interfaces 142-143, 488
 - interfaces, built-in 142
 - initializing 211
 - routines not in ROM 142
 - ToolScratch 167, 168
 - Trace 247
 - Tracing compiler directive 208
 - tracing programs 124
 - Transfer... 227
 - trap intercept routines, in code resources 170
 - trap intercept routines, in drivers 156
 - trash 188
 - trash icon (see LightsBug, trash)
 - tree 96
 - triple-clicking
 - on lines 81
 - trunc 219, 355
 - \$Type 411, 438
 - type 266
 - type casting 276, 301
 - in LightsBug 195
 - type casting, in LightsBug 188
 - type coercion 276
 - type compatibility 281
 - type identity 280
 - type statement 413
 - example 420
 - type-check disabling 326
 - type-declaration 266
 - type-declaration-part 263, 282
 - types 176-179, 266-283
 - array 178, 273
 - boolean 177, 268
 - char 177, 269
 - classes 277-278
 - computational 177
 - conversion routines 355-356
 - double 177
 - enumerated 177, 269-270
 - extended 177
 - file 179, 277
 - in libraries 137
 - integer 176, 268
 - longint 176
 - numbers 259
 - objects 277-278
 - ordinal 267-270
 - packed array 178, 273
 - packed record 178, 485
 - pointers 177, 279-280
 - public 137
 - real 177, 271-272
 - record 178, 274-276
 - set 179, 276-277
 - simple 267-272
 - string 177, 278-279
 - structured 272-278
 - subrange 270
 - using in other units 132
 - Types box, SAdRez 459
 - Types Files... button 455
 - types, in resource descriptions 414
 - align 417
 - array 418
 - Boolean 415
 - character 415
 - cstring 416
 - example 420
 - fill 417
 - numeric 414
 - point 417
 - pstring 416
 - rectangle 417
 - string 416
 - switch 419
 - wstring 416
- ## U
- unary operations 287
 - unchanged attribute 412
 - Undo 80, 228
 - union 294
 - units 131-136, 327-331
 - opening 76
 - syntax 328
 - unit dependencies 330
 - using 131
 - using in other units 132
 - writing 132

univ qualifier 326
 UnloadA4Seg 162, 169
 unlocked attribute 412
 unpack 476
 unprotected attribute 412
 unsigned-constant 289
until 309
 \$USES 404
uses option 217
uses clause 132, 134-136, 329
 USES Extensions option 134, 217

V

\$V 209
 value parameters 182, 321, 322
 using @ 297
var parameter 182
 variable display 187, 191
 variable parameters 321, 322
 using @ 298
 variable-declaration 283
 variable-declaration-part 263
 variable-reference 283
 variables 283-287
 compile-time 214
 compiler 216
 declaring 283
 display format in LightsBug 191
 editing, in LightsBug 195
 examining in LightsBug 191, 192, 194
 in libraries 139
 in preprocessor directives 432
 public 139
 qualified 284
 referring to 283
 SARez string variables 411
 scope 264
 scope in LightsBug 191
 using in other units 132
 variables, in resource descriptions 436
 variant 275
 variant records 275-276
 variant-part 275
 \$\$Version 437
View Options... 110, 132, 244

W

watchpoints 187, 194
 WDEF resource 164
 \$\$Weekday 438
while statements 310
 exiting 368
 Whole Words option 84
 Width of decompiled strings option 459

WIND resource
 sample window resource file 424
 windows 76
 arranging 77
 automatically opening 77
 hiding 77
 opening 77
 saving positions 77
 WInlineF 373
 Wirth, Niklaus 9
with statements 286, 313
 \$\$Word() 426, 438
 word-symbols (see reserved words)
 write 339, 344
 Write Rez 1.0 compatible output option 459
 WriteDraw 370
 writeln 347
 wstring type, in resource descriptions 416

X

XCMD resource 164
 XFCN resource 164

Y

\$\$Year 438

Z

zone
 zone size 119
 zones 174
 examining, in LightsBug 197

.



Symantec Corporation
10201 Torre Avenue
Cupertino, CA 95014-2132
408/253-9600