SYMANTEC.™

THINK C™

The Professional's Choice

User Manual

THINK C

The Professional's Choice

User Manual

Credits

User Manual	Philip Borenstein and Jeff Mattson
THINK C Application	Michael Kahl, Greg Howe, Mike Cote, and John McEnerney
THINK Class Library	Dan Podwall and Gregory H. Dow
Quality Assurance	David Allcott, Michael Rockhold, and Paul Vetri
Technical Support	Michael Carland, Mark Geschelin, and Phil Shapiro
Marketing Manager	Susan Smith
Product Manager	Philip Borenstein

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

Contents

THINK C

Getting Started

Part One

Welcome 1

Welcome to THINK C. This chapter tells you what's in your THINK C package, what equipment you need, and what you need to know to write C programs on your Macintosh.

If you don't read manuals

Read this chapter, the next chapter, and one of the tutorials. Be sure to keep this manual nearby to answer any questions that come up as you work.

If you're an experienced THINK C user

Read Chapter 2, "Installing THINK C 5.0," before installing the new version. Read Appendix A, "What's New," in this manual for information on the new THINK C 5.0 features, including the global optimizer and 100% ANSI-conformant compiler. If you read nothing else, read "Porting to THINK C 5.0" on page 458 to learn how to port your old code.

If you're learning the C language

Read Chapter 3, "Tutorial: Hello World," to learn how to run simple programs in THINK C. If you need a book on C, see "Learning C" on page 10. If you're learning C from a book written for UNIX or MS-DOS computers, look at these chapters in the *Standard Libraries Reference* to learn how to port example programs to THINK C: Chapter 2, "Using the Standard Libraries" (especially the section "Using Files and Streams"), and Chapter 3, "Using Console Windows."

Contents

What is THINK C?

THINK C is a unique development environment for the Macintosh. It features a very fast compiler, a faster linker, a time-saving optimizer, an integrated debugger, a text editor, an auto-make facility, 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 C knows when edited source files need to be recompiled. And if you edit a header file, the auto-make facility recompiles all the source files that depend on it for declarations.

With THINK C you can build Macintosh applications, desk accessories, device drivers, and any kind of code resource. The standard C libraries include all the functions specified in the ANSI C standard, as well as some additional Unix operating system functions.

You can run your program from THINK C as you work on it. Your program runs exactly as if you had opened it from the Finder, not under a simulated environment. And if you use MultiFinder or System 7, your program runs in its own partition while THINK C remains active, so you can examine and edit your source files as you watch your program run.

The THINK C development environment includes a source-level debugger that lets you debug your code exactly as you wrote it. No more translating assembly language back into C. The debugger lets you set breakpoints, step through your code, debug objects, examine variables, and change their values while your program is running. And because the debugger works along with THINK C, you can edit your source files while you're debugging.

What You Need

THINK C requires a hard drive and at least 1 Mb (megabyte) of RAM. With only 1Mb you won't be able to take advantage of MultiFinder or System 7.0, and you won't be able to use the debugger. With 2Mb, you can run small programs under MultiFinder and System 7.0. With 4 Mb, you can run comfortably under MultiFinder or System 7.0.

Which Macintosh models?

You can run THINK C on a Macintosh Classic, LC, Plus, Portable, the Macintosh SE series, or the Macintosh II series. You can run THINK C on a Macintosh 512Ke if you've upgraded it to at least 1Mb of RAM.

You can run THINK C without the debugger on any Macintosh computer with at least 1Mb of RAM. To use the debugger, you need at least 2Mb of RAM.

How much disk space?

The complete THINK C system takes up about 5.75 megabytes on your disk, not including your own files. The actual size of your system may be smaller, depending on the kinds of programs you work on.

Which System/Finder?

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

THINK C is designed to work best under MultiFinder or System 7. If you're using a Macintosh with 1Mb RAM, however, you're better off using the latest version of System 6 recommended for you machine with MultiFinder turned off.

What's in the Package

Your THINK C package consists of four double-sided floppies, this manual, the *Standard Libraries Reference*, and the *Object-Oriented Programming Manual*.

What's in the Manual

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

Getting Started

This is the section you're reading. It contains this chapter and the installation instructions. Even if you don't read manuals, be sure to read the installation instructions in Chapter 2.

Learning THINK C

This section contains three tutorials.

This chapter	describes...
Tutorial: Hello World	How to write a minimal program that uses the standard C libraries and introduces you to the basics of using THINK C.
Tutorial: MiniEdit	How to build a Macintosh application. This tutorial is based on the Sample program in *Inside Macintosh*. It shows you how to fix bugs, how to use resource files,

	and how to build a double-clickable application.
Tutorial: Bullseye	How to use THINK C's source level debugger.

Using THINK C

This section contains nine chapters that describe the different components of THINK C.

This chapter	**describes...**
Overview	How THINK C works.
The Project	The four different kinds of projects. It gives you the details of building applications, desk accessories, device drivers, and code resources. This chapter contains several code examples to make writing your program easier.
The Editor	The THINK C integrated text editor. The editor has several features to make editing C source files easier and a sophisticated searching facility.
Files and Folders	The best way to arrange your THINK C files. This chapter describes how THINK C looks for files on your disk and how it names files.
The Compiler	How THINK C compiles your source files. It also tells you how to compile ANSI-conformant code, use the global optimizer, generate code for the 68881 floating point coprocessor, use strict type enforcement, and port code.
Toolbox Routines	How to call the Macintosh Toolbox routines from your programs. It also tells you how to write Pascal callback routines and use some THINK C extensions that help you write programs for the Macintosh.

The Debugger	The source-level debugger. This chapter tells you how to control execution, how to set breakpoints, and how to examine and modify your variables as you debug.
Assembly Language	THINK C's inline assembler. This chapter also explains C and Pascal calling conventions so your assembly language routines will integrate smoothly with both your C functions and the Macintosh Toolbox routines.
Libraries	How to build and use libraries in your THINK C programs, and how to convert object code from other compilers and assemblers into THINK C libraries.

Reference

This section contains three reference chapters.

This chapter	**describes...**
THINK C Menus	The THINK C menu commands.
Debugger Menus	The source-level debugger's menu commands.
Language Reference	THINK C's implementation of the C language. Its sections are keyed to the ANSI standard.

Utilities

This section describes three utilities that help you program with THINK C.

This chapter	**describes...**
The Profiler	How to analyze the way your program runs. You can time your functions, and trace your program.
oConv	How to convert object files from other development environments to THINK C's format.

Resource Utilities

This section describes two utilities to help you create and examine resource files.

This chapter	describes...
Resource Description Files	How to write text files that you compile with SARez to produce resource files
Using SARez	How to use SARez to create resource files from resource description files.
Using SADeRez	How to use SADeRez to create resource description files from resource files.

Note

THINK C also includes ResEdit, which helps you edit resource files. It's described in the book *ResEdit 2.1 Reference* (Addison-Wesley) by Apple Computer, Inc.

Appendices

The appendices contain information on THINK C 5.0's new features, troubleshooting, error messages, and this manual's terms.

This appendix	describes...
What's New	The new features in THINK C 5.0.
Troubleshooting	How to troubleshoot your program.
Error Messages	The error messages that THINK C generates. Many descriptions include examples and tips on solving.
Glossary	Definitions for many of the terms used in this manual.

Conventions in 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 C notation: `0x3EFA` instead of Pascal notation (`$3EFA`).

In this manual, the term **Toolbox routine** means any routine described in *Inside Macintosh*. The Macintosh ROM actually consists of two different 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, interrupts, device management, etc. The Toolbox deals with high-level aspects like the drawing environment, the window mechanism, menus, dialogs, etc.

What You Should Know

This manual assumes you already know, or are at least learning, how to program in C. If you're just getting started in C, THINK C 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 official reference that describes the more than 1,000 Macintosh Toolbox routines. The Toolbox is the set of operating system and user interface routines that make a Macintosh a Macintosh. It's beyond the scope of this manual to show you how the different parts of the Toolbox work together.

Learning C

As the popularity of C grows, more and more introductory level books appear on the shelves. Some books assume that you're just learning how to program, and others assume that you already know how to program in another language. Some books spend time telling you how to use the development environment: the editor, the linker, the make facility. These things are done very differently in THINK C, so when you choose a book, choose one that doesn't dwell too much on these aspects of programming.

If you're learning C from a book, or if you're using THINK C to do course work, be sure to do the first tutorial. It shows you how to set things up to write and run C programs that use the standard C libraries.

The standard references for the C programming language are Kernighan & Ritchie's *The C Programming Language, Second Edition* (Prentice Hall) and Harbison & Steele's *C: A Reference Manual* (Prentice Hall). *The C Programming Language, Second Edition* is an update to *The C Programming Language* that incorporates what was then the latest draft of the ANSI standard. These books assume that you're already an experienced programmer.

Standard C (Microsoft Press) by P. J. Plauger and Jim Brodie is a guide to writing C programs that conform to the ANSI C standard. Both of the authors were officers of the committee that drafted the ANSI standard.

Software Engineering in C (Springer-Verlag) by Peter Darnell and Philip Margolis is an excellent introduction to the C programming language. This book is ideal for new C programmers who have programmed in other languages.

C Traps and Pitfalls (Addison-Wesley) by Andrew Koenig is a good book for intermediate and advanced C programmers. It contains a detailed discussion of common C programming problems.

Numerical Recipes in C (Cambridge University Press) by Press, Flannery, Teukolsky and Vetterling is a detailed technical description of numerical methods with implementation examples in C.

Portability and the C Language (Hayden Books) by Rex Jaeschke is a good book on writing portable C programs for advanced C programmers. It gives you guidelines on how to write programs that can be compiled with different compilers and run on multiple platforms. It points out changes introduced with ANSI C. You'll find it useful if you port code from one platform to another or from an old (pre-ANSI) version of THINK C to THINK C 5.0.

Portable C Software (Prentice Hall) by Mark R. Horton is also about writing portable C programs, but emphasizes porting among C compilers for UNIX and MS-DOS.

The document describing the ANSI standard is *American National Standard for Information Systems—Programming Language—C X3.159-1989.* Cost is $50, plus $5 for shipping and handling. To order a copy, write or call:

American National Standards Institute (ANSI)
Sales Dept.
1430 Broadway
New York, NY 10018
(212) 642-4900

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 Mac Plus, Macintosh SE, and Macintosh 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 (see page 14), technical bookstores and computer stores, and in some general bookstores.

THINK Reference, published 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 in-

cludes customizable bookmarks and templates that let you copy Toolbox calls right into your own program.

The *Macintosh C Programming Primer: Inside the Toolbox Using THINK C, Volumes I and II* (Addison-Wesley) are a good introduction to Macintosh programming for those already familiar with C. Volume I, by Dave Mark and Cartwright Reed, 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. Volume II, by Dave Mark, covers more advanced topics like Color QuickDraw and code resources.

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, CA 95014-6299

(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, CA 94709

1 Welcome

Installing THINK C 5.0

2

Installing THINK C 5.0 takes just a few minutes and some mouse clicks. The installation programs know where to place the files so you don't have to.

This chapter explains how to install THINK C 5.0 on your hard drive. If you're installing THINK C 5.0 for the first time, be sure to read the whole chapter. If you need to restore a few things from the original disks, read "What's in the Archives" on page 22.

Note

You must have a hard disk to use THINK C 5.0.

Contents

Summary

This chapter tells you how to set up THINK C on your hard disk. This setup ensures that THINK C 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 C 5.0 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 C 5.0` application and the `THINK C Debugger 5.0` into the newly created `THINK C 5.0 Folder`. If you want to install only some parts of THINK C 5.0, read "What's in the Archives" on page 22 to find out which archives you need to run.

Before You Start...

Before you install THINK C 5.0, you'll want to take care of these steps:

- If there is a file named `READ ME` on the disk `THINK C 1`, read it before proceeding. It contains information that didn't make it into the THINK C manuals. It's a text file that you can read with any word processor, including TeachText.
- Make a copy of you THINK C 5.0 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 five megabytes (5Mb) of disk space free. If you don't have enough room, either clean up your hard disk, or read "What's in the Archives" on page 22 and install only the parts of THINK C 5.0 that you need.

Note

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 C 5.0.

Instructions

Here's how to install THINK C 5.0:

1. In the Finder, create a new folder and name it `Development`.
2. Insert the disk `THINK C 2`, and double-click on `Headers & Libs.sea`.
3. A standard file dialog, like Figure 2-1, appears. Move to your `Development` folder and click Extract.

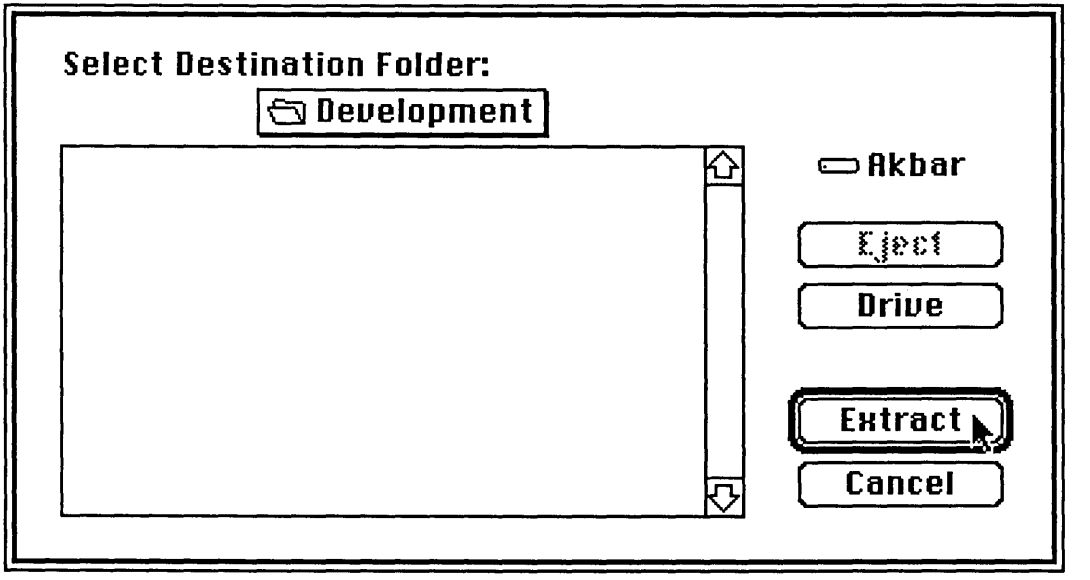


Figure 2-1 Placing files in your Development folder

4. The archive decompresses its files and places them on your hard disk. It displays its progress in the dialog in Figure 2-1. The archive quits when it's done.
5. Repeat steps 2–4 for the each of the other archives on `THINK C 2`, `THINK C 3`, and `THINK C 4`.
6. Insert `THINK C 1`, and move `THINK C 5.0` and `THINK C Debugger 5.0` into the `THINK C 5.0 Folder` in your `Development` folder.

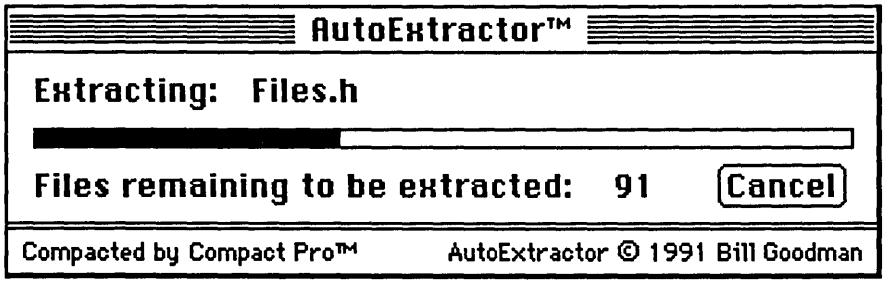


Figure 2-2 Watching the archive install your files

When you're done, the `Development` folder contains these folders, as shown in Figure 2-1:

- `THINK C 5.0 Folder`
- `THINK C 5.0 Utilities`
- `THINK C 5.0 Demos`
- `TCL 1.1 Demos`

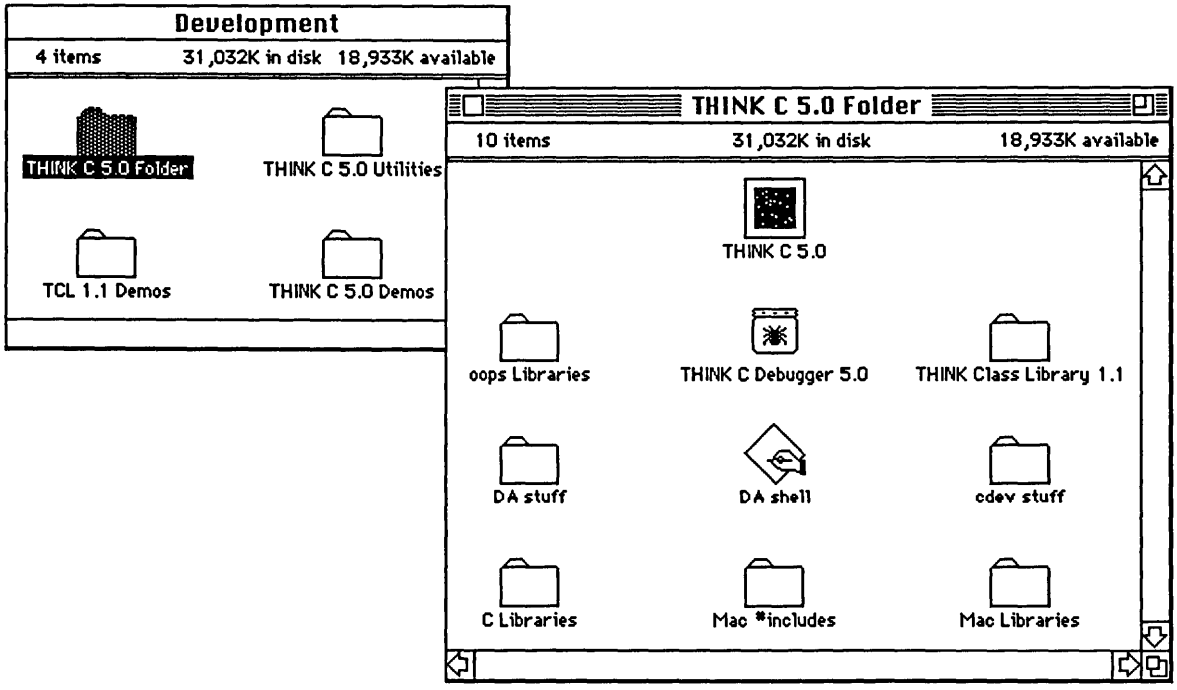


Figure 2-3 The THINK C 5.0 Folder

What's in the Archives

You don't need to install everything included in your THINK C 5.0 package. This table describes which archives you should run to get what you need.

If you want the...	Run these archives...
Basic THINK C system	`Headers & Libs.sea` on `THINK C 2` and copy `THINK C 5.0` and `THINK C Debugger 5.0` into your `THINK C 5.0 Folder` from `THINK C 1`.
THINK Class Library	`THINK Class Library 1.1.sea` and `TCL 1.1 Demos.sea` on `THINK C 3`. (Even if you've used the THINK Class Library before, you'll need the demos for examples of the new features.)
cdev and DA support	`DA/cdev Tools.sea` on `THINK C 2`.
Tutorials and example programs	`THINK C 5.0 Demos.sea` on `THINK C 2`.
Resource utilities, like ResEdit and SARez	`Resource Utilities.sea` on `THINK C 4`.
Other utility programs, like oConv	`THINK C Utilities.sea` on `THINK C 4`.

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

Headers & Libs.sea

This archive places these folders in the `THINK C 5.0 Folder`:

- `oops Libraries`
- `C Libraries` folder, which includes the `ANSI`, `ANSI-A4`, `ANSI-small`, and `unix` libraries
- `Mac #includes` folder, which includes `MacHeaders` and `Mac #includes.c`
- `Mac Libraries` folder, which includes `MacTraps` and `MacTraps2`

DA/cdev Tools.sea

This archive places these files in the THINK C 5.0 Folder:

- cdev stuff folder
- DA stuff folder
- DA shell application

THINK C 5.0 Demos.sea

This archive places these folders in the THINK C 5.0 Demos folder:

- Hex Dump DA folder
- MiniEdit Folder
- Sample cdev folder
- Bullseye Folder
- OOP Demos folder, which includes the Object Bullseye folder and LearnOOP 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 Demos.sea

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

- NewClassDemo Folder
- Starter Folder
- TinyEdit Folder
- Art Class Folder

Resource Utilities.sea

This archive places these files in the THINK C 5.0 Utilities folder:

- ResEdit 2.1
- Rez Utilities folder, which includes SARez and SADeRez

THINK C Utilities.sea

This archive places these files in the THINK C 5.0 Utilities folder:

- Compare
- oConv
- Prototype Helper

About the Self-Extracting Archives

The self-extracting archives included with THINK C 5.0 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 C

Learning THINK C

Part Two

Tutorial: Hello World

3

Hello World shows you how to put together an application with THINK C. The idea here is not to write a fancy program, but to show you how to build an application in THINK C. The program writes the words "hello world" in a window on the screen.

Before you begin

Be sure you followed the instructions in Chapter 2, "Installing THINK C 5.0," to install THINK C 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 user's manual that came with your Macintosh.

Contents

Creating the Project

First, you need to create a folder for your project in the `Development` folder. This is the folder you'll use for all your development work.

Create a folder in the `Development` folder and call it `Hello Folder`. Do this now, before you start THINK C. You can use a different name if you like, but remember that your dialog boxes won't match the pictures in this chapter.

Generally speaking, you'll have a folder for each project you work on. The folder should contain your source files, your #include files, and the application's resource file.

When you've created the `Hello Folder`, open the `THINK C 5.0 Folder` (the one that contains the THINK C application) and double click on the THINK C icon.

You'll see a dialog box that asks you to open a project.

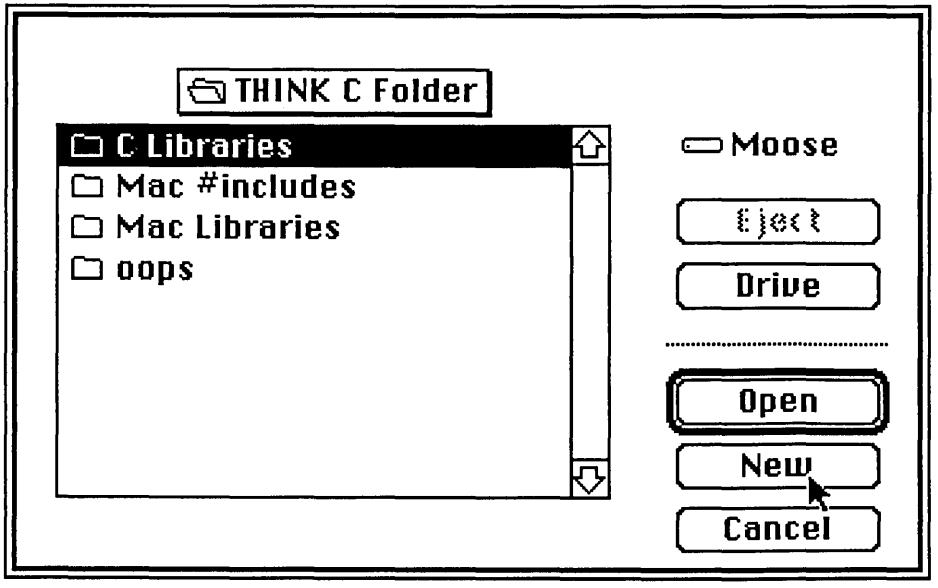


Figure 3-1 Opening a project

Since you're creating a new project, click on the New button. You'll see another dialog box, one that lets you create projects.

Move back to the `Hello Folder` you just created.

Warning
It's very important that you move to the `Hello Folder`.

Name the project `hello project`, and click on the Create button.

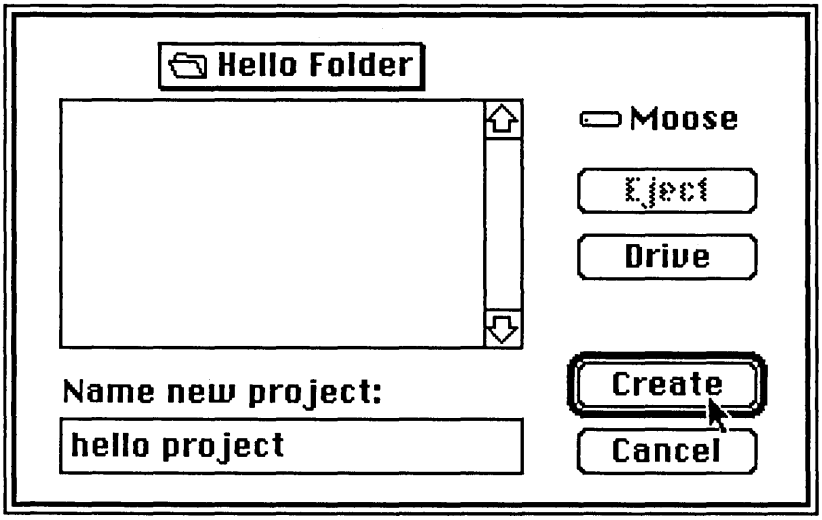


Figure 3-2 Creating your new project

THINK C creates a new project document on disk and displays a project window:

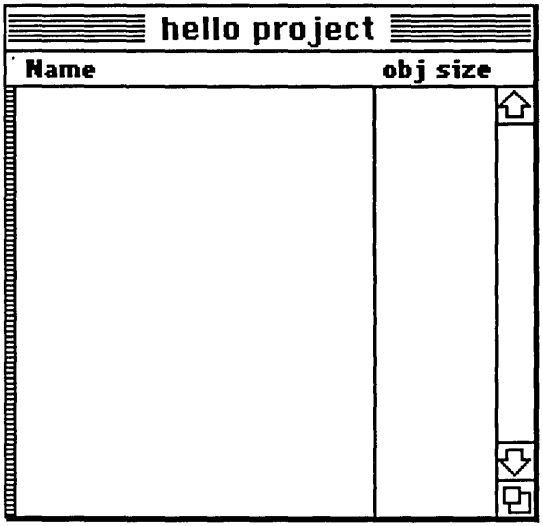


Figure 3-3 The project window for a new project file

The **Name** column shows the names of all the source files and libraries in your project, and the **obj size** column displays their sizes in bytes.

Creating the Source File

Now you're ready to create your source file. Choose **New** from the **File** menu to bring up an empty editing window.

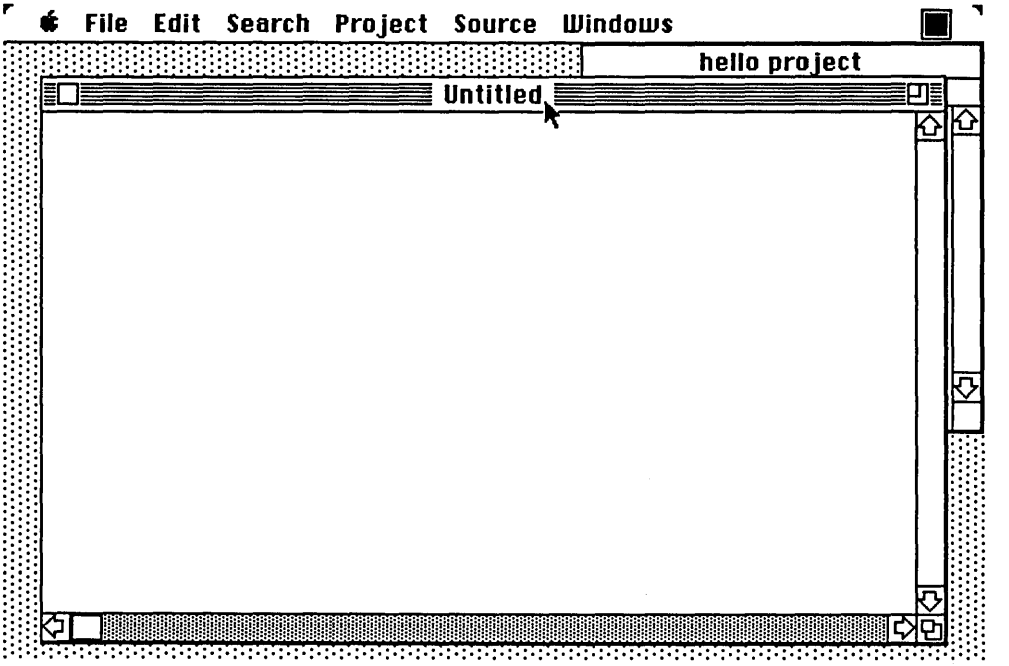


Figure 3-4 An empty editing window

Type this program into the editor window (you don't need to type in the comments if you're in a hurry):

```
/*****
 * hello.c
 *
 * The hello world program for THINK C
 *
 *****/

#include <stdio.h>

main()
{
      printf("hello world\n");
}
```

The THINK C text editor works like most other text editors on the Macintosh. You can drag to select a range of text or double click to select words. You can also triple click to select an entire line. If you have a keyboard with arrow keys, you can use them to move around your file.

For more information about the THINK C text editor, see Chapter 8, "The Editor."

The text editor does not wrap text back to the left edge of the window when you type past the right edge of the window. Use the horizontal scroll bar at the bottom of the window to see any text that goes past the right edge.

When you've typed in the program, select **Save As...** from the **File** menu to save it. You'll get a dialog box like the one below. Name the file `hello.c`, and click on the Save button.

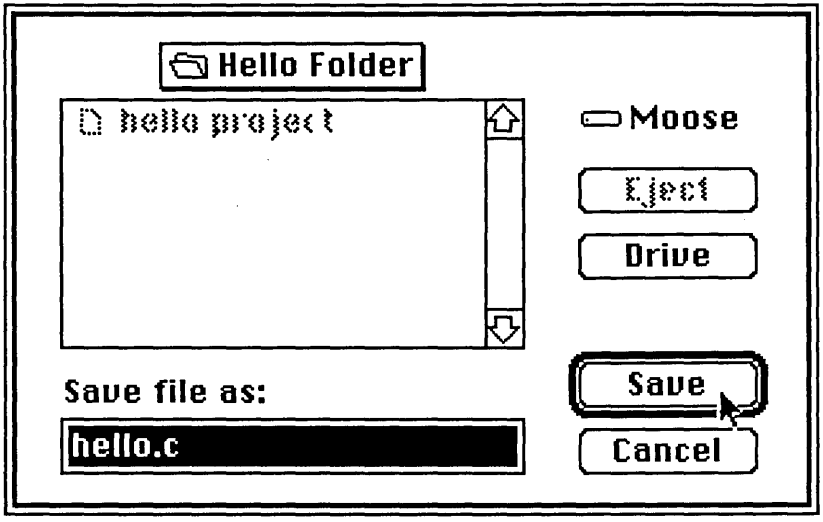


Figure 3-5 Saving a new file

THINK C will compile only files that end in `.c`, but you can edit any text file with the THINK C editor.

Compiling the Source File

*Using the **Compile** command in THINK C is like using the `cc` command in UNIX, except that THINK C adds the object code to your project, instead of creating a separate object file.*

Now you're ready to compile your source file. Select **Compile** from the **Source** menu. THINK C displays a dialog box that shows how many lines have been compiled.

When THINK C compiles a source file, it adds its name and size to the project window. Your project window should now look like this:

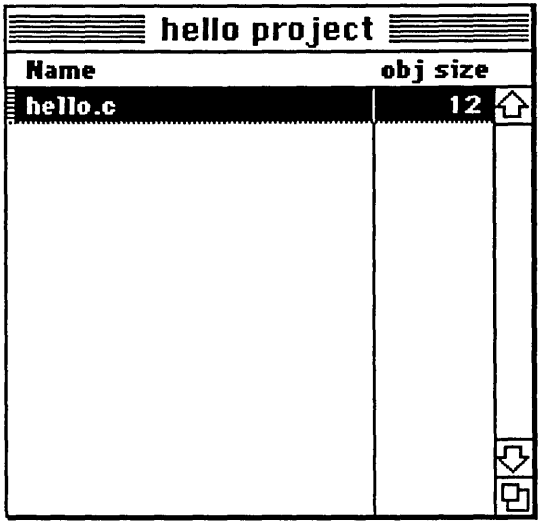


Figure 3-6 The project window with one file

THINK C keeps all the object code for your source files in the project document.

Did you get an error?

If you made a mistake typing the program, THINK C will display an error message in a dialog box. The message may say "syntax error." In this small program, about the only syntax error you can make is forgetting a quote, a parenthesis, or a semicolon.

Click anywhere in the dialog box to get rid of it. THINK C puts the insertion point in the line with the error. Look over your program to make sure everything is correct. Then select **Compile** from the **Source** menu.

If you get an error message that says "can't open file" like this one:

Figure 3-7 An error message

it means that THINK C wasn't able to find the #include file `stdio.h`. THINK C can't find the #include files if you didn't move the `Libraries` folder into the `THINK C Folder`. The best thing to do now is to start over from the beginning.

Quit THINK C and move the `Hello Folder` to the Trash. Then look in Chapter 2, "Installing THINK C 5.0," to make sure you installed THINK C correctly. Once you're sure everything is OK, start again from the beginning of this chapter.

Warning

Throw the `Hello Folder` into the Trash only if you're starting all over. If you didn't get the "can't open file" error message, go on.

Adding the Library

If you tried to run your program now, you'd get linking errors because the project doesn't know where the `printf()` function is defined.

To learn more about the routines in the `ANSI` *library, see the Standard Libraries Reference.*

Next, you need to add the `ANSI` library to your project. It defines the routine `printf()` and all the standard C library routines. To add the `ANSI` library,

choose **Add...** from the **Source** menu. You'll see a dialog like the one in Figure 3-8.

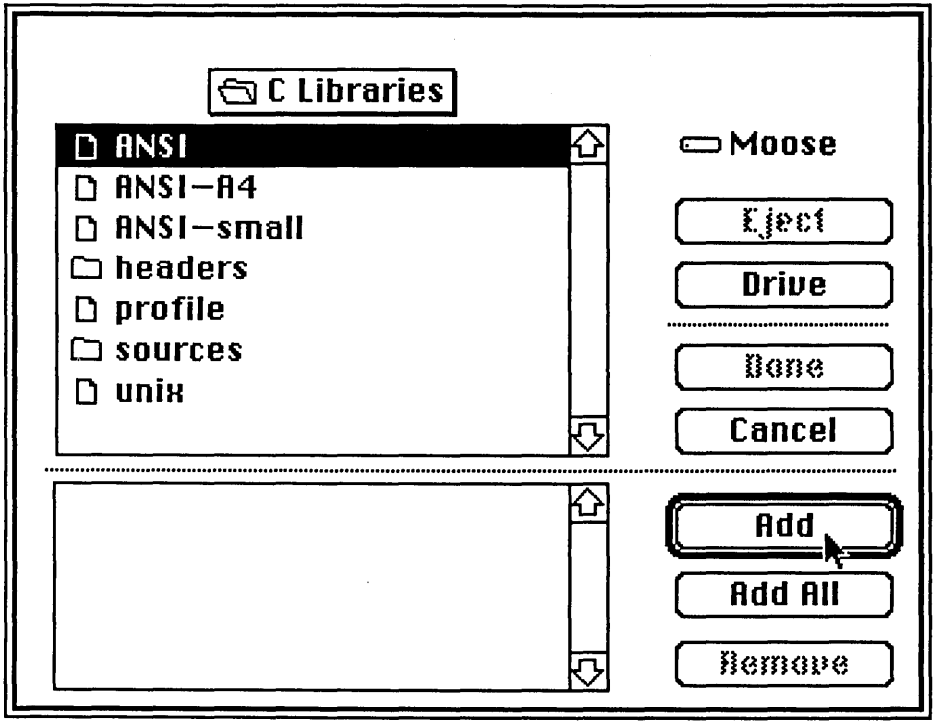


Figure 3-8 The Add... dialog.

The top list displays the source files and libraries in the folder you're currently in. The bottom list shows the files THINK C will add to your project when you click Done. The Add and Add All buttons move files into the bottom list. The Remove button removes files from the bottom list.

Move to the folder called `C Libraries`. This folder contains all the libraries for ANSI and UNIX compatibility, including the `ANSI` library. Select `ANSI`, and click on the Add button. The file moves to the bottom list. Now click on the Done button

THINK C adds the name ANSI to the project window. Your project window should look like this:

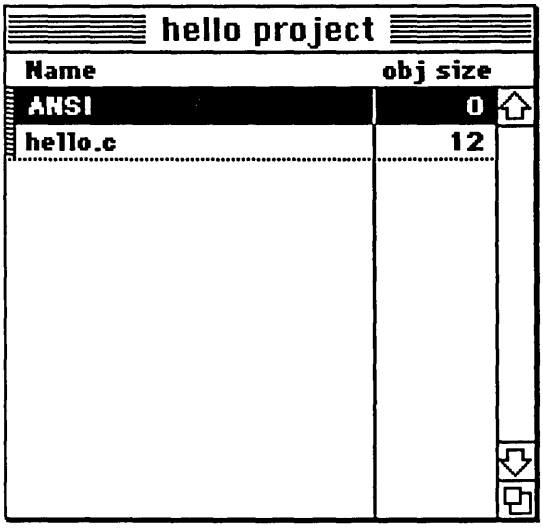


Figure 3-9 The hello project window with both its files

The object size for the ANSI library is zero since THINK doesn't load a library's code until you need it. This lets you add several libraries without waiting for them to load.

THINK C loads a library automatically when you run the project. Another way to load a library is to click on its name in the project window, and then choose **Load Library** from the **Source** menu. For this example, let THINK C load it for you.

Running the Project

*Using the **Run** command in THINK C is like doing all this in most other compilers: compiling your project's files, linking it, and running it. However, **Run** doesn't save a copy of your application to disk before running it. To save it, use **Build Application...**, described below.*

Everything is all set to run the project. The source file is in the project window along with the libraries you'll be using. Now select **Run** from the **Project** menu.

THINK C notices that the library needs to be loaded, so it puts up a dialog box asking you if you want to bring the project up to date:

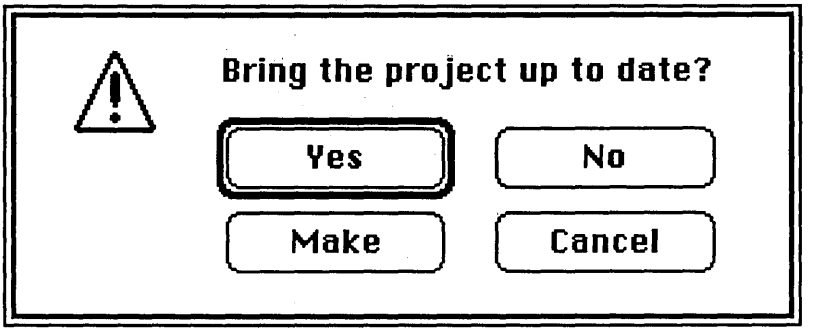


Figure 3-10 A dialog box asking if you want to update your project

Click on the Yes button. THINK C goes to disk to load the code for the `ANSI` library. It may take THINK C a little time to load it. Once it's loaded into the project, though, THINK C doesn't need to load it again.

Any time you choose to run your project and THINK C notices that you've made changes (added libraries or source files or edited source files) it will ask you if you want to update the project. If you say yes, it will compile the new or changed files and load the new libraries.

For more information on console windows, see the Standard Libraries Reference, *Chapter 3, "Using Console Windows."*

Since this program uses the `ANSI` library, everything `printf()` writes goes to a console window. A console window a Macintosh window that behaves like a simple display terminal, the kind of terminal that many MS-DOS and

UNIX computers use. You'll see the "hello world" string at the bottom of this window.

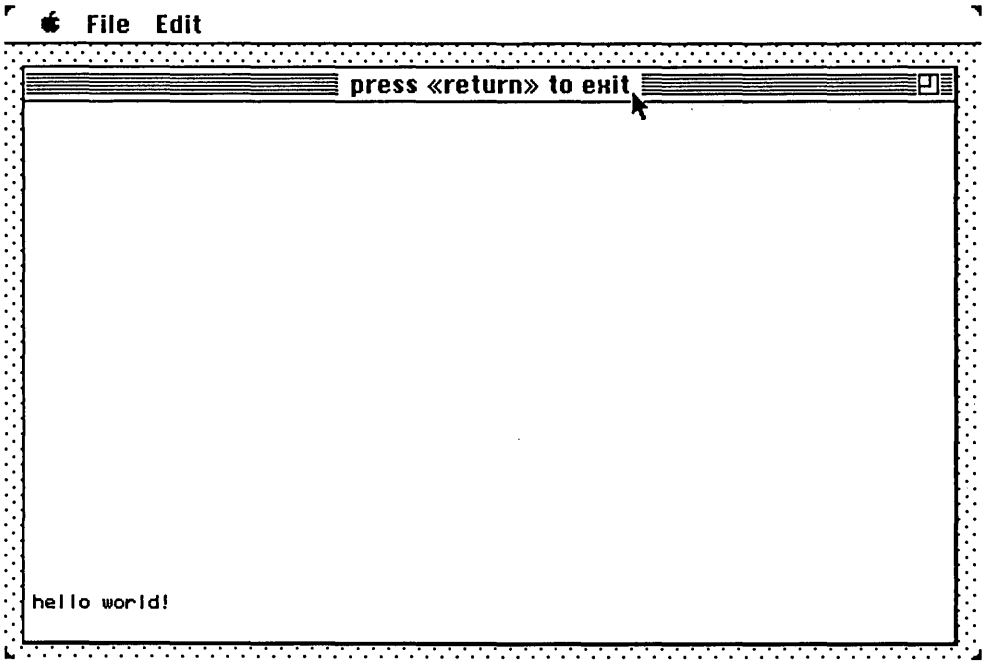


Figure 3-11 Your program running in a console window

To exit the program, press the Return key or choose **Quit** from the **File** menu.

Creating the Application

As you develop a large application, you make changes to your source files. Each time you run your project, THINK C will recompile only those files that have changed. When you're ready to turn your project into a stand-alone double-clickable application, select **Build Application...** from the **Project** menu.

You'll see a dialog box asking you to name your application. Name it `hello appl`. Leave the Smart Link box checked. This option tells THINK C to make your application as small as possible

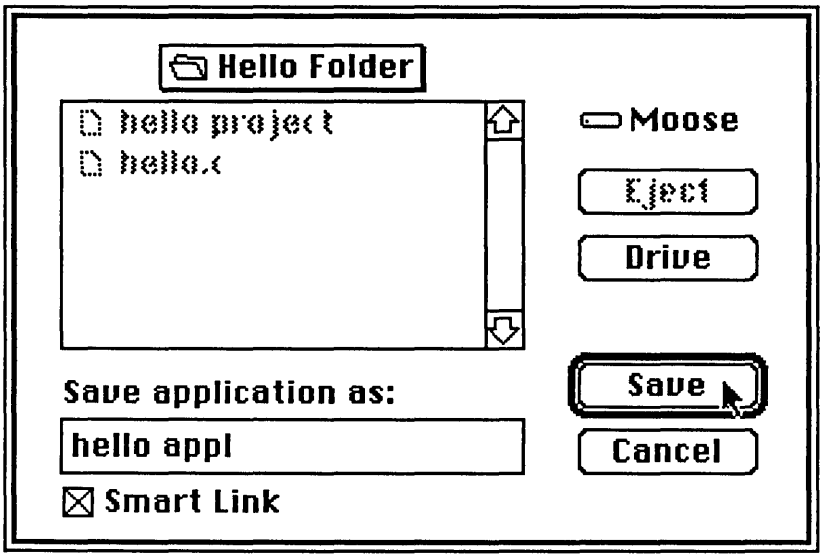


Figure 3-12 Building your application

THINK C puts up a dialog box telling you it's linking your application. When it's finished, the application will be in the folder you chose.

If you're running without MultiFinder or System Software 7.0, quit THINK C to run your application. Use the **Quit** command in the **File** menu. If you're using MultiFinder or System Software 7.0, you don't need to quit first. Just bring up the window with the folder your application is in. Double click on your application and watch it run. That's all there is to it.

Where to Go Next

The tutorial in the next chapter is a more elaborate example of building an application with THINK C. It describes how THINK C reports errors when you compile and link, and it will show you some advanced features of the THINK C editor.

If you would rather explore on your own, read the chapters of the "Using THINK C" section that interest you. Or if you want to learn how to use THINK C's source level debugger, now, go to Chapter 5, "Tutorial: Bullseye," and follow the tutorial there.

Tutorial: MiniEdit 4

MiniEdit shows you how to use the more advanced features of THINK C. You'll build a small text editor based on the sample application described in Chapter 1 of *Inside Macintosh I*. One of the source files has a small, intentional bug to show you how THINK C makes it easy to fix mistakes.

You'll learn how to create a project, how to fix mistakes, how to run a project, how to build an application, and how to use a resource file.

Before you begin

If you didn't follow the "hello world" example in the last chapter, read it now to get an idea of how THINK C works in general.

Make sure the `MiniEdit Folder` is on your hard disk, since it contains all the files you need to follow this example. If you followed the directions in Chapter 2, "Installing THINK C 5.0," it should be inside the `THINK C 5.0 Demos` folder inside your `Development` folder. To install it, run the self-extracting archive `THINK C 5.0 Demos.sea` on `THINK C 2`, and select your `Development` folder as the destination folder.

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 documentation that came with your Macintosh.

You will need to know which folder contains the file `MacTraps`.

Contents

Creating the Project

Make sure the `MiniEdit Folder` is on your hard disk, since it contains all the files you need to follow this example. If you followed the directions in Chapter 2, "Installing THINK C 5.0," it should be inside the `THINK C 5.0 Demos` folder inside your `Development` folder. To install it, run the self-extracting archive `THINK C 5.0 Demos.sea` on `THINK C 2`, and select your `Development` folder as the destination folder.

Generally speaking, you'll have a folder for each project you work on. The folder should contain your source files, your header files, and the application's resource file.

Open the `THINK C 5.0 folder`, and double click on the THINK C icon to begin. You'll see a dialog box that asks you to open a project, as in Figure 4-1.

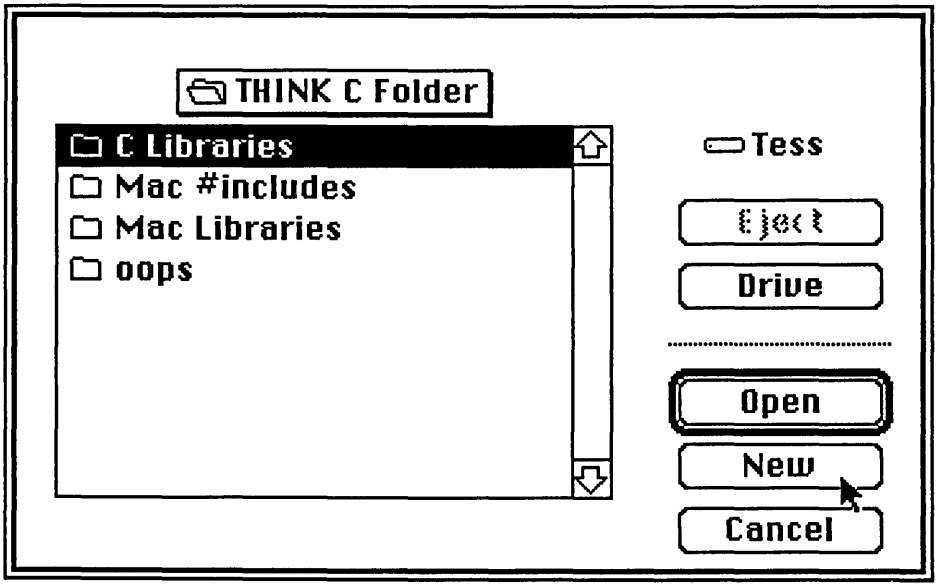


Figure 4-1 Opening a project

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

When you get the next dialog box, move to the `MiniEdit Folder`, and name your project `MiniEdit.π`. Project names don't have to end in `.π`,

though it's a good idea. For this example, it is important that you name your project `MiniEdit.π`, as in Figure 4-2. (To make a π, type Option-p.)

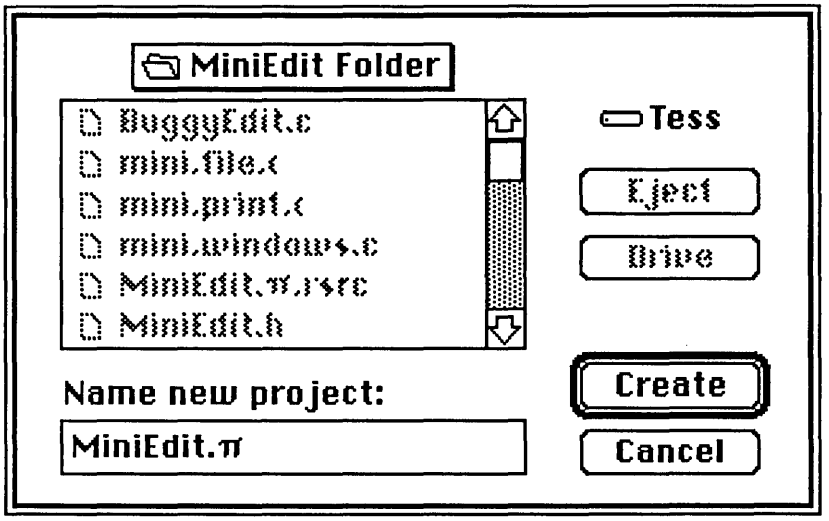


Figure 4-2 Creating a new project

Warning
It's very important to move back to the `MiniEdit Folder`.

Click on the Create button. THINK C creates a project document on disk, and displays an empty project window, like Figure 4-3.

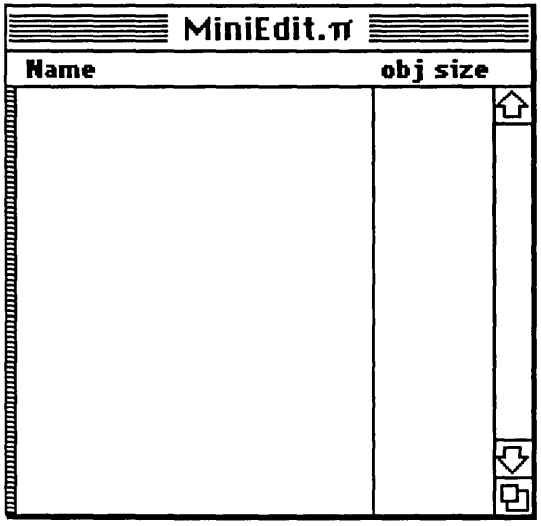


Figure 4-3 The project window for a new project file

Adding the Source Files and Libraries

Now you're ready to add the source files and the `MacTraps` library to your project. All the source files for `MiniEdit.π` are in the `MiniEdit Folder`.

Select **Add...** from the **Source** menu. You'll see the dialog in Figure 4-4.

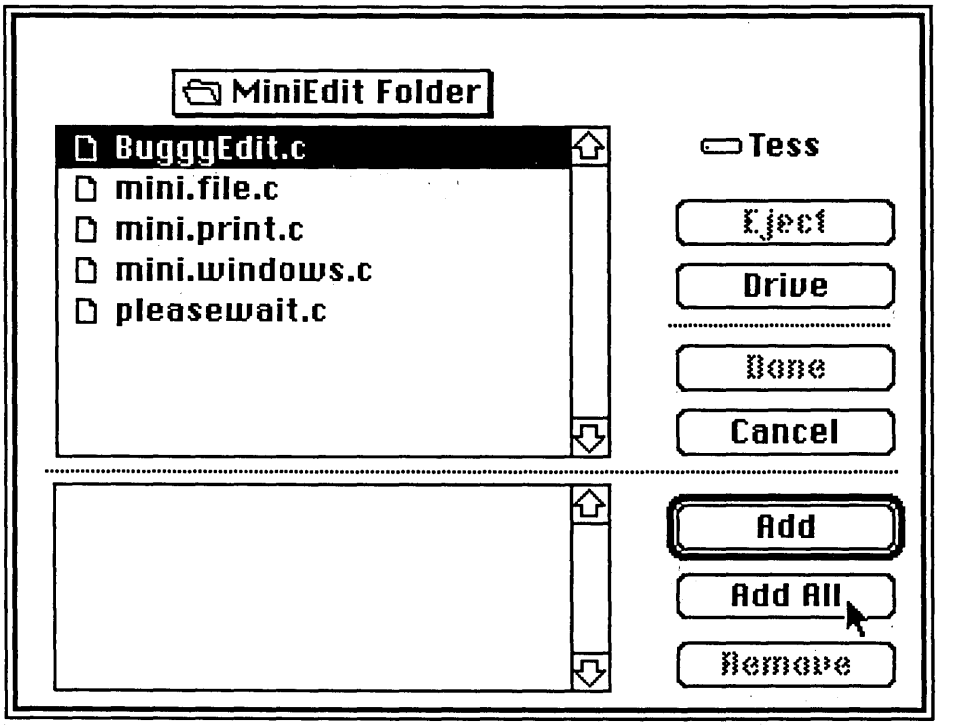


Figure 4-4 The **Add...** dialog.

The top list displays the source files and libraries in the folder you're currently in. The bottom list shows the files THINK C will add to your project when you click Done. The Add and Add All buttons move files into the bottom list. The Remove button removes files from the bottom list.

Click on the Add All button. THINK C moves all the source files in the `MiniEdit Folder` into the bottom list. The dialog should look like Figure 4-5.

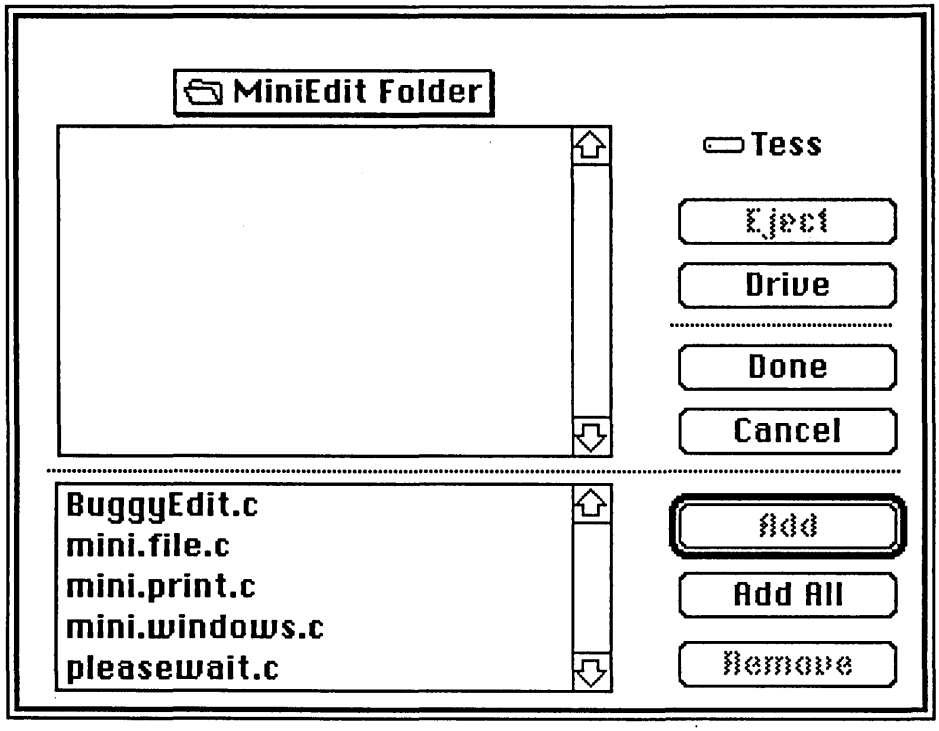


Figure 4-5 Adding the MiniEdit source files.

Don't click on the Done button yet. You still need to add a library. Move to the `Mac Libraries` folder, and add the `MacTraps` library by selecting it and clicking the Add button, as in Figure 4-6.

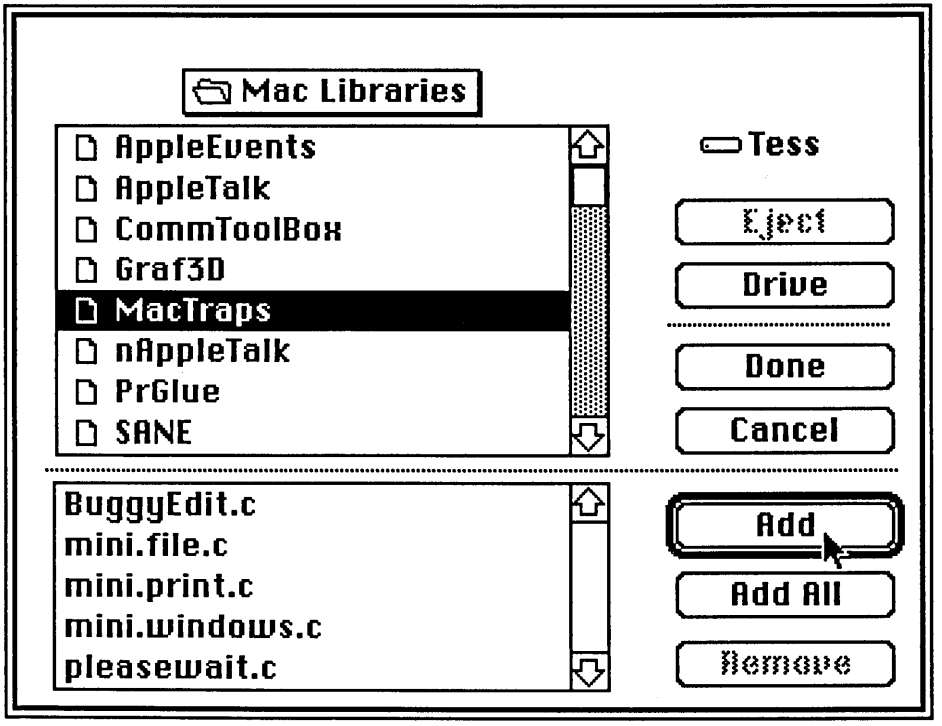


Figure 4-6 Adding MacTraps to your project

All the files you need for this project are now in the project window. Click on the Cancel button now. Your project window should look like Figure 4-7.

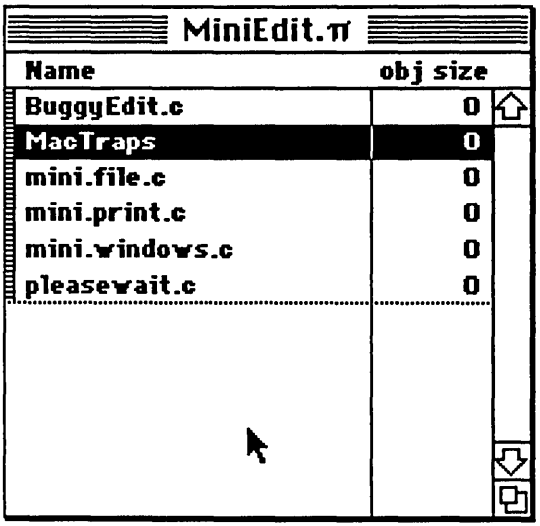


Figure 4-7 The MiniEdit.π project window with all its files

The **obj size** column displays the object size in bytes for each file. The sizes are all zero because you haven't compiled any files or loaded the `MacTraps` library.

Compiling and Running the Project

Before you can run your project, you need to compile the source files and load the `MacTraps` library. You can use the **Compile** and **Load Library** commands in the **Source** menu, or you can let THINK C take care of everything for you.

THINK C uses the project document to keep track of which files need to be compiled, so you can go ahead and run your project. Choose the **Run** command from the **Project** menu.

None of the files in the project have been compiled, so THINK C asks you if you want to bring the project up to date, like Figure 4-8.

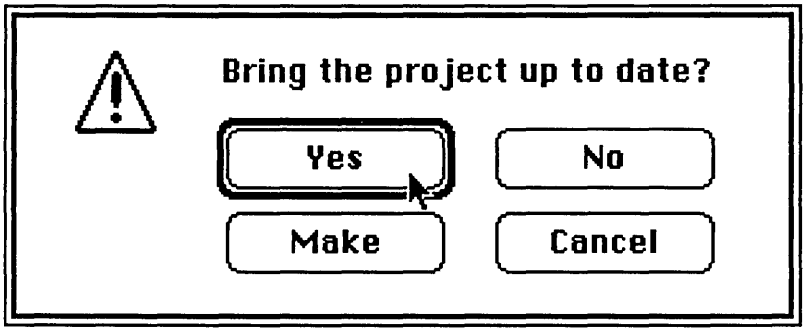


Figure 4-8 A dialog box asking if you want to update your project

Click on the Yes button.

THINK C starts compiling the first file in the project. It displays a dialog box that shows how many lines have been compiled. (THINK C adds the number of lines in #includes files in the line count.)

In this example, THINK C doesn't get very far because `BuggyEdit.c` has a small intentional bug.

Fixing a Bug

When THINK C finds an error in your source file, it opens the file that contains the error and displays an error message in a dialog box. THINK C se-

lects the line the line that contains the error. In Figure 4-9, THINK C complains that a variable hasn't been defined.

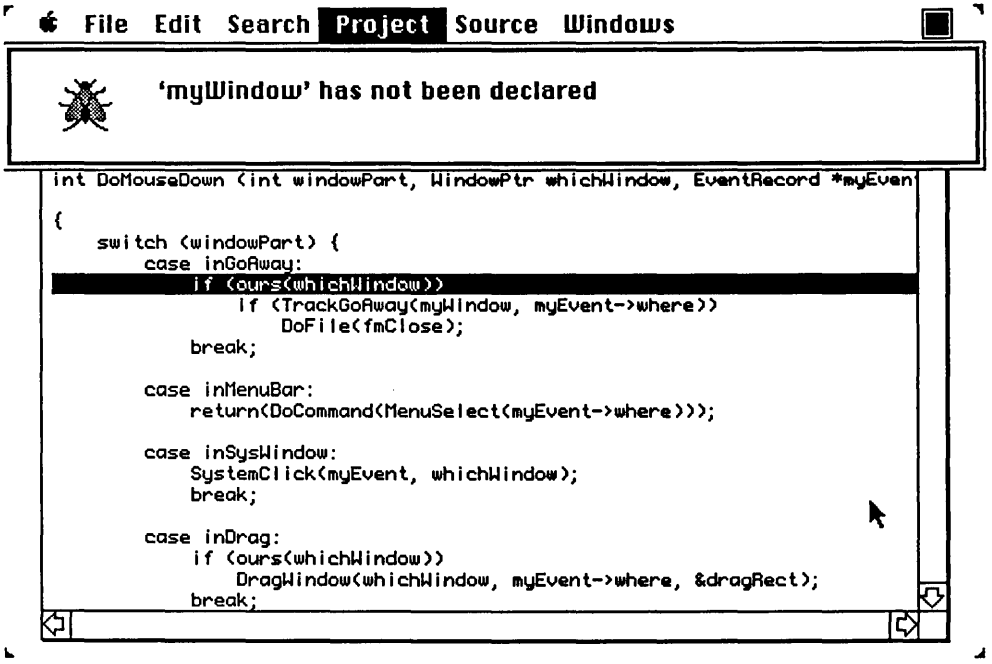


Figure 4-9 THINK C displays an error message.

To get rid of the dialog box, click anywhere in it or press the Return or Enter key.

Scroll toward the beginning of the file, and you'll see that the declaration for `myWindow` is commented out, like Figure 4-10.

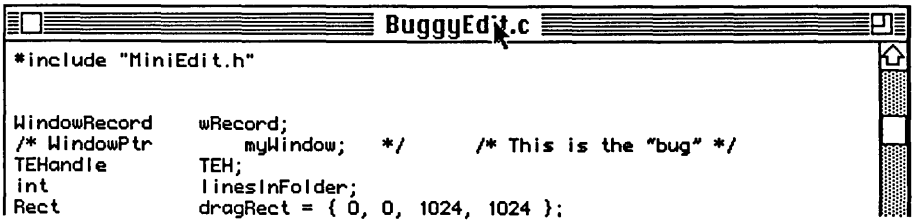


Figure 4-10 A declaration that was commented out

Remove the comments surrounding the declaration of `myWindow`.

Now, compile the file `BuggyEdit.c`. Choose the **Compile** command from the **Source** menu. THINK C will compile the source file without errors this time. Note that you don't have to save a file to recompile it.

Before you run the project again, save the changes you've made to `BuggyEdit.c`. Since the file no longer contains a bug, save it with a different name. Choose the **Save As...** command from the **File** menu, and save the corrected file as `MiniEdit.c`, as in Figure 4-11. (Make sure you're in the `MiniEdit Folder`.)

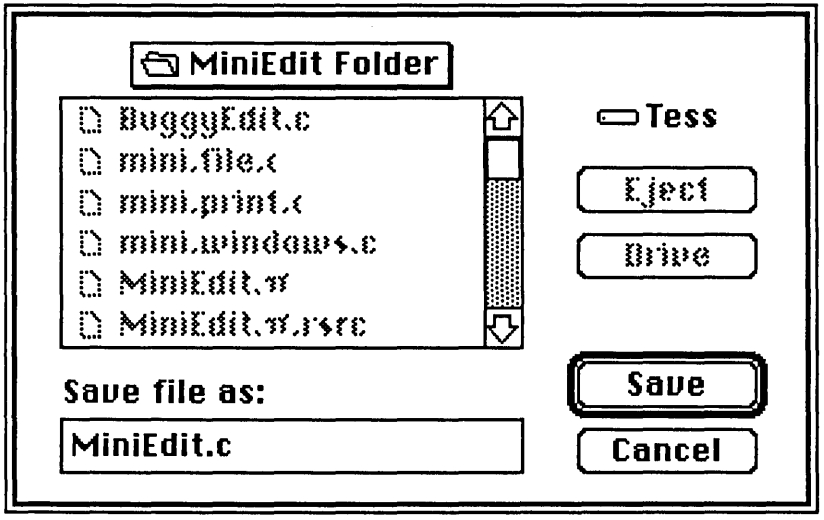


Figure 4-11 Saving a file with a new name

Now click on the project window. When you use the **Save As...** command on a file that is already in the project, THINK C changes the file's name in the

To save a file with a different name without affecting the project, use the Save A Copy As... command.

project window as well. The file's object code is now associated with the new name, as in Figure 4-12.

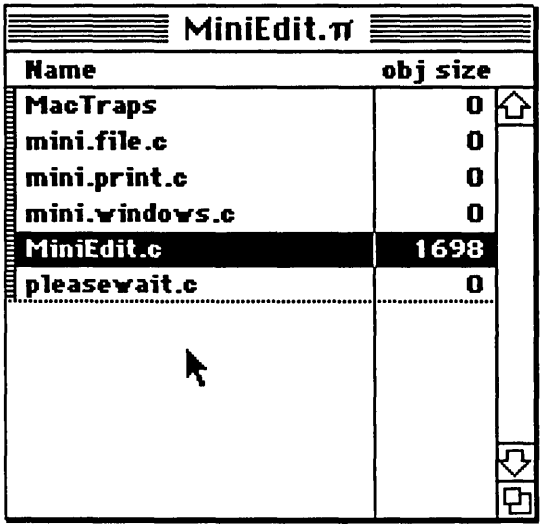


Figure 4-12 The MiniEdit.π project window with the new file name

Now that you've fixed the bug, you can try running the project again.

Running the Project Again

Choose **Run** from the **Project** menu. When THINK C asks you if you want to bring the project up to date, click on the Yes button.

THINK C loads the `MacTraps` library, then it compiles all the files in the project. Since you already compiled `MiniEdit.c`, THINK C doesn't recompile it.

Once THINK C compiles the whole project, it launches it as if you had opened it from the Finder, like Figure 4-13.

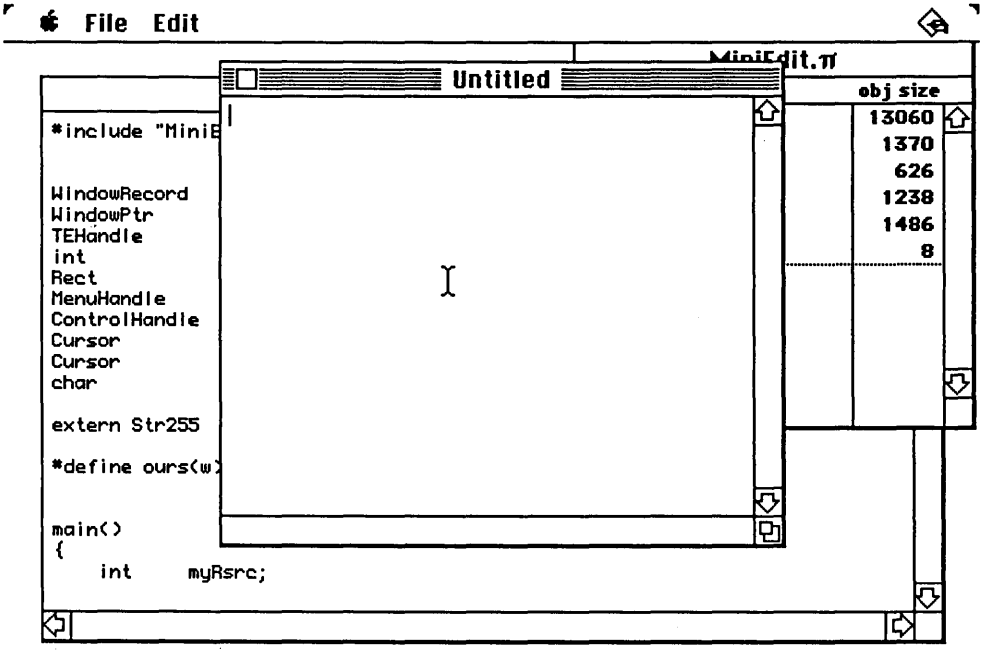


Figure 4-13 Running MiniEdit with the THINK C environment

If you're using MultiFinder or System Software 7, THINK C launches your project in its own partition, so you can shift from your project back to THINK C. If you're not using MultiFinder or System Software 7, THINK C launches your project as if you had started it from the Finder. When you quit running your project, THINK C starts up again automatically.

Play with the MiniEdit application for a while if you like. You might want to make some changes. When you're satisfied with how the project runs, you're ready to turn it into a double-clickable application.

Building the Application

Turning your THINK C project into an application is easy. Choose the **Set Project Type...** command from the **Project** menu. You'll see the dialog box in Figure 4-14:

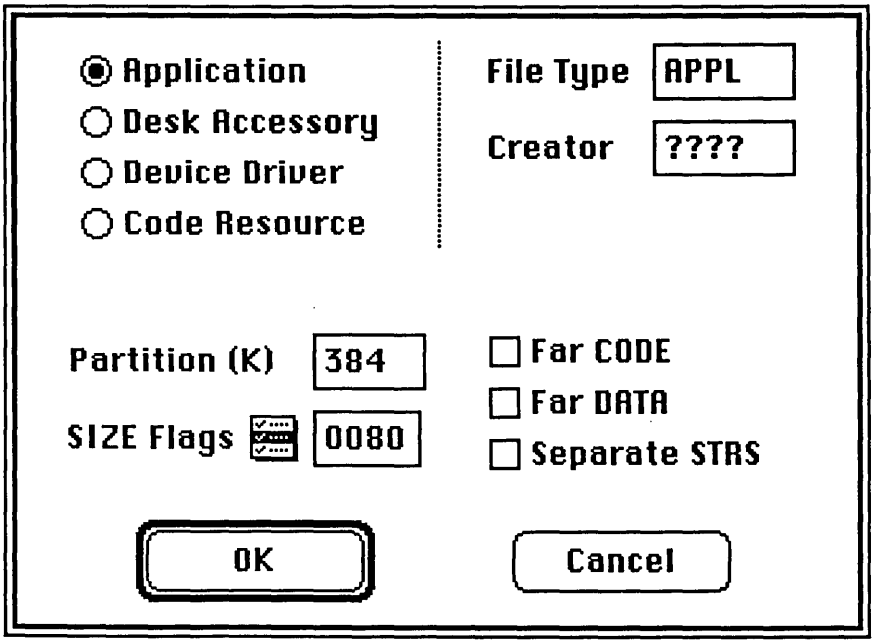


Figure 4-14 The Set Project Type... dialog

CEM8 doesn't stand for anything. It's just unlikely that any other application on your disk has that signature.

Set the creator to CEM8. This will ensure that your application will have the right icon when you build it.

Set the partition size to 128K. Since MiniEdit is such as small program, it doesn't need the default 384K partition size. The Macintosh uses the partition size to determine how much memory to give to your application.

When the dialog box looks like Figure 4-15, click on the OK button:

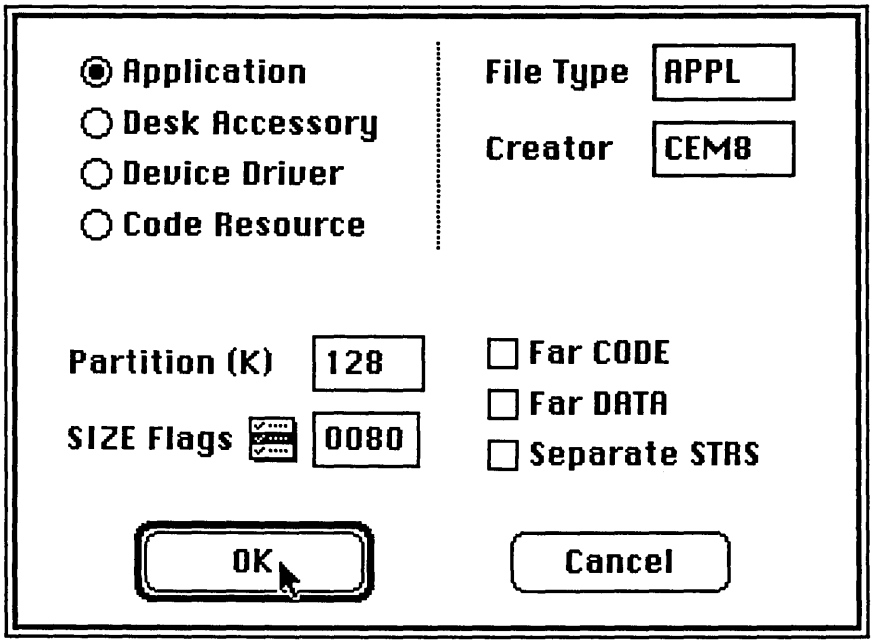


Figure 4-15 Setting the project type for MiniEdit

Choose **Build Application...** from the **Project** menu. You'll see the dialog box in Figure 4-16.

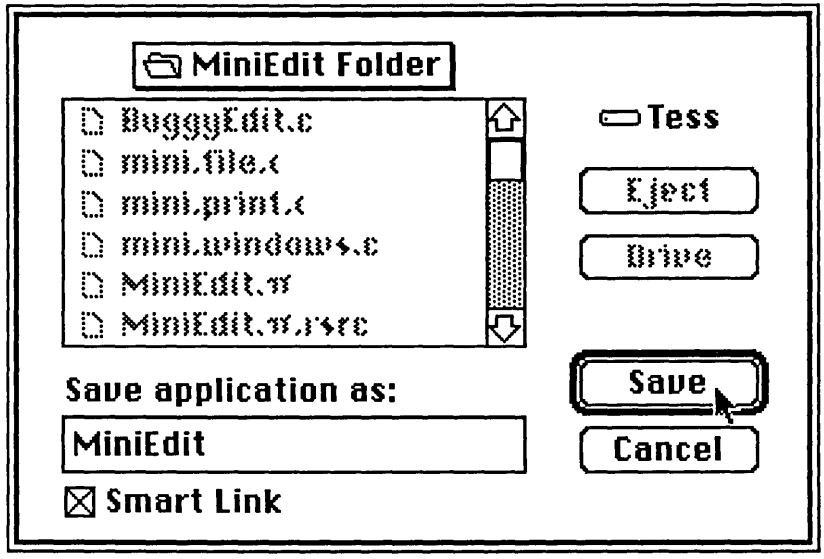


Figure 4-16 Building the MiniEdit application

Name the application `MiniEdit`. Leave the Smart Link box checked. This option tells THINK C to make the application as small as possible.

As it's building the application, THINK C gives you status messages. First it links all the object code. Then it copies the resource file for the project into the application. (The next section tells you how to use a resource file with a project.) When it's finished, you'll have a new application in the `MiniEdit Folder`.

MiniEdit

Figure 4-17 The MiniEdit icon

Using a Resource File

The `MiniEdit Folder` you copied from disk `THINK C 4` contains a file called `MiniEdit.π.rsrc`. This file contains the resources that the MiniEdit project uses.

For more information on SARez, see Chapter 18, "Using SARez." For more information on resource files, see Chapter 17, "Resource Description Files."

When THINK C runs your project, it looks for a file named *projectname*`.rsrc`. (That is, the name of your project plus the characters `.rsrc` appended to it.) This file should contain the resources (menus, alerts, dialogs, etc.) that your project uses.

To create a resource file, you can use Apple's SARez or ResEdit utilities (they're included in your THINK C package). `MiniEdit.π.rsrc` was created with ResEdit, so there's no resource description file for it.

Finishing Up

When you're finished working on a project, you can either close the project or quit THINK C. To close the project, use the **Close Project** command in the **Project** menu. When you close a project, THINK C displays a dialog box that lets you open or create projects.

To quit THINK C, choose **Quit** from the **File** menu.

Where to Go Next

The tutorial in the next chapter shows you how to use THINK C's source level debugger. It shows you how to activate the debugger, how to trace through your code, and how to examine and change the values of your variables.

If you feel comfortable with what you know so far, you might want to start creating your own applications right away. Use the next part of the manual, "Using THINK C", when you need help on a particular topic. You'll probably find Chapter 8, "The Editor," useful now.

Tutorial: Bullseye 5

Bullseye shows you how to use THINK C's source level debugger. Bullseye is a simple application that draws a series of concentric circles in a small window. Its **Width** menu lets you select how wide each of the rings is.

Before you begin

Make sure you're using System Software 7 or running MultiFinder under System 6.0.X on a Macintosh with at least 2Mb of memory.

Make sure the `Bullseye Folder` is on your hard disk, since it contains all the files you need to follow this example. If you followed the directions in Chapter 2, "Installing THINK C 5.0," it should be inside the `THINK C 5.0 Demos` folder inside your `Development` folder. To install it, run the self-extracting archive `THINK C 5.0 Demos.sea` on `THINK C 2`, and select your `Development` folder as the destination folder.

Make sure that the file `THINK C Debugger` is in the same folder as THINK C (the `THINK C 5.0 folder`). This file must be named `THINK C Debugger`.

What you should know

Before you try this example, you should know how THINK C works. You should know how to open a project, how to edit source files, and how to run a project. If you're not familiar with any of these operations, go back and read (or try) the examples in the last two chapters.

Contents

Opening the Bullseye Project

If you're at the Finder, double click on the `Bullseye.π` project in the `Bullseye Folder`. If you're already in THINK C, use the **Open Project...** command in the **Project** menu to open the `Bullseye.π` project.

Bullseye consists of three source files and the `MacTraps` library shown in Figure 5-1.

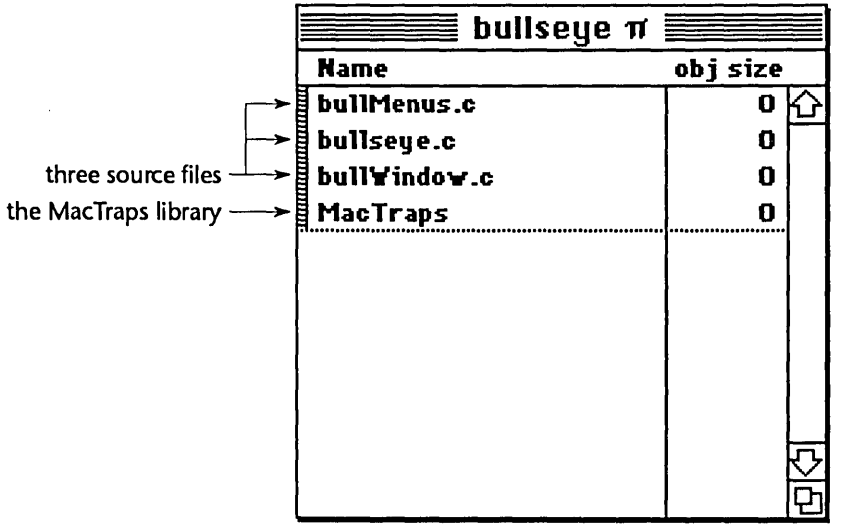


Figure 5-1 The bullseye.π project window

Note that none of the files have been compiled, and that the `MacTraps` library hasn't been loaded.

Turning the Debugger On

THINK C ordinarily runs your project without the debugger. Choose the **Use Debugger** command from the **Project** menu.

When the source debugger is on, THINK C adds a "bug" column in the project window to the left of the Name column.

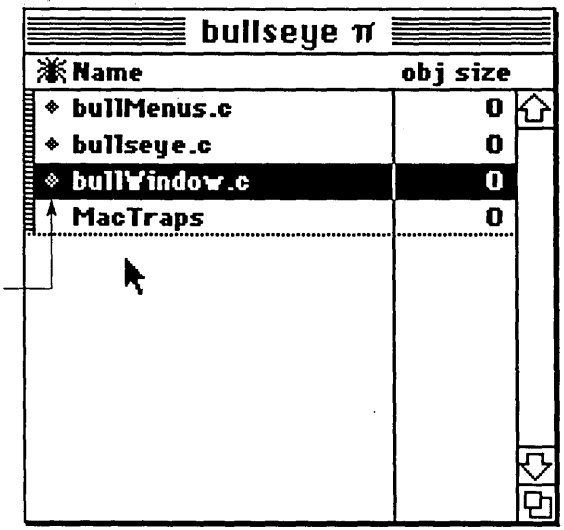


Figure 5-2 The bullseye.π project window with the debugger turned on

Generating the debugging tables

The gray diamonds in the "bug" column let you know that THINK C will generate special debugging tables for a source file. These tables go into your project document along with the source files' object code. THINK C never generates additional code when you run the debugger.

Running the project

Choose **Run** from the **Project** menu to let THINK C compile and load all the files in your project. THINK C will generate debugging tables for all the files as well.

Watching the Program Run

Instead of running your project, THINK C launches the source debugger, which controls the execution of your program. The debugger displays two windows at the bottom of your screen. If you're using two screens, the debugger windows are on the second screen instead.

The window on the left is the Source window. It contains the source text of your program. The window on the right is the Data window. You use this

window to examine and set the values of your variables as you debug your program.

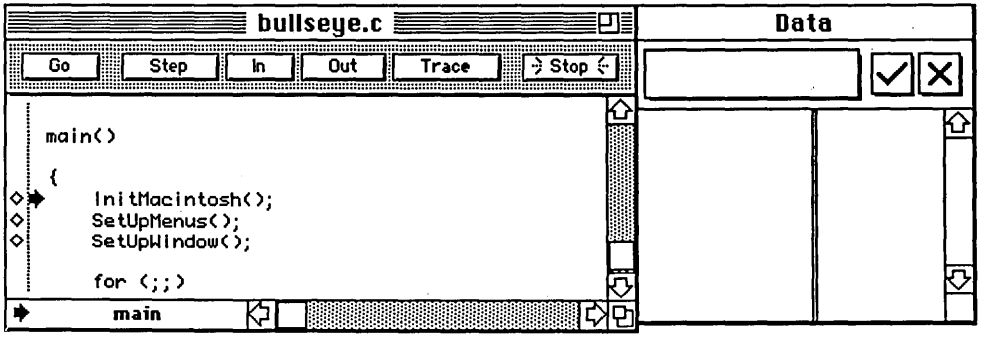


Figure 5-3 The THINK C debugger

The Source window

The Source window shows you the source of your program. The title of the window is the name of the source file you're looking at.

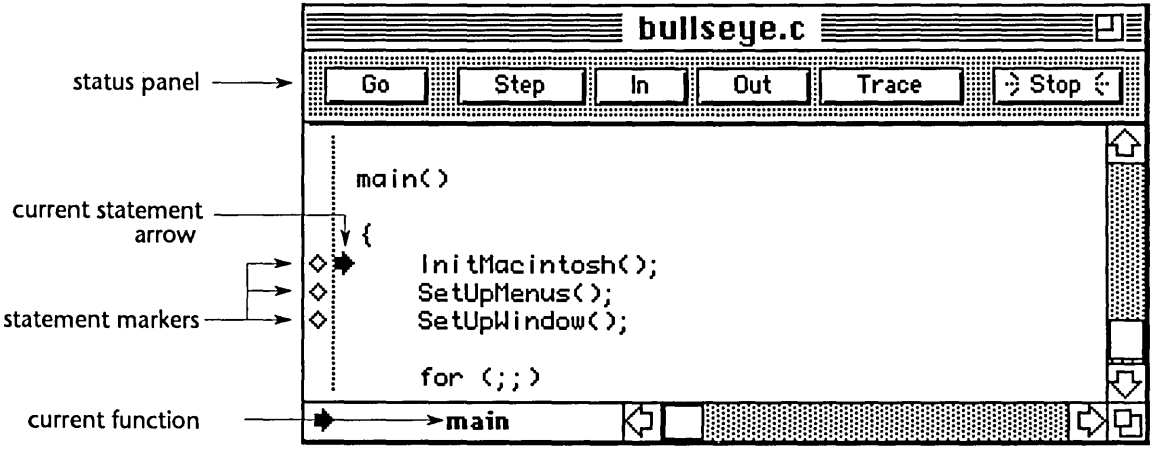


Figure 5-4 The source window of the THINK C debugger

The Source window shows the **source text** of your program. When you start the debugger, this window shows the file that contains the `main()` routine of your application

The top of the Source window has a six button **status panel**. These buttons control the execution of your program. The names of these buttons match

the commands in the debugger's **Debug** menu. Right now, the Stop button is lit to show you that your program is stopped.

The black arrow to the left of the first line of the program is the **current statement arrow**. This indicator shows you the **current statement**, the one that the debugger is about to execute.

The column of diamonds running along the left side of the source text are **statement markers**. Every line of your program that generates code gets a statement marker. Later, you'll use the statement markers to set breakpoints.

The source debugger uses the space at the lower left of the source window for the name of the **current function**. When you click here and hold the mouse button down, the debugger displays a pop-up menu that shows the call chain—the names of the functions that were called to get to the current function.

Stepping through statements

Click on the Step button in the status panel, like in Figure 5-5

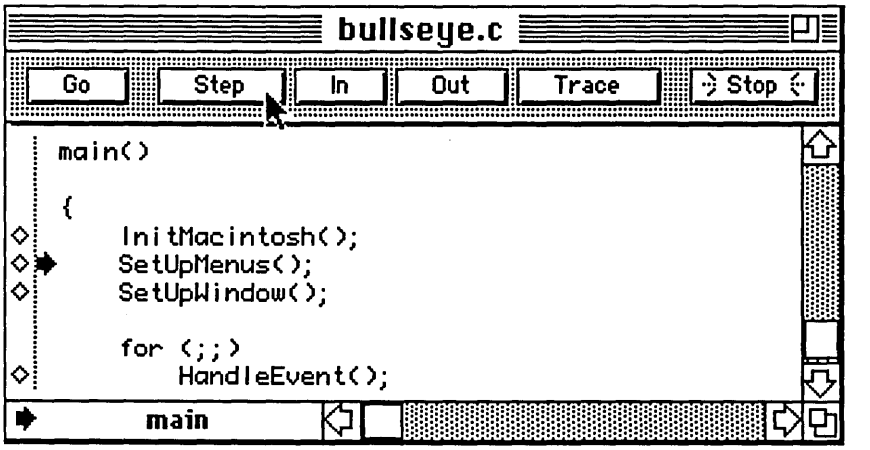


Figure 5-5 Stepping through the program

The Step button lights up for a moment, the current statement arrow moves to the second statement, and the program stops again.

The Step button lets you execute your program line by line. You can use the **Step** command in the **Debug** menu or type Command-S to do the same thing.

Press the Step button (or type Command-S) one more time so the current statement arrow points to the call to `SetUpWindow()`. This function creates the window that Bullseye uses.

Stepping into functions

*Instead of pressing the In button, you can also choose the **Step In** command in the **Debug** menu or type Command-I.*

To see how `SetUpWindow()` works, press the In button on the status panel.

Now the current statement arrow points to the first line of the `SetUpWindow()` function. This function is in the file `bullWindow.c`, so the title of the window changes to let you know what file it's displaying.

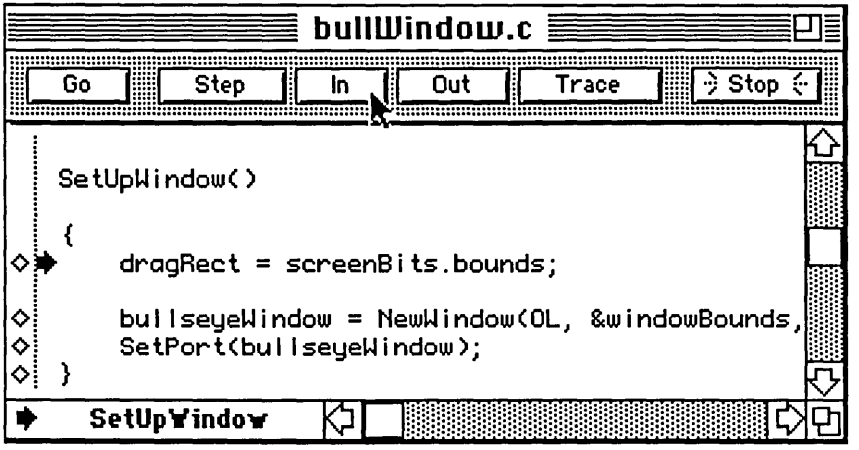


Figure 5-6 Inside SetUpWindow()

Note

> The current statement arrow doesn't have to be right before a function call for the In button to work. The **Step In** command executes every statement until the program counter is no longer in the current function. Another way to think of the **Step In** command is: "Keep going until you fall into a function." (**Step In** also stops execution if you fall out of the current function.)

Stepping out of functions

*Instead of pressing the Out button, you can also choose the **Step Out** command in the **Debug** menu or type Command- O.*

You can see the entire `SetUpWindow()` function in the source window. It's a pretty straightforward function, and you can rest assured it works.

Click on the Out button to leave the `SetUpWindow()` function.

The Source window now shows that the debugger is ready to execute the function `HandleEvent()` in the `bullseye.c` file.

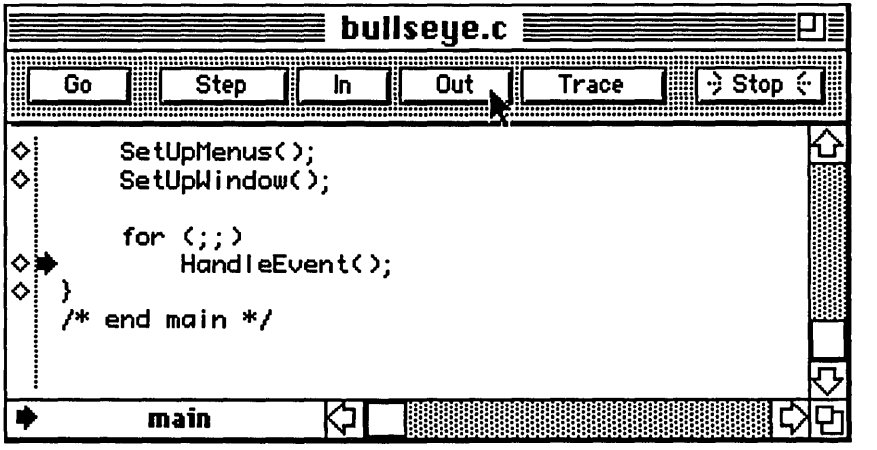


Figure 5-7 Outside SetUpWindow()

The Out button steps through each statement in the current function until the current statement arrow leaves the function.

Tracing every statement

*Instead of pressing the Trace button, you can also choose the **Trace** command in the **Debug** menu or type Command-T.*

Now click on the Trace button.

The current statement arrow now points to the first statement of the `HandleEvent()` function.

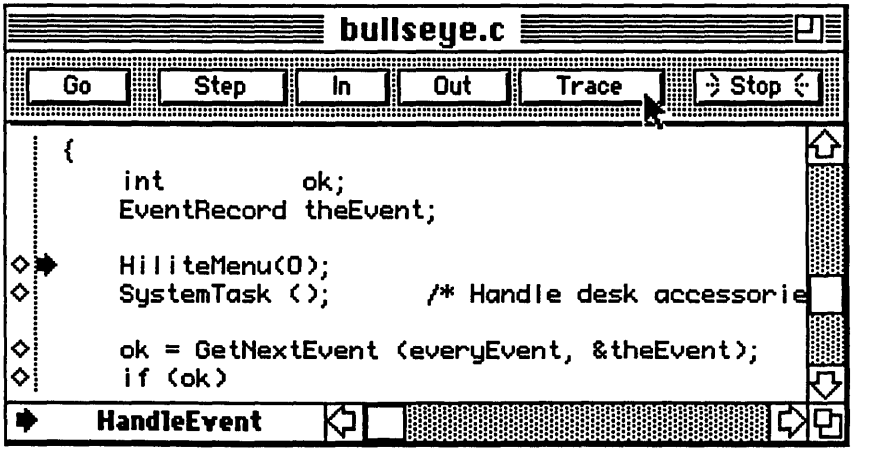


Figure 5-8 Inside HandleEvent()

Tracing takes you to the next statement even if it has to step into a function. If you were to continue tracing, you'd stop at every statement. Stepping, on the other hand, *never* dives into a function.

Note

The In button actually does a Trace until the current statement arrow leaves the current function.

Setting a breakpoint

Since the `SetUpWindow()` function opened a new window, the program will get an activate event the first time through the event loop. In Bullseye, all the program does on activate events is call `InvalRect()` on the whole window, so the second time through the event loop it gets an update event.

You could Step or Trace to verify that this is what really happens. A faster way is to set a breakpoint at the function that redraws the window.

Scroll down in the source window until you get to the code that handles update events. Click on the statement marker to the left of the `DrawBullseye()` function.

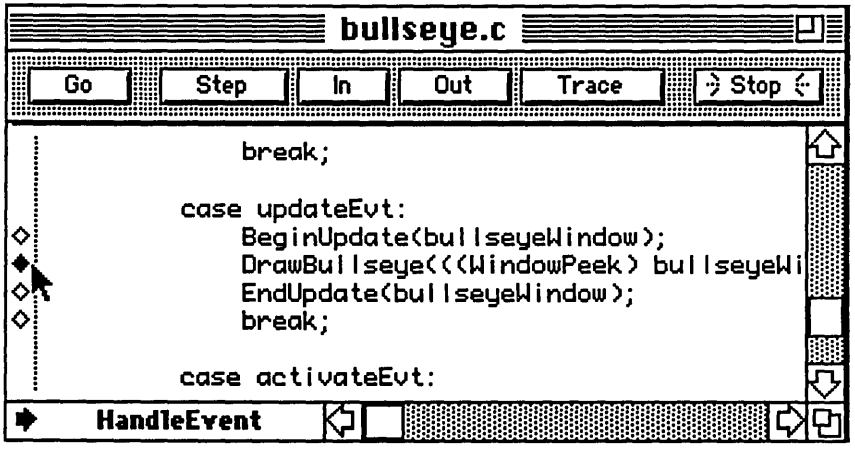


Figure 5-9 Setting a breakpoint

The hollow diamond turns black to indicate that you've set a breakpoint. You can set as many breakpoints as you like this way. When your program is about to execute a statement that has a breakpoint, it will stop. To remove a breakpoint, just click in the filled diamond.

To start your program running, press the Go button. The program runs for a few moments and then stops. The current statement arrow is at your breakpoint.

Press the In button to step into the `DrawBullseye()` function. (This function is in the `bullWindow.c` file, so that's the file you see in the source window now.)

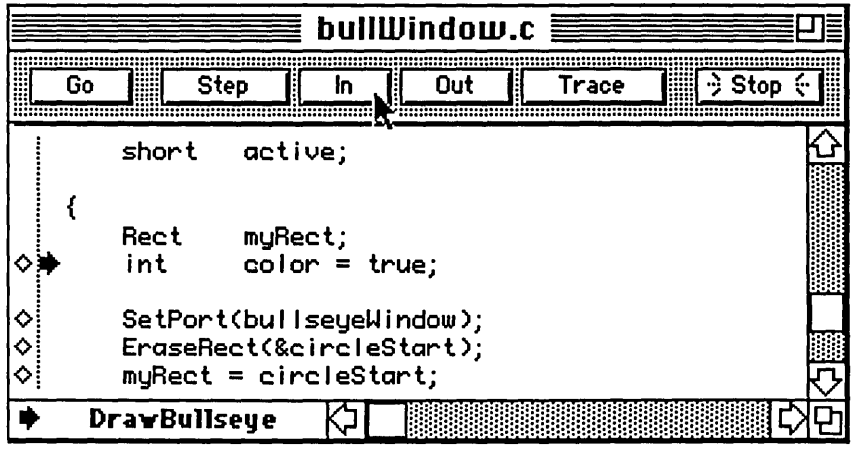


Figure 5-10 Inside DrawBullseye()

Click on the Step button to watch how the program draws a bullseye in the window. If you get bored, press the Out button. Whether you Step or Step Out, you'll eventually end back at the call to `DrawBullseye()`.

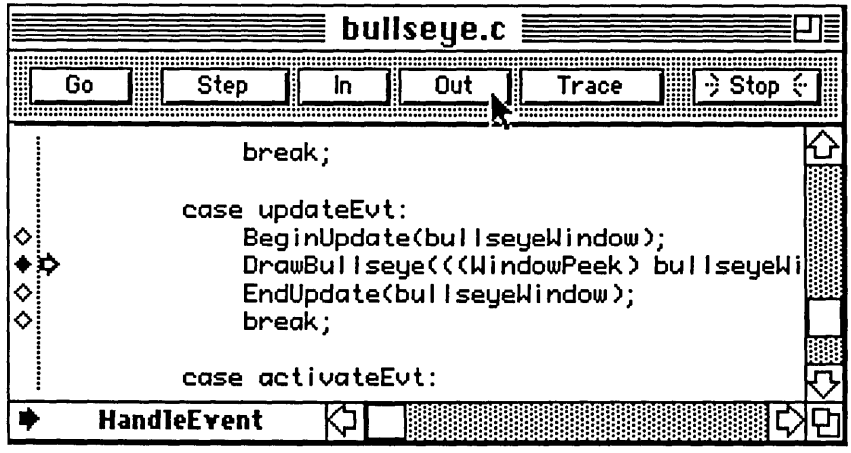


Figure 5-11 Outside DrawBullseye()

Note that the current statement arrow is hollow. This means that there are still some instructions left to execute in the statement. You'll see hollow arrows when the statement is making an assignment or cleaning up the stack after stepping out of a function.

Before you go on, clear the breakpoint. Just click on the filled diamond.

Letting the program run

Press the Go button to let the program run. You can set and clear breakpoints while your program is running.

When you click in the Source window to set breakpoints, your application will go to the background, and the debugger comes to the foreground. If you press the Go button when your program is running, the debugger brings it to the foreground.

Stopping the program

To stop your program, click on the Stop button or, if the debugger is the front-most application, press Command-Period. Your program will stop as it's coming out of one of the event-fetching routines (`GetNextEvent()` or `WaitNextEvent()`).

If your program is stuck in a loop or if you want to stop it without waiting for control to return to the event loop, you can use the **panic button**, Command-Shift-Period, to stop your program. Be careful when you do this, though, because the debugger will stop execution no matter what it's doing.

Viewing other files

The Source window usually shows the file that contains the current statement. To look at another file in the Source window (to set breakpoints in it, for example) you tell THINK C to send the text to the debugger:

1. Click on the THINK C project window
2. Click on the name of the file you want to look at
3. Select **Debug** from THINK C's **Source** menu.
4. The file that you chose appears in the Source window.

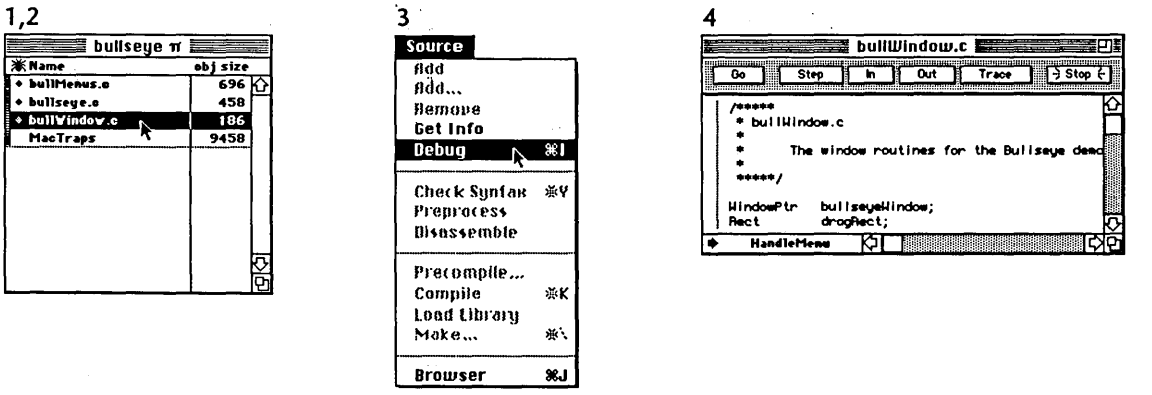


Figure 5-12 Viewing another file in the source window

Now you can examine the file and set breakpoints in it. To get back to the current file, click on the current function indicator at the lower left of the Source window or press the Enter key.

Examining and Setting Variables

Tracing your program's execution lets you see what your program is doing. But to really fix bugs, you need to be able to examine your variables. That's what the Data window is for.

If you've quit the Bullseye program, start it up again. Select **Run** from the **Project** menu, and when you see the debugger window, press the Go button in the status panel.

The Data window

The Data window appears to the right of the Source window. The best way to think about this window is to treat it as a spreadsheet.

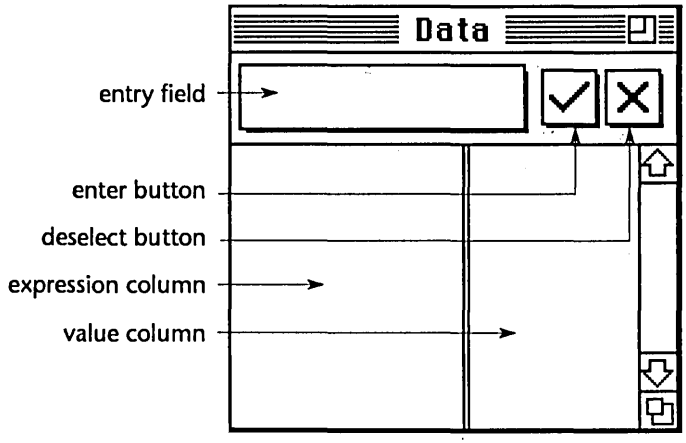


Figure 5-13 The data window of the THINK C debugger

Expressions you type into the **entry field** appear in the left column when you press the **enter button** (check mark) or when you press the Return or Enter key. Pressing the Enter key leaves the expression selected. Pressing the Return key leaves the entry field empty so you can type in the next expression. The enter button works just like the Enter key.

If you change your mind and don't want to enter an expression, press the **deselect button** (X mark). The expressions you enter appear in the left column and their values are displayed in the right column.

You can drag the center bar to make a column wider.

To clear an expression in the Data window, select it and choose **Clear** from the **Edit** menu or press the Clear key.

Examining variables

Suppose you want to watch the value of the `menuID` variable in the `HandleMenu()` function. First, make sure the `bullMenus.c` file is displayed in the source window. If it's not, bring the project window to the front, click on the name `bullMenus.c`, and select **Debug** from the **Source** menu.

Next, scroll down until you see the `HandleMenu()` function, and set a breakpoint at the `switch` statement. Remember that you can set breakpoints even while your program is running.

After you set the breakpoint, click once on the line that contains the switch statement to select it.

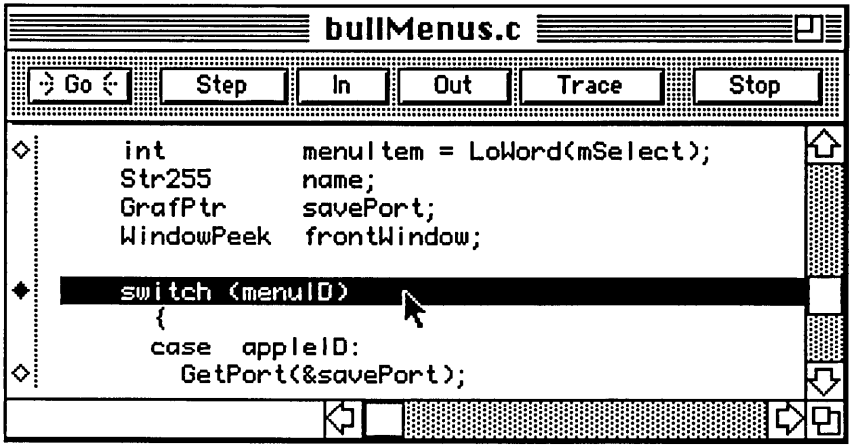


Figure 5-14 Selecting the context for the debugger

You select a line to give the debugger a **context** for evaluating `menuID`. In this case, you're saying that you want to know the value of `menuID` right before the `switch` statement.

Expressions in the Data window have either **local scope** or **global scope**. An expression has local scope if it refers to variables with dynamic storage; in other words if it refers to non-static variables local to a function. All other expressions have global scope.

Click in the Data window. You'll see the insertion point blinking in the entry field. Type `menuID` in the entry field and press the Return key.

The debugger compiles the expression (it takes about a second) in the context of the selected line. Right now, the Data window doesn't show a value for `menuID` because the program isn't stopped there.

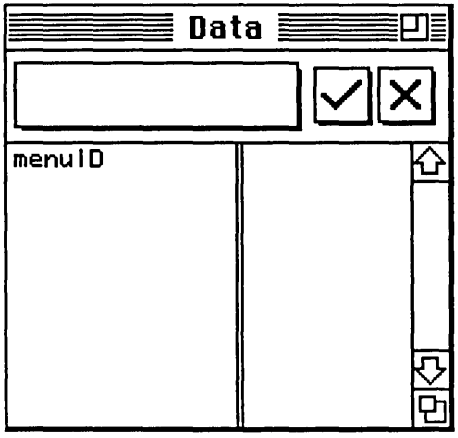


Figure 5-15 Entering menuID into the data window

Now, go back to the Bullseye program. Click on the Bullseye window, click on the Go push button (or type Command-G) to bring Bullseye to the foreground.

Next, select **7** from the **Width** menu. Your program stops at the breakpoint when you release the mouse button, and the value of `menuID` appears in the value column.

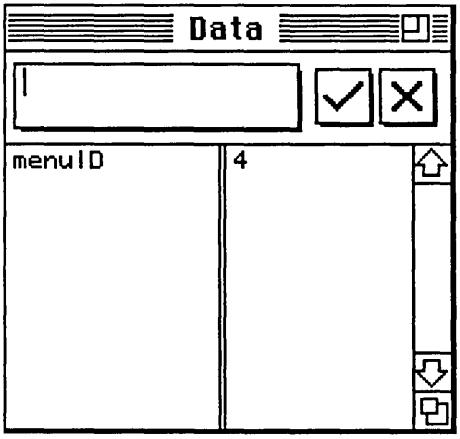


Figure 5-16 Examining the value of menuID

Any time your program stops, the source debugger displays the values of expressions that have global scope. Then it displays the values of expressions with local scope whose context is the same as the current function. Finally, the debugger clears the values of local expressions whose context is not the current function.

Changing the value of a variable

Click in the Data window again, and type `menuItem`. This variable contains the item number of the selected menu item. When you press the Return key, the source debugger shows you its value.

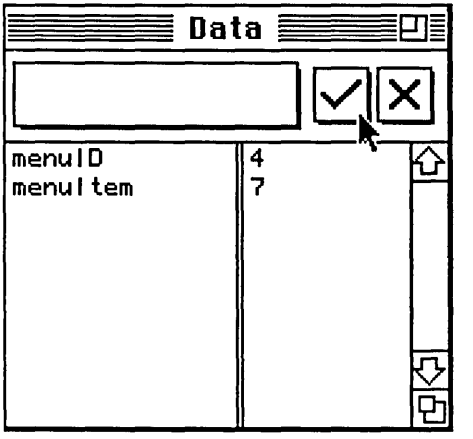


Figure 5-17 Entering menuItem in the data window

To change the value of a variable, click on its value and type a new one in the entry field. When you click on the enter button, the value of the variable changes. Here's an example.

Click on the value of `menuItem` (the right column) to select it. Its value appears in the entry field as well. Now type 8 as a new value for it. Click on the enter button to assign the new value to the variable.

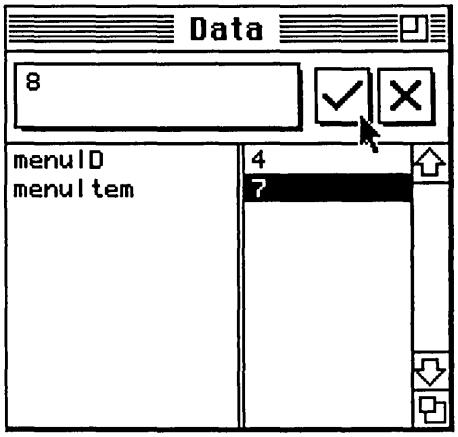


Figure 5-18 Changing the value of menuItem

When you click on the Go button, the Bullseye program behaves as if you had chosen **8** from the **Width** menu.

To remove an expression from the Data window, select it and choose **Clear** from the **Edit** menu or press the Clear key.

Note

You can enter the same expression more than once in the Data window. You might want to do this to lock one of the expressions so you can compare it to the same expression later in the program. See "How and when the source debugger evaluates expressions" on page 81.

Now choose the **Clear All Breakpoints** command in the **Source** menu to make sure there aren't any breakpoints set before you go on to the next section. Then click on the Go button to start the program running again.

Examining structs and arrays

The data window lets you examine and modify structs and arrays, not just simple variables. When you display a struct or union in the data window, its value appears as `struct 0x000000` or `union 0x000000`. Arrays appear as `[] 0x000000`. (The real address appears instead of `0x000000`, of course.)

When you double click on one of these values, the debugger displays another window for the struct or array.

Note

Anything you read here about structs applies to unions as well.

To see how this works, make sure the Bullseye program is still running. Display the file `bullseye.c` in the Source window, and set a breakpoint on the line right after the call to `GetNextEvent()` in the function `HandleEvent()`.

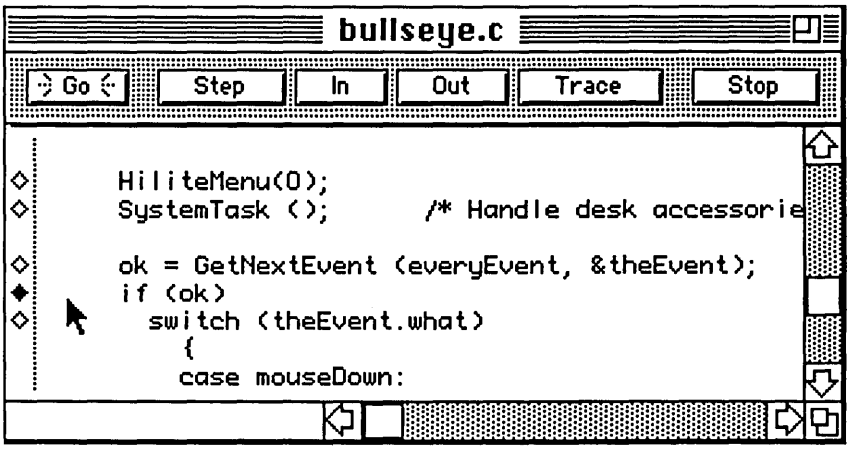


Figure 5-19 Setting a breakpoint in HandleEvent()

Now click in the Bullseye window. The program will stop at the breakpoint. When it does, type `theEvent` in the entry field of the Data window, and press the Return key.

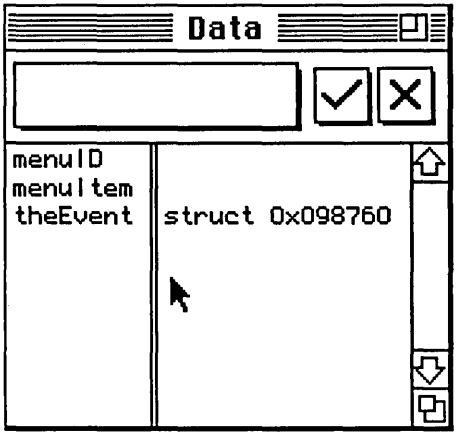


Figure 5-20 Entering theEvent in the data window

The debugger displays the word `struct` and the address of the struct. If you can't see the entire value, click on the center separator bar and drag it to the left. Or you can make the window bigger.

Note

If you don't select a line to give a variable a context, the debugger uses the current statement.

Double-click on the value of `theEvent`. The debugger displays a window. The names on the left are the fields of the struct.

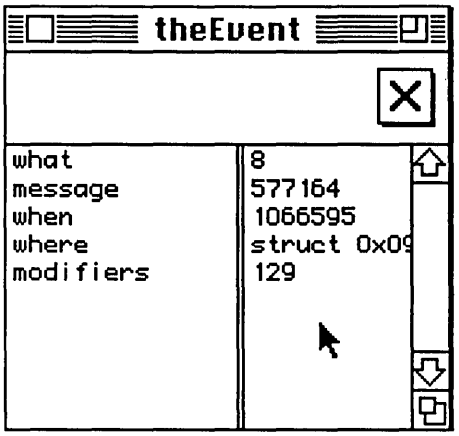


Figure 5-21 Entering theEvent in the data window

You can edit the values of the fields, but you can't edit the names.

The `what` field indicates that you're looking at an activate event (`activateEvt = 8`). In activate events, the `message` field points to the window record that gets the activate event.

Double-click on the `message` field. The debugger enters a new expression in the main Data window: `theEvent.message`.

Note

Double-clicking on the left column of any Data window creates a new entry in the main Data window.

Edit the expression so it reads `*(WindowPtr)theEvent.message` so you can look at the `WindowPtr`.

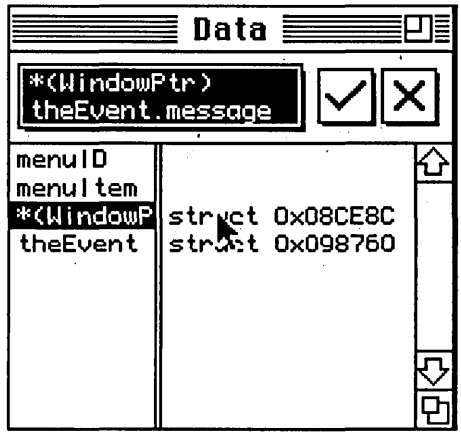


Figure 5-22 Entering *(WindowPtr)theEvent.message

Double-click on the value of the new expression. The debugger displays another struct window.

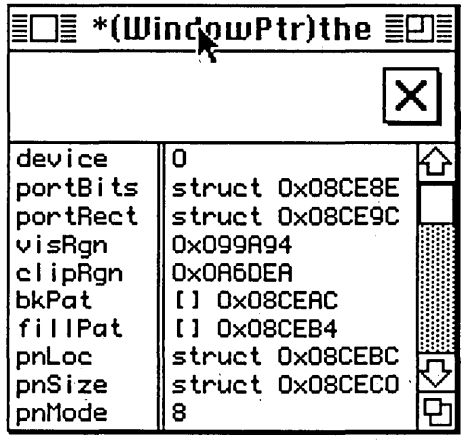


Figure 5-23 Examining the fields of *(WindowPtr)theEvent.message

Scroll down to the pnPat field, and double-click. You'll see an array window.

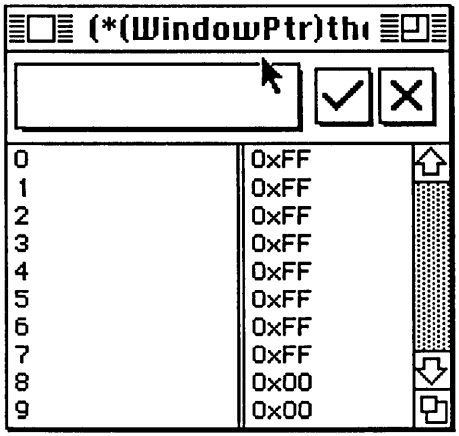


Figure 5-24 The array *(WindowPtr)theEvent.message.pnPat

Because C compilers don't enforce array bounds, array windows have "infinite" scroll bars. Unlike structs, you can select an index in the left column and change it. When you do so, the window shows the array from the index you entered

When you double click on the value of a pointer variable, the debugger inserts a dereferenced expression in the Data window and displays its value. To see a pointer as an array, change its format to Address.

To get rid of a struct or array window, you can click on its close box, press the Clear key, or select **Clear** from the **Edit** menu. When you use the Clear key or the **Clear** command, the debugger removes the expression associated with the window from the main Data window. If you click on the window's close box, only the window goes away; the expression in the main Data window remains.

Expressions and Contexts

You can enter any expression that does not have a potential side effect. That means you cannot enter assignment statements, function calls, or expressions involving ++, --, +=, etc.

Every expression you type in the entry field is compiled in a context. The context is the selected line of the Source window. If no line is selected, the context is the line that the current statement arrow points to.

To see the context of an expression, click on the expression in the left column of the Data window, and select **Show Context** from the **Data** menu. The source debugger will display the context in the source window.

To change the context of an expression, click in the source window at the line you want to use as a context. Then select an expression in the Data window and choose **Set Context** from the **Data** menu. A shortcut is to hold down the Option key as you click on the enter button.

If you edit an expression, its context will be the context of the original expression. You can change its context by holding down the Option key as you click on the enter button as described above.

How and when the source debugger evaluates expressions

Expressions in the Data window have either local scope or global scope. An expression has local scope if it refers to variables with dynamic storage; in other words, if it refers to non-static variables local to a function. All other expressions have global scope.

Every time your program stops, the debugger evaluates the expressions in the Data window. It displays the values for expressions with global scope and the values of expressions with local scope. The values of expressions that don't have a global or local context are cleared to make the window less cluttered.

If you want to make sure that the debugger doesn't redisplay a value, select it and choose **Lock** from the **Data** menu. A small lock icon appears next to the expression. This command is useful if you want to compare the value of the same expression at different times. You can also lock expressions to keep their values from being cleared when they go out of scope.

Display formats

The way the debugger displays expressions depends on their type. You can change the format with the formatting commands in the **Data** menu. To change a format, select an expression from the Data window. Then choose the format from the debugger's **Data** menu.

Not all formats are available for all types. Defaults are in italics:

Type	Formats Available
integers	*decimal*, hex, char
unsigned	*hex*, decimal, char
pointers	*pointer*, address, hex, C string, Pascal string
arrays	*address*, C string, Pascal string
structs	*address*
unions	*address*
functions	*address*
floats	*floating point*

This is what the display formats look like:

Format	Example
decimal	`4523345, -23576`
hex	`0xA09E1487`
char	`'c', 'TEXT'`
C string	`"abcdef\nghi\33"`
Pascal string	`"\pabcdef\nghi\33"`
pointer	`0x7A7000`
address	`[] 0x09FE44, struct 0x08FC14`
floating point	`1961.0102`

The C string and Pascal string formats display non-printing characters in backslash form. Whenever it can, the debugger uses the built-in escape characters (`\n`, `\r`, `\a`); otherwise, it uses `\nn`, where is an octal value.

Of course, you can use type casting to use formats that aren't normally available. For example, if you really wanted to see the integer `i` as a C string, you would type this expression: `(char *) i`.

To see any pointer as an array, just change its format to Address. This way, when you double-click on its value, you'll see an array window instead of the value of what the pointer points to.

Quitting the Debugger

The best way to quit the debugger is to quit your application. You should use the **ExitToShell** command in the debugger's **Debug** menu only when you can't use your application's **Quit** command.

THINK C

Using THINK C

Part Three

Overview 6

This chapter describes the THINK C environment and how to write a program in THINK C. The rest of the chapters in this part of the manual discuss specific aspects of the components of the THINK C environment.

Contents

The THINK C Environment

THINK C is a complete integrated development environment, not just a C compiler for the Macintosh. Traditional development environments consist of three separate applications: the editor, the compiler, and the linker. It is up to you create your source files with a text editor, run each file through the compiler, and finally link all your object files.

In THINK C, the three components work together as parts of the same application. This way, the THINK C compiler knows when you've edited a file. The compiler produces object code that the linker can put together in an instant. Then THINK C can launch your program. And because THINK C is still running, it can launch the source level debugger so you can debug your program.

The Project

The project is at the heart of the THINK C development environment. What you see on the screen is a project window. It contains a list of all the files that comprise your program. Next to each file name is the size of that file's object code.

Rather than producing a separate binary object code file, THINK C keeps all the object code in the project document in ready-to-link form.

Because the project document knows all the files that make up your program (including header files), it can keep track of changes. When you edit a source file, the project manager marks it for re-compilation. When you edit an #include file, the project manager marks all the files that use it.

Writing a Program in THINK C

Writing a program in THINK C is like writing a program in any other development environment. You create your source files, compile them, then link the object code to create an executable file. The difference is that in THINK C, you use the same application to do all of this.

Creating source files

When you write a program in THINK C, you create a project document. Usually, the project document is in a folder for all the files related to your program. Next you create your source files and add your libraries. THINK C source files are standard text files, so you can use existing source files. The THINK C editor provides some features that help you edit C source code. Its search facilities include a pattern matching option based on Grep, and a multi-file search that looks for strings in any file in your project.

Adding libraries

Virtually every program you write will need to access the Macintosh Toolbox. You can call any Macintosh Toolbox routine exactly as it's described in *Inside Macintosh*. The code for Toolbox routines marked [Not In ROM] as well as the glue code needed to call some of the other Toolbox routines is in the `MacTraps` and `MacTraps2` libraries. `MacTraps` handles the functions in *Inside Macintosh I-V* and the Gestalt Manager. *MacTraps2* handles *Inside Macintosh VI*.

Your THINK C package also includes several other libraries you can use in your programs. The `ANSI` library contains the standard ANSI functions defined in the ANSI standard. The `unix` library contains UNIX system functions including memory calls. You can use these libraries when you port code from other systems. You can also create your own libraries in THINK C.

Compiling the program

The THINK C project manager knows when files need to be recompiled. If you edit a source file, the project manager marks it. If you edit an #include file, the project manager marks all the files that depend on it. You can ask THINK C to bring your project up to date, or you can rely on the Auto-Make facility to do it for you when you run the program.

Running the program

THINK C lets you run your program from THINK C. The project manager recompiles all the marked source files and loads any unloaded libraries. Then the THINK C linker links all your code together instantly.

THINK C launches your program as if you had double-clicked on it from the Finder. This way, you know exactly how your program will behave in actual conditions. If you're running under MultiFinder, THINK C launches your program in its own partition. Since THINK C is still running, you can look at your source files while your program is running.

Debugging the program

To help you get your program working correctly, you can use THINK C's source level debugger. The debugger lets you step through your code, set breakpoints, and examine and modify variables. You can set conditional breakpoints that stop execution only when certain conditions are true.

Building the application

Finally, when you're ready to put together the final application, THINK C's smart linker examines the object code in your project to make the final file as small as possible.

Using THINK C

The best way to learn THINK C is to follow one of the tutorials in Part Two of this manual. The rest of the chapters in this part of the manual describe the components of THINK C in detail. This summary shows you the basic steps you'll take when you write a program in THINK C.

1. **Create a project folder.** Use the Finder's **New Folder** command in the **File** menu to create a folder for your project. This folder will contain the project and all the source files your project needs.

2. **Start THINK C.** Double click on the THINK C icon from the Finder.

3. **Create a new project.** When THINK C starts, it will ask you (with a standard file dialog) to open an existing project or to create a new one. Click on the New button. A second standard file dialog will appear. Move to the folder you created for your project, name your project, and click on the Create button.

4. **Create source files.** Use the **New** command in the **File** menu to get an empty edit window. To open existing source files, use the **Open...** command in the **File** menu.

5. **Save source files.** Use the **Save** command in the **File** menu to save source files. Source files must end in `.c` for THINK C to use them in a project.

6. **Add source files.** Use the **Compile** command in the **Source** menu to compile the active source file and add it to the project window automatically. If you want to add a source file without compiling it, use the **Add** command in the **Source** menu to add the front-most source file. Use the **Add...** command to add source files you haven't opened with the THINK C editor.

7. **Add libraries.** Use the **Add...** command in the **Source** menu to add libraries to the project window. A dialog appears with two lists. The top one lists the contents of the current folder. The bottom one lists the files that THINK C will add to your project. You move files Add files to the bottom list with the Add and Add All buttons. When you're done, click the Done button.

8. **Run project.** Use the **Run** command in the **Project** menu to run your project. If there are un-compelled or changed source files or

libraries that need to be loaded, THINK C will ask you if you want to bring the project up to date. Click on the Yes button. If you want to use the source level debugger, choose the **Use Debugger** command before running your program to turn it on.

9. **Build application.** Use the **Build Application...** command to turn your project into a stand-alone application. A standard file dialog will prompt you for the name of your application.

The Project 7

You can write applications, desk accessories, device drivers, and any kind of code resource in THINK C. This chapter tells you how to build these four types of projects. The first section is about projects in general. It describes some of the internal components of projects, how to use resource files with projects, and the different types of projects. The second section tells you how to break up your project into segments. The remaining sections describe the four project types.

What you should know

You should know how THINK C works. If you haven't done so, run through Chapter 4, "Tutorial: MiniEdit." That chapter takes you step by step through building an application in THINK C and gives you the practical background you need. This chapter deals with the more technical aspects of projects.

This chapter shows you how to use THINK C to build programs, but it won't give you step-by-step instructions that teach you how to put them together.

Contents

Anatomy of a Project

The project is at the heart of the THINK C development environment. It takes over the functions of several other files in traditional development environments. The project holds the object code of all your compiled source files and maintains the dependencies and connections among them. It keeps track of files that need to be recompiled or that depend on an edited header file. And if you're using the source level debugger, the project keeps the tables that the debugger needs.

Warning

You cannot put aliases in a project. However, you can use an alias of a project. See "Using Aliases" on page 155.

The project types

THINK C lets you build four kinds of projects: applications, desk accessories, device drivers, and code resources. To set the project type, use the **Set Project Type...** command in the **Project** menu. This command displays a dialog box that lets you set type-specific attributes for your project. For every project you can set the type and creator of the final file.

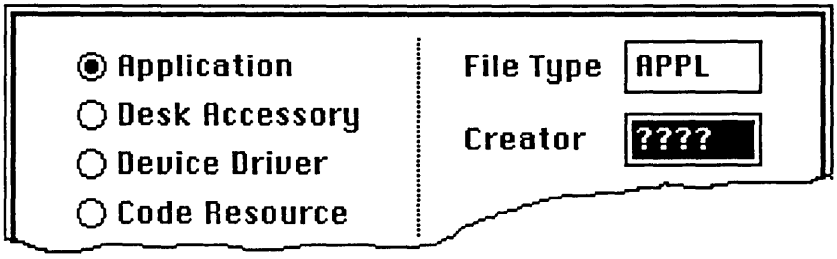


Figure 7-1 The file type and creator in the **Set Project Type...** dialog

The best time to set the project type is when you create a new project. You can change project types as you wish, but you have to recompile everything if you do so. There is a section in this chapter for each project type. Each section describes the type-specific settings.

To learn about File Types and Creators (also called signatures), see Inside Macintosh III, *Chapter 1, "The Finder Interface" or* Inside Macintosh VI, *Chapter 9, "The Finder Interface."*

The File Type and Creator of a file let the Finder know what icon to display. The Finder treats some types, like `RDEV`, `INIT`, and `cdev`, specially.

When you set the project type to something other than **Application**, the name of the **Build Application...** command in the **Project** menu changes accordingly. For example, if you set the project type to **Desk Accessory**, the menu reads **Build Desk Accessory...**.

Changing the type of a project changes the way a project is built, not the way a program behaves. You can't turn an application into a desk accessory merely by changing the project type. Desk accessories are structurally different from applications. The same applies for the other types.

When you choose one of the **Build...** commands (**Build Application...**, **Build Desk Accessory...**, etc.) from the **Project** menu, THINK C creates a file and creates the CODE resources for applications, a DRVR resource for desk accessories and drivers, or the type of resource you specify for your code resource. THINK C also creates the resources it uses to manage global data and inter-segment calls.

Components of a project

Each source file or library in a project has up to four object components: (Not all project types use all four components.)

This component...	Contains...
CODE	The executable code for the project.
DATA	The global and static variables.
STRS	String literals, in applications that have the "Separate STRS" option on.
JUMP	The jump table, which multi-segment projects use to determine the address of a routine.

To examine the size (in bytes) of each component, select a source file in the project window and choose **Get Info...** in the **Source** menu. The display also shows the segment and project totals. (For some components there is a small amount of overhead per segment or per project.)

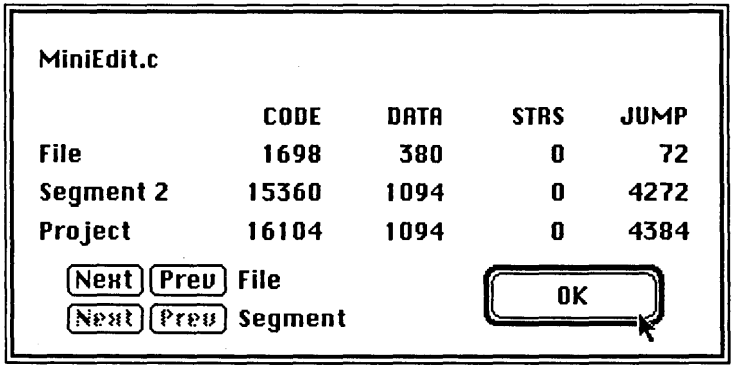


Figure 7-2 The Get Info... dialog

Depending on the kind of project you build, there are different limits on the sizes of these components:

If project is a(n)...	**The limits are...**
Application	CODE: 32K per segment DATA: unlimited per project, 32K per file, with Far DATA (32K per project, without Far DATA) JUMP: 256K per project, 32K per file, with Far CODE (32K per project without Far CODE) STRS: unlimited per project, 32K per file, if "Separate STRS" is on (No STRS if "Separate STRS" is off)
Single-segment desk accessory or device driver	CODE: 32K per project DATA: 32K per project JUMP is not used for single segment drivers
Multi-segment desk accessory or device driver	CODE: 32K per segment DATA + JUMP: 32K per project
Single-segment code resource	CODE + DATA: 32K per project JUMP is not used for single segment code resources
Multi-segment code resource	Segment with `main()`: CODE for segment + DATA for project + JUMP for project: 32K Other segments: CODE: 32K

If you exceed any of these limits, you'll see an error at link time.

THINK C generates an 8 byte JUMP table entry for each function that is not declared `static` and for each function used in a non-call context (i.e. whose address is taken). The jump table built by the **Build ...** command may be smaller. The entry for each function that is only referenced within a single segment is removed.

How THINK C puts projects together

When you choose a **Build...** command from the **Project** menu, THINK C puts all the pieces of your project together as compactly as possible. THINK C uses a technique called **smart linking** to extract only the CODE elements it needs to build your application, desk accessory, device driver, or code resource.

To learn more about libraries in THINK C see Chapter 14, "Libraries."

THINK C searches the project for files or libraries whose code is never referenced and ignores them during the link. If you use a project as a library, THINK C takes only the code it needs from the project. If you use a library that you converted from a `.rel` file or one that you built with the **Build Library...** command, all of its code is included in the final file.

Smart linking yields smaller code, but it takes several seconds longer to produce the final file. You can choose not to use smart linking; your files are larger, but they come out faster.

Note

For desk accessories, code resources, and applications, THINK C checks the components' size limits (described above) *after* smart-linking.

To turn smart linking off, choose your **Build...** command from the **Project** menu as you normally do, then click on the Smart Link check box to clear the option. (It's on by default.) See Figure 7-3

After THINK C finishes linking the object code, it produces the application, desk accessory, device driver, or code resource file. Finally, it copies any resources you put in the project resource file into the final file.

Using resource files with projects

Most Macintosh applications use resources for menus, window templates, dialogs, etc. THINK C makes it easy to use resources with your applications.

SARez is described in Chapter 18, "Using SARez." ResEdit is described in the book ResEdit 2.1 Reference *(Addison/Wesley) by Apple Computer, Inc.*

Use ResEdit or SARez to create a file that contains the resources your program needs. Save the resource file with the same name as your project plus `.rsrc`. For instance, if the name of your project is `MyProject`, name your resource file `MyProject.rsrc`. If your project is named `SuperWizzy.project`, THINK C looks for the project's resources in `SuperWizzy.project.rsrc`.

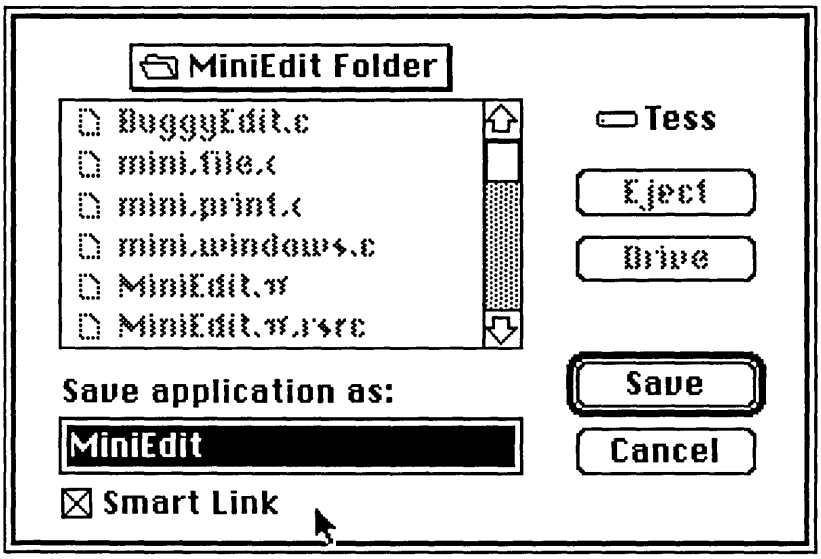


Figure 7-3 The Smart Link check box in a **Build...** dialog

Note

Make sure the resource file is in the same folder as your project so THINK C can find it.

When you choose **Run** from the **Project** menu, THINK C opens the resource file automatically, so your program can access its resources.

If you're building a code resource and the Merge option is on, the project's resource file isn't copied. See "Merging code resources into files" on page 123.

When you choose one of the **Build...** commands from the **Project** menu (**Build Application...**, **Build Desk Accessory...**, etc.) THINK C copies the resource file into the finished application.

Warning

THINK C does not support aliases. If you use an alias as your project's resource file, THINK C will not resolve the alias, but will use the alias file's `alis` resource as your project's sole resource.

Segmentation

Most Macintosh programs are made up of several **segments**. Segments are units of object code that can be swapped in and out of memory as needed.

To learn more about segments, see Inside Macintosh II, *Chapter 2, "The Segment Loader."*

The Macintosh Operating System limits segments to 32K, so if you're writing a large program, you must segment your code.

THINK C lets you segment not only applications, but also desk accessories, device drivers, and code resources as well. Applications can have up to 254 segments. Desk accessories, device drivers, and code resources can have up to 30 segments.

Moving files into segments

Dotted lines separate segments, and the current segment has gray hatching running along the left edge. When you add files to your project, they're added to the current segment.

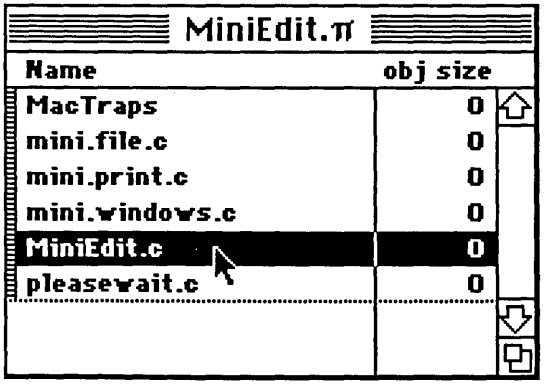


Figure 7-4 A project with one segment

To move a source file into another segment, click on its name in the project window and drag it below the dotted line.

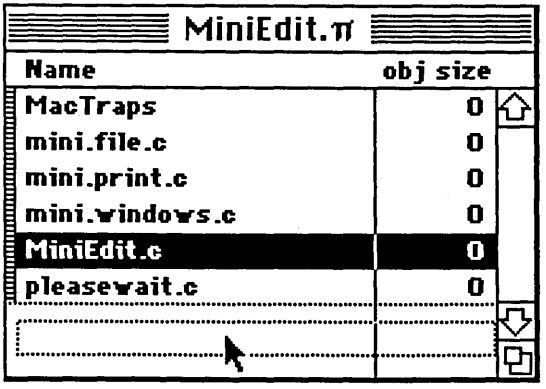


Figure 7-5 Moving a file to a new segment

Source files appear in alphabetical order within segments, so even if your program is small enough to fit in one or two large segments, you might want to break it down into smaller segments for cleaner organization.

Moving entire segments

If you want to rearrange whole segments at a time, hold down the Option key as you click and drag on any file name in the segment. The entire segment is placed before the segment where the drag ends. Moving segments affects only the project window display.

Deleting a segment

To delete a segment, move all the files from it into other segments. THINK C automatically gets rid of a segment when you move the last file out of it. When you use the **Add...** command, new files are placed in the current segment.

Building Applications

THINK C is set up to build applications by default. The **Set Project Type...** dialog gives you an opportunity to set some application attributes. Choose **Set Project Type...** from the **Project** menu, and you see the dialog in Figure 7-6:

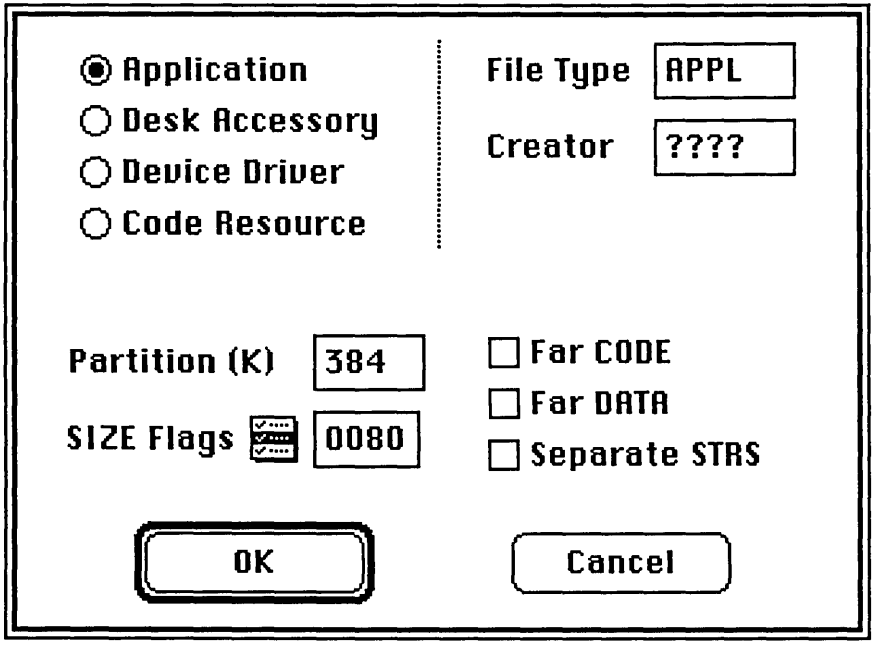


Figure 7-6 The Set Project Type... dialog for applications

To learn about File Types and Creators (also called signatures), see Inside Macintosh III, *Chapter 1, "The Finder Interface" or* Inside Macintosh VI, *Chapter 9, "The Finder Interface."*

Setting the application file type and creator

When you're building an application, the default file type is APPL. (For applications to work with the Finder, applications must be of type APPL.) Set the Creator to whatever you want your application's signature to be. The Finder uses the file's creator to link icons with specific applications (See *Inside Macintosh III*, Chapter 1, "The Finder Interface" to learn more about applications and icons.)

Using a larger jump table with Far CODE

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

- Two entries for each virtual method
- One entry for each routine called by a routine in another segment
- One entry for each routine that you take the address of

If you need a large jump table, check the "Far CODE" option. It allows a jump table as large as 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.

Note

If the "Far CODE" option is on, each of your project's files can contribute only 4095 jump table entries (or 32K).

Using more global data with Far DATA

If you use a lot of global data, check the "Far DATA" option. It allows you to have an unlimited amount of global variables, since it uses 32-bit absolute addresses instead of 16-bit relative addresses. Your application will be about 8% larger because of the longer addresses. If the "Far DATA" option is off, your project can contain only 32K of global variables.

Note

If the "Far DATA" option is on, each of your project's file can contribute only 32K of global data. This limit means, for example, that you can't use an array that's over 32K.

How Far CODE and Far DATA work

Far CODE and Far DATA both work similarly. You can have larger applications with more space for your jump table and global variables since THINK

C handles addresses into these areas differently. When these options are off, addresses into these areas are given as offsets from A5. In the MC68000 family, register-relative offsets must be 16-bits long, so these areas can contain only 32K. When the option is on, addresses to these areas are absolute addresses and can be 32-bits long. THINK C includes code in your application that adds the value of A5 to these absolute addresses before your code runs.

Mixing Near CODE and DATA with Far CODE and DATA

If the "Far CODE" or "Far DATA" option is on and your application contains libraries that were compiled with those options off, put those libraries together in their own segment (or segments). THINK C places the jump table entries for those libraries in first 32K of the jump table. If the libraries are grouped together, THINK C 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.

Using a separate STRS component

THINK C normally places string literals in the DATA component of a project. When you check "Separate STRS" option, THINK C uses a separate STRS component to store string literals.

When this option is off (the default), THINK C uses a 2 byte offset from register A5 to access the string literals. Not only is the code smaller but there's no need to do run-time address fix-ups. When the "Separate STRS" option is on, THINK C uses a 4 byte absolute address to access the string literals.

The "Separate STRS" option is useful only if you need a library compiled with a previous version of THINK C. If your project contains a library that was compiled with the "Separate STRS" option on, you must turn on the "Separate STRS" option in your project, too. If you need more space for global data, use the "Far DATA" option, described in "Using more global data with Far DATA" on page 100.

Setting the partition size and SIZE resource flags

THINK C uses the values you set in the Partition and SIZE Flags fields to build the SIZE resource your application needs to run under MultiFinder or System 7.0.

The **partition size** determines how much memory your application gets. The default partition size is 384K, which is more than enough for moderate size applications. You'll want to use a higher value for larger applications or a lower value for small applications when memory is tight.

See "Apple Programmer's and Developer's Association (APDA)" on page 14 for more information about APDA.

The SIZE Flags field sets the flags that tell MultiFinder and System 7.0 how compatible your application is and which features it uses. To learn how to write an application for the MultiFinder under System 5.0 or 6.0, order the *Programmer's Guide to MultiFinder* from APDA. To learn how to write application for System 7.0, see *Inside Macintosh VI*. For more information on the `SIZE` resource, see the chapter "The Event Manager" in *Inside Macintosh VI*.

You can type in the value of the flag (in hexadecimal) in the field, or you can set the bits from the pop-up menu next to the field. The pop-up menu is in Figure 7-7.

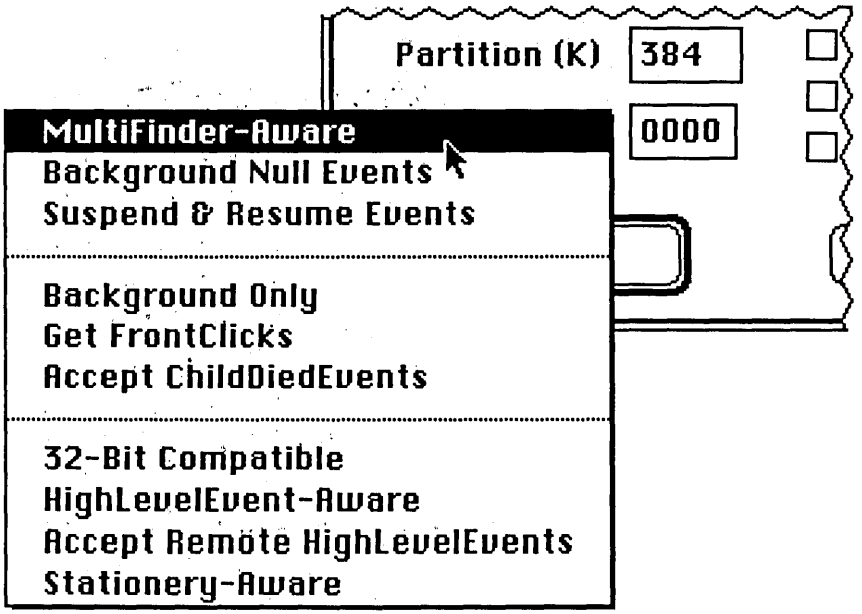


Figure 7-7 The SIZE Flags menu

The bits fall into three categories: basic MultiFinder support, extra MultiFinder features, and System 7.0 features. The first three bits describe how compatible your application is with MultiFinder and System 7.0:

- If the MultiFinder-Aware bit is set, the Finder expects you to conform to the guidelines for shifting from the foreground to the background layers. Your application gets suspend/resume events as your application shifts from foreground to background but not activate/deactivate events. If MultiFinder Aware is checked, Suspend & Resume events should be checked as well.

- If the Background Null Events bit is set, your application gets regular null events when your application is in the background. Otherwise, your application gets only update events.
- If the Suspend & Resume Events bit is set, your application gets these events as it shifts from the foreground to the background layers in addition to the activate/deactivate events you normally receive.

Note

If you're using the THINK Class Library, be sure you check MultiFinder-Aware and Suspend & Resume Events.

The next three bits explain which of the extra features of MultiFinder and System 7.0 your application uses.

- If the Background Only bit is set, your application runs only in the background. Set this bit if your application has no user interface and cannot run in the foreground.
- If the Get FrontClicks bit is set, your application receives the mouse-down and mouse-up events that bring your application to the foreground.
- If the Accept ChildDiedEvents bit is set, your application is notified when an application it launched quit or crashed.

The last four bits tell the Finder which of these new System 7.0 features your application supports: 32-bit memory, AppleEvents, and stationery documents.

- If the 32-bit Compatible bit is set, your application runs under the 32-bit version of Memory Manager. Don't set this flag unless you've tested your application on a 32-bit system, like a Macintosh IIci running System 7 in 32-bit mode or a Macintosh running A/UX.
- If the HighLevelEvent-Aware bit is set, your application receives all high-level events, including AppleEvents, when it calls `WaitNextEvent`. There is no way to mask out the types of high-level events your application can't handle. If this bit isn't set, your application receives no high-level events.

- If the Accept Remote HighLevelEvents bit is set, your application receives high-level events, including AppleEvents, from your computer and other computers on your network. If this flag isn't set, your application receives high-level events from your computer only.

- If the Stationery-Aware bit is set, the Finder expects your application to handle stationery documents. If this bit isn't set, the Finder handles stationery documents for you, by duplicating the document and asking the user for a name for the duplicate.

Running the project

The **Run** command in the **Project** menu lets you run your application project as you work on it. When you choose the **Run** command, THINK C launches your application as if you had opened it from the Finder. If you're using MultiFinder or System 7.0, your application runs in its own partition.

For more information on the debugger, see Chapter 12, "The Debugger."

Under MultiFinder or System 7.0, you can watch your application run, examine your source code, and use the THINK C debugger at the same time. You can switch back and forth between your application, THINK C, and the debugger to edit your source files and examine your variables. When you quit your application, you're be back in THINK C, and the auto-make facility is ready to recompile your changed files. While your application is running, the **Run** command changes to **Resume**. Choosing **Resume** brings your application to the foreground.

When you're running under MultiFinder or System 7.0, you have to be a little more careful about stray pointers and much more conscientious about backups. A pointer to random memory may be pointing into another application's partition. A stray pointer to THINK C's data structures may damage your project, and the damage is copied out to disk. Be careful with stray pointers. If this happens to you, delete the damaged project, and start over with your backup and bring it up to date with the Use Disk button in the **Make...** dialog.

Building Desk Accessories and Device Drivers

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

This section won't teach you how to write a desk accessory or a device driver from scratch. To learn how to write them, read *Inside Macintosh I*, Chap-

ter 14, "The Desk Manager," *Inside Macintosh II*, Chapter 6, "The Device Manager," and *Inside Macintosh V*, Chapter 23, "The Device Manager."

Note

For System 7.0, Apple recommends that you write small applications instead of desk accessories.

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 must recompile your source files and reload your libraries.

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

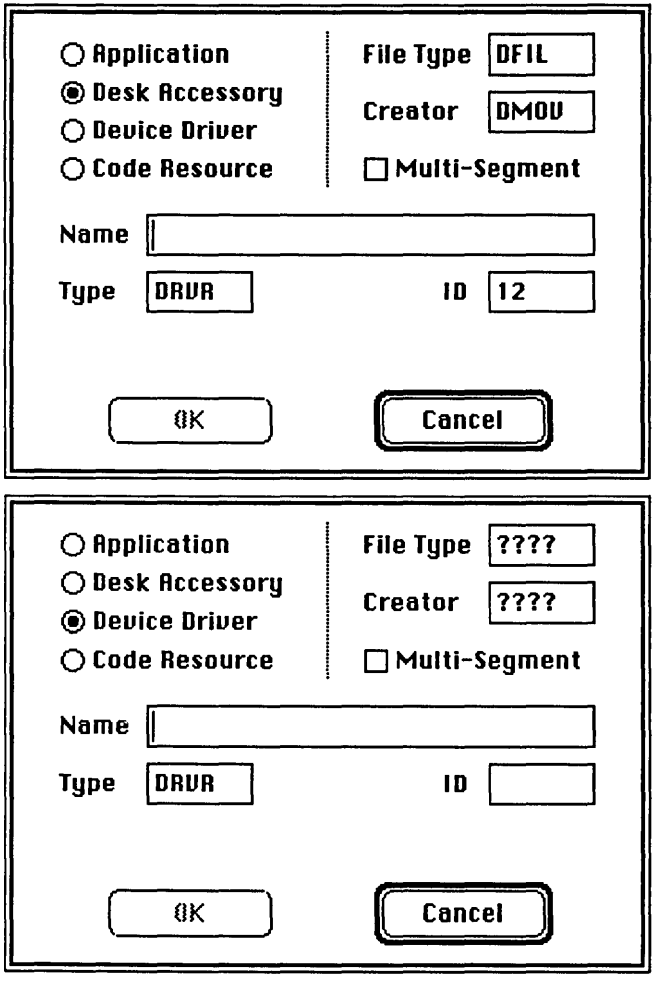


Figure 7-8 The **Set Project Type...** dialogs for desk accessories and drivers

The dialog boxes are the same, except for the default settings of some fields. The Desk Accessory dialog presets the File Type and Creator so the resulting file is a Font/DA Mover file. The ID is set to 12. The Font/DA Mover renumbers 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 null. THINK C provides the null for you automatically.

Note

In System 7.0, you don't need the Font/DA Mover, since the Finder lets you install fonts and desk accessories.

The Device Driver dialog leaves all the options empty for you to set. Device driver names begin with a period. If you don't provide one in the Name field, THINK C automatically provides one for you.

The default settings in the project type dialogs aren't the only difference between desk accessories and device drivers. See "Setting the fields of a driver's header" below to learn about the internal differences between desk accessories and device drivers.

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.

Note

To use objects in a driver, you must turn on the Multi-Segment option, even if your driver has 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 C, a short assembly-language stub (or glue routine) translates the Device Manager's request into a call to the C function `main()`. THINK C automatically places the driver glue at the beginning of your driver.

THINK C does two things 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. These services are discussed later in this section.

How to write main() for a driver

The driver's `main()` function takes three arguments. The function returns a `short int` and is *not* declared `pascal`. A typical driver skeleton looks like this:

```
short main (CntrlParam *paramBlock,
   DCtlPtr devCtlEnt, short n)
{
   switch (n) {
      case 0 :     // Open
      case 1 :     // Prime
      case 2 :     // Control
      case 3 :     // Status
      case 4 :     // Close
   }
}
```

`MacHeaders` *includes the files* `Files.h` *and* `Devices.h` *by default.*

`ParamBlock`, defined in `Files.h`, is a pointer to an I/O parameter block. This is the value that is passed in address register `A0` to the assembly-language entry point of the driver.

`DevCtlEnt`, defined in `Devices.h`, is a pointer to the driver's device control entry. This is the value that is passed in address register `A1` to the assembly-language entry point of the driver.

`N` is a selector that specifies which entry point actually received the call. Use the value of `n` to dispatch control to the appropriate routine.

Getting the event record pointer from paramBlock

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`. Since `csParam` is defined as an array of `int`s, this is how you cast the field to get a pointer to the event record (assume that `eventPtr` is declared `EventRecord *`):

```
eventPtr=(*(EventRecord **) paramBlock->csParam)
```

Global data in drivers

You can declare global and static variables in drivers. The THINK C 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

In drivers, string literals are stored the same way as global variables.

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 C 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 is 0, and your driver should display some error message (without using any globals!) and close itself.

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 do not move.

Using driver globals in callback and trap intercept routines

The THINK C driver glue sets up register `A4` for you whenever it's 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 header file `SetUpA4.h` defines a set of macros that take care of saving, setting, and restoring `A4` for you.

Suppose your driver calls `ModalDialog()` with a `filterProc`. Since you're not sure if the value of `A4` is correct when `ModalDialog()` calls your `filterProc`, you need to save it. Your `filterProc` needs to set `A4`

to the saved value and then restore it before it exits. Your call to `ModalDialog()` would look like this:

```
engage_in_dialog()
{
    extern pascal Boolean myFilter();
    int item;
    ...
    RememberA4();
    ModalDialog(myFilter, &item);
    ...
}
```

Your `filterProc` would look like this:

```
pascal Boolean myFilter(DialogPtr dp,
                        EventRecord eventp,
                        short *itemp)
{
    Boolean result;

    SetUpA4();
    ...
    RestoreA4();
    return(result);
}
```

The calls to `RememberA4()` and `SetUpA4()` must appear in the same source file.

Note

If your routine uses an aggregate initilaizer for a local variables, that variables must be declared `static` or it will have the wrong value.

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 libraries in drivers

You can use libraries in drivers as long as the libraries don't reference global variables accessed through register `A5`.

The `MacTraps` and `MacTraps2` libraries don't reference any globals, so you can use them in your drivers. `MacTraps` does define the QuickDraw globals, though. If you access these globals from a driver, you'll find that

they aren't the real QuickDraw globals but simply the driver's own variables that QuickDraw knows nothing about.

You can access the real QuickDraw globals from a driver 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`. You might want to use the inline assembler to get to the QuickDraw globals. See Chapter 12 for details.

Other libraries supplied with THINK C reference their globals from register `A5`, so you must modify them. To make a library use register `A4` for its globals, first make a copy of the library. Then change the project type of the library to anything but Application, and recompile and rebuild the library. See Chapter 13 to learn how to build libraries.

Setting the fields of a driver's header

A driver begins with a header containing 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 C presets these fields to reasonable default values. If your device driver or desk accessory requires different settings for these fields, modify them on the fly in the device control entry.

These are the default settings for desk accessories:

Field	Value
`dCtlFlags`	`dReadEnable 0`
	`dWritEnable 0`
	`dCtlEnable 1`
	`dStatEnable 0`
	`dNeedGoodbye 0`
	`dNeedTime 0`
	`dNeedLock 0`
`dCtlDelay`	N/A because `dNeedTime = 0`
`dCtlEMask`	`0x016A` (mouseDown, keyDown, autoKey, update, activate)
`dCtlMenu`	0

These are the default settings for device drivers:

Field	Value	
dCtlFlags	dReadEnable	1
	dWritEnable	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)	

Suppose you want a desk accessory to remain locked between calls and to be called once per second (every 60 ticks). Just include this code in the driver's Open routine. (devCtlEnt is the second argument to main(), the pointer to the device control entry.)

```
devCtlEnt->dCtlFlags |= dNeedLock|dNeedTime;
devCtlEnt->dCtlDelay = 60;
```

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.

You can set the fields of the device control entry directly as shown above, but the Device Manager copies the dCtlFlags, dCtlMenu, dCtlDelay, and dCtlEMask fields of the driver's header to the corresponding fields of the device control entry *every time* the Open routine is called, even if the driver is already open. So you need to set these fields to their proper values each time. Your Open routine might look something like this:

```
doOpen()
{
   devCtlEnt->dCtlFlags|= dNeedLock; // or
                                     // whatever
   if (already_open)     // already_open is a
      return;            //   driver global
   already_open = 1;     // one-time
}                        //   initialization.
```

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 isn't 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 C driver glue preserves 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 makes it seem as though your `Close` routine returned 0, meaning the `Close` was successful.).

Warning

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 is stored in the `ioResult` field of the I/O parameter block, but 0 (no error) is returned to the Device Manager.

The jIODone problem

THINK C 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 "good-bye 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` attempts to examine the driver's I/O queue, and since the queue is usually empty it ends up examining low-memory locations beginning at `0x0000`. 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 C avoids this dilemma. When a driver written in THINK C 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. 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.

You can unload driver segments manually with this function:

```
void UnloadA4Seg(ProcPtr);
```

This function works just like `UnloadSeg()` does in applications.

Warning

Do not use `UnloadSeg()` instead of `UnloadA4Seg()` by mistake!

When THINK C creates a multi-segment driver, it puts the code segment that contains the `main()` function in a DRVR resource. This is called the owning resource. You set its ID in the **Set Project Type...** dialog, which must be 63 or lower. THINK C puts the other code segments in DCOD resources and numbers them from 0 to 31. These are called owned resources.

Read the Segmentation section above to learn how to break up a project into different segments.

Building Code Resources

You can use THINK C 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, `main()`.

You might want to write code resources for several reasons. You might want to write a window definition function that you can use in several other programs, or you might want to write an `INIT` to run at start-up. You may define your own code resource types to make a function you've written in THINK C available to a program written in another language. The "client" program simply loads the resource and calls it at its beginning. It's up to you whether you use C or Pascal calling conventions. (For more information about calling conventions, see Chapter 12.)

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

To build a...	**Read *Inside Macintosh*...**
ADBS resource	Vol. V, Chap. 20, "The Apple Desktop Bus"
CDEF resource	Vol. I, Chap. 10, "The Control Manager"
cdev resource	Vol. V, Chap. 18, "The Control Panel"
FKEY resource	Technical Note 3 (also see below)
INIT resource	Vol. IV, Chap. 29, "The System Resource File" Vol. V, Chap. 19, "The Start Manager"
LDEF resource	Vol. IV, Chap. 30, "The List Manager Package"
MBDF resource	Vol. V, Chap. 13, "The Menu Manager"
MDEF resource	Vol. I, Chap. 11, "The Menu Manager" Vol. V, Chap. 13, "The Menu Manager"
WDEF resource	Vol. I, Chap. 9, "The Window Manager"
XCMD resource	Appendix A, *HyperCard Script Language Guide*
XFCN resource	Appendix A, *HyperCard Script Language Guide*

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 must recompile your source files and reload your libraries.

Choose **Set Project Type...** from the **Project** menu. When the project type dialog appears, click on the Code Resource check box.

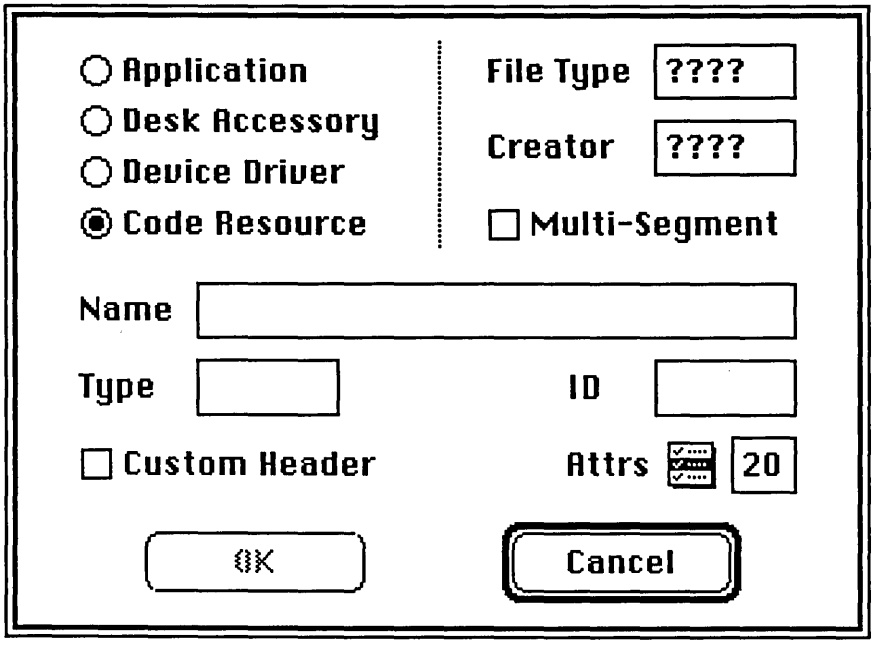


Figure 7-9 The **Set Project Type...** dialog for code resources

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 "Attrs" (attributes) pop-up menu to set the resource attributes for your code resource. If you prefer, you can enter a hex value in the Attrs field. To learn about resource attributes, see *Inside Macintosh I*, Chapter 5, "The Resource Manager."

Code resources can have more than one segment. If your code resource has multiple segments, select the Multi-Segment option. Read the section "Multi-segment code resources" for more information.

Warning

To use objects in a code resource, you must turn on the Multi-Segment option, even if your code resource has only one segment.

How to write main() for a code resource

The way you write your `main()` routine depends on the kind of resource you're writing. The main point to remember is that if your code resource is going to be called from a Pascal program or as a callback routine, it must be declared `pascal`.

An `FKEY` resource, for example, is called by the Event Manager. It doesn't have any arguments. You would define `main()` like this:

```
main()
{
    ...
}
```

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 `long`. This is how you would write `main()` for a `WDEF`:

```
pascal long main (short varCode,
                  WindowPtr theWindow,
                  short message, long param)
{
    ...
}
```

Global data in code resources

Code resources, like applications and drivers, can have global and static variables.When you build a single-segment code resource, the DATA component is appended to the CODE component, so CODE and DATA together must be less than 32K. When you build a multi-segment code resource, the DATA and JUMP components from all the segments are appended to the CODE component of the segment that contains `main()`, so all the DATA and JUMP information and the CODE component for the main segment must be less than 32K.

Code resource globals are addressed as offset 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.

Note

If your `main()` routine uses an aggregate initilaizer for a local variable, that variables must be declared `static` or it will have the wrong value.

Warning

In multi-segment code resources, THINK C treats string literals the same way as globals. If you're using literals and constants but not globals, you still need to set up A4. In single-segment code resources, string literals are *not* treated the same way as globals.

When main() is called, A0 points to your code resource. This is the same value that your code resource expects A4 to have to find your globals.

The header file SetUpA4.h contains a set of macros that help you set up the A4 register. Immediately after you enter main(), call RememberA0(). This macro saves the value of A0 where another macro, SetUpA4(), can find it. You must call RestoreA4() before you return from main(). For example:

```
main()
{
    RememberA0(); // To access resource globals
    SetUpA4();
    ...

    RestoreA4();
}
```

Warning

This technique works only when you use the default code resource header. If you use a custom header, you must set up A4 another way. See "Code resource headers" below to learn how to do this.

Using 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 MacTraps and MacTraps2 libraries do not access any globals, so you can use them in your code resources. MacTraps does define the QuickDraw globals, though. If you access these globals from a code resource, you'll find that they aren't the real QuickDraw globals but simply the resource's own variables that QuickDraw knows nothing about.

You can access the real QuickDraw globals from a code resource 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. You might want to use the inline assembler to get to the QuickDraw globals. See Chapter 12 for details.

The other libraries supplied with THINK C reference their globals from register A5, so you must modify them. To make a library that uses register A4 for its globals, first make a copy of the library. Then change the project type of the library to anything but Application, and recompile and rebuild the library. To learn more about libraries, see Chapter 13.

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, register A0 contains a pointer to your code resource. If you need to lock it, you would write main like this:

```
#include <SetUpA4.h>

main()
{
    Handle h;

    RememberA0();          // To access resource
    SetUpA4();             //   globals

    asm {
        _RecoverHandle     // a0 already points
        move.l   a0, h     //   to resource
    }
    HLock(h);

    ...

    HUnlock(h);
    RestoreA4();
}
```

Warning

This technique works only when you use the standard code resource header. If you use a custom header, you must get the address of your code resource another way. See "Code resource headers" below.

If your code resource can be called reentrantly, it should not unconditionally be unlocked each time it returns. Instead, it should be restored to the same state of locked-ness it had on entry.

Code resource headers

If the Custom Header option is off, THINK C creates a code resource with a standard header:

Offset	Contents
0	`BRA.S .+0x10` (branch to header code)
2	`0x0000` (unused)
4	`'TYPE'` (resource type)
8	`0x000A` (resource ID)
10 (0xA)	`0x0000` (unused)
12 (0xC)	`0x0000` (unused)
14 (0xE)	`0x0000` (unused)

The standard header code puts the address of your code resource in register `A0` and then branches to your `main()` routine, but the file containing `main()` is *not* guaranteed to be the first file in the code resource. You can do anything you like with the unused words.

Note

Older versions of THINK C put the address of your code resource in the low memory global `ToolScratch`. The standard code resource header does not do this. Use inline assembly if you need to reference `A0` directly.

If the Custom Header option is checked, THINK C does not generate the standard header. Instead, your code resource starts with the first function in the file where `main()` is defined. The routine `main()` doesn't have to be the first function in the file, so your resource can begin any way you like. (In fact, `main()` may never be called.) This option is useful for certain types of code resources that must begin with a table of some kind, rather than with code.

When you use a custom resource header, A0 does not contain the address of your code resource. This means that you can't use the RememberA0() macro to set up register A4 to use your code resource globals until you set up A0 to point to your code resource.

The following code shows one way to set up your code resource globals when you use a custom resource header.

Warning

SetUpA4.h generates code, so if you include it at the top of your file, the internal function defined in SetUpA4.h is your header. This is not what you want.

```
/* #includes, globals, declarations */

/* declare main() so we can JMP to it */
extern main();

header()  /* First function in file */
{
   asm {
   DC.L  0  /* header information */
   ...
   /* end of header */

   /* put address of code resource in a0 */
   LEA   header, a0
   JMP   main
   }
}

/* SetUpA4.h generates code, so don't put it */
/* at the top of the file with the other     */
/* headers                                   */
#include <SetUpA4.h>

main()
{
   RememberA0();
   SetUpA4();

   ...

   RestoreA4();
}
```

Merging code resources into files

Unlike other project types, you can build code resources into already existing files. In the **Build Code Resource...** dialog, check the Merge option, and type the name of the file you want your code resource placed into.

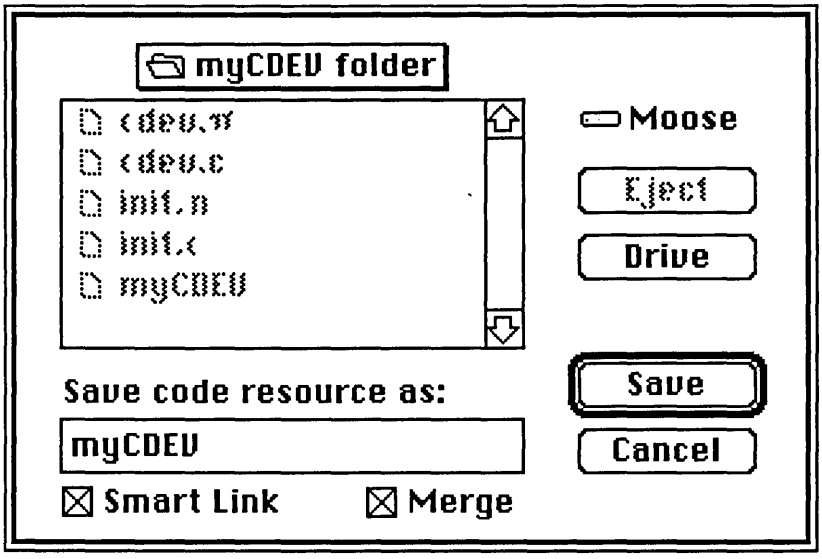


Figure 7-10 The Merge check box in the **Build Code Resource...** dialog

You might want to do this when you're building a library of XCMDs or building a combination cdev and INIT. However, when you check the Merge option, the project's resource file is not be copied into the file, so you should make sure that any resources needed by your code resource are already present in the file.

Multi-segment code resources

If the Multi-Segment option is checked in the **Set Project Type...** dialog, code resources can contain multiple segments. 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:

```
void UnloadA4Seg(ProcPtr);
```

This function works just like `UnloadSeg()` does in applications. To unload all segments at once, call `UnloadA4Seg()` with a null pointer; that is, `UnloadA4Seg(0L)`.

Warning
Do not use `UnloadSeg()` instead of `UnloadA4Seg()` by mistake!

In a multi-segment code resource, the code segment that contains the `main()` function is called owning resource. The other segments are called owned resources. When THINK C builds your resource, it uses the Type and ID number from the **Set Project Type...** dialog for the owning resource. The ID must be 63 or lower. THINK C puts the owned code segments in CCOD resources and numbers them from 0 to 31.

In code resources, the owning segment is special. The DATA and JUMP components for all the segments is appended to the CODE component of the owning segment. So all the DATA and JUMP information and the CODE component of the main segment must be under 32K.

Read the Segmentation section above to learn how to break up a project into different segments.

The Editor 8

Editing and creating files in an integral part of programming. This chapter describes the features of the built-in THINK C editor. The editor uses standard Macintosh editing techniques so you're familiar with its basic operation. Although you can use it to edit any text file, the editor has some features that make editing your source and header files easier.

Contents

Creating and Opening Files

To create or open a file, you must have a project window open. You can open as many files as the memory in your Macintosh allows, and each file appears in its own edit window. Although you usually use the THINK C editor to create and open source or header files, it can open any text file.

Creating a new file

To create a new file, choose **New** from the **File** menu. An untitled edit window appears, ready for you to start typing into it.

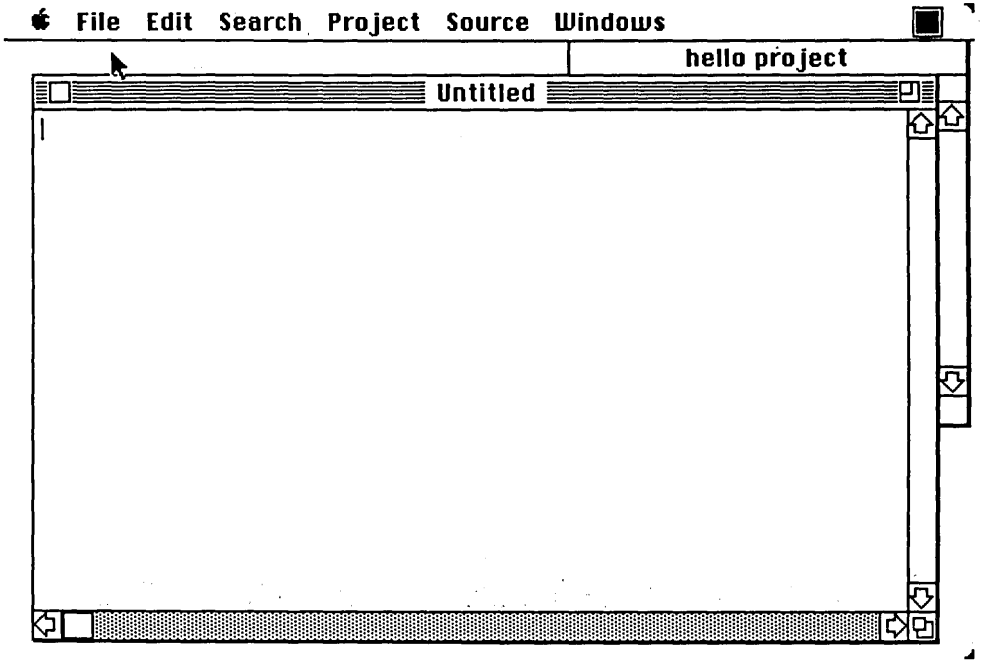


Figure 8-1 An new editing window

Opening a text file

The **Open...** command in the **File** menu opens any text file. A standard file dialog box displays the names of all text files in the current folder, even if they weren't created with THINK C. The file you open appears in its edit window.

Opening a source file

To open a source file that's already in your project window, double-click on its name in the project window. If the file is already open, double-clicking on its name brings the file's edit window to the front.

You can select a file by typing the first few characters of its name when the project window is the active window. You can then open it by pressing Return or Enter. Since the files in the project window are in alphabetical order, the selected file is the first file in the project that matches the characters typed so far. If what you type doesn't match any name in the project window, any selected file is deselected. When what you type matches more than one file name, you can use the Tab key to cycle among all the names that match. The up- and down-arrow cursor keys also change the selection in the project window.

Note

In the project window, you can use Backspace to mean up-arrow, and Shift-Backspace to mean down-arrow.

Opening header files

Sometimes when you're working on a source file, you want to look at the header files associated with it. Naturally, you can use the **Open...** command in the **File** menu to open header files, but THINK C gives you a way to open these files quickly.

Holding down the Command key while clicking in a title bar brings up a menu of markers, described on page 134.

To get a pop-up menu of all the files included in a source file, like in Figure 8-2, hold down the Option key as you click in the title bar of its window. To open a header file in its own edit window, choose its name. If the file is already open, THINK C brings its window forward.

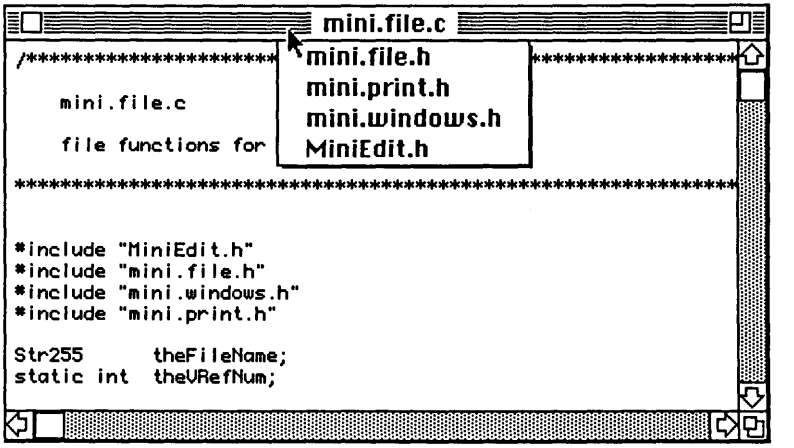


Figure 8-2 The list of a file's header files

Holding down the Option key as you click in the title bar of the project window brings up a pop-up menu containing the names of all the header files used in the project.

To learn more about precompiled headers see "Precompiled Headers" on page 164."

THINK C builds the pop-up menus only for compiled source files. Any header files that are part of a precompiled header don't appear in the pop up menus.

To open a header file that you've added to a source file since the last compilation, or to open a header file for a file you haven't compiled yet, use the **Open Selection** command described below.

Opening the current selection

The **Open Selection** command in the **File** menu lets you open a header file by selecting its name in the current source file (double clicking on the name works here). You don't have to select the `.h` extension. The selection is automatically extended to include it as long as you've selected the first part of the file name.

Open Selection can deal with path names. If the character following the selection is `.` (period) or `:` (colon), the selection is extended to the end of the next word following, and this process is repeated. A partial path name is searched for as though it appeared in an `#include "..."` statement in the file being edited. (**Open Selection** is not smart enough to look for angle brackets.) If the editing window is untitled, only the project and THINK C trees are searched.

To learn about the project and THINK C trees, see Chapter 9, "Files & Folders."

Editing a File

The editor uses all the standard Macintosh editing techniques as well as some designed for editing C programs. Double-clicking on a word selects the entire word. Triple-clicking anywhere in a line selects the entire line. **Cut**, **Copy**, **Paste**, and **Undo** all work the way they do in other Macintosh applications.

Typing text

The THINK C text editor does not have the word wrap feature you might be used to in other editors. If you type past the right edge of the window, use the horizontal scroll bar at the bottom of the window to see past the right edge.

Undoing changes to a file

If you make an unintentional change to your file, use the **Undo** command in the **Edit** menu. **Undo** remembers only the last thing you did. You can also

use **Undo** to redo what you undid. The wording of the **Undo** command always reflects what it will undo. For instance, if you cut a range of lines, the **Undo** command reads **Undo Cut**. If you choose **Undo**, the command now says **Redo Cut**. You can undo the most recent replace (see "Searching and Replacing" on page 139), but you can't undo a **Replace All** command.

If you've made numerous changes to your file, and you want to undo all of them, or if you want to undo a **Replace All**, use the **Revert** command in the **File** menu. Revert discards all of the changes you've made since you last saved (or opened) the file.

Scrolling to the insertion point

The THINK C editor lets you examine an area of your file, then instantly jump back to where you were. To see the current insertion point or the start of the current selection, press the Enter key. Since scrolling in any direction does not affect the insertion point or the current selection, pressing Enter takes you back to your previous position in the file.

If the start of the selection is already visible, pressing the Enter key makes the end of the selection visible, so you can toggle between the two ends of the current selection. This shortcut is particularly useful after using the **Balance** command (see "Balancing parentheses, brackets, and braces" on page 133).

Using the arrow and function keys

The arrow keys move the insertion point up, down, left, and right. At the end of a line, the left (or right) arrow wraps around to the end of the previous line (or beginning of the next line). Pressing the Option key with any arrow key moves the insertion point as far up, down, left, or right as possible. Shift-arrows and Shift-Option-arrows extend the current selection.

This table summarizes how the arrow and editing keys move the insertion point.

Press this...	**To move the insertion point...**
Up-arrow	Up one line.
Down-arrow	Down one line.
Left-arrow	Left one character.
Right-arrow	Right one character.
Command-Left-arrow	Left one word.
Command-Right-arrow	Right one word.
Option-Up-arrow	To the beginning of the file.
Option-Down-arrow	To the end of the file.
Option-Left-arrow	To the beginning of the line.
Option-Right-arrow	To the end of the line.

Note

If you're using a Macintosh Plus keyboard, the editor treats the +, *, /, and = keys on the numeric keypad like Shift-arrow keys. For example, the + key on the keypad is treated as a Shift-Left-arrow. On other keyboards, the +, *, /, and = keys work correctly.

If you have an Apple Extended Keyboard, you can use the keys above the arrow keys. This table describes what they do:

Press this key...	**To do this action...**
Forward Delete (⌦)	Delete the next character.
Home	Scroll to the beginning of the file.
End	Scroll to the end of the file.
Page Up	Scroll to the previous screen.
Page Down	Scroll to the next screen.
Option-Page-Up	Scroll to the left.
Option-Page-Down	Scroll to the right.
Option-Home	Scroll all the way to the left
Option-End	Scroll all the way to the right

Note

The Home, End, Page Up, and Page Down keys do not move the insertion point. They just scroll the file.

Moving to a specific line

The **Go To Line...** command in the **Edit** menu lets you move to a specific line in your file. Lines are numbered consecutively, with the first line in the file being number 1.

When you choose **Go To Line...**, you'll see the dialog in Figure 8-3.

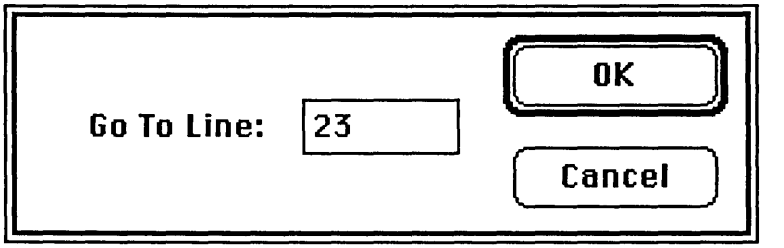


Figure 8-3 The **Go To Line...** dialog

When the dialog appears, the line number specified is the line that contains the insertion point. To go to a particular line, type its number and click OK. If you change your mind, click Cancel. The editor moves the insertion point to the beginning of the line you entered.

Selecting lines

To select a line, triple click anywhere on the line. To select a range of lines, drag the mouse after you triple click.

Indenting

When you press Return to start a new line, the editor indents the new line with the same number of leading tabs and spaces as the previous line has.

To change the indentation, backspace over it. To keep the editor from auto-indenting a line, hold down the Option key when you press Return.

Shifting blocks right and left

To change the indentation level for a range of lines, use the **Shift Left** and **Shift Right** commands in the **Edit** menu. **Shift Left** deletes the leading tab from each selected line. **Shift Right** inserts a tab at the beginning of each selected line. Both **Shift Left** and **Shift Right** extend the current selection to include entire lines.

Warning

Do not type anything while doing **Shift Left** or **Shift Right** on a selected region. Typing replaces the selected text with what you typed.

Balancing parentheses, brackets, and braces

The **Balance** command in the **Edit** menu extends the current selection in both directions until it encloses the smallest block of text enclosed in parentheses `()`, brackets `[]`, or braces `{}`. Successive invocations select larger sequences of text.

Note

The **Balance** command looks for any parentheses, brackets and braces. It doesn't ignore text in strings or comments.

You can use **Balance** to check whether all the parentheses, brackets, and braces in your file are properly balanced. Go to the beginning of your file and search for the first left brace (`{`). Use **Balance** and **Find Again** repeatedly until you get to the end of your file.

Changing font and tab settings

When you open a new edit window with the **New** command, the font is 9-point Monaco, and there's a tab stop every four spaces. You can change these settings with the **Set Tabs & Font...** command in the **Edit** menu. When you choose this command, you'll see the dialog in Figure 8-4:

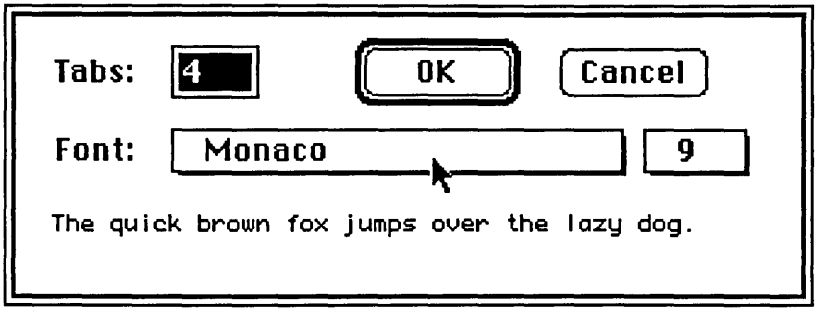


Figure 8-4 The **Set Tabs & Font...** dialog.

Type a number to set the number of spaces per tab. To change the font, click on the font name. You'll see a pop-up menu with the names of the fonts in your System file. Click on the font size to see a pop up menu of the sizes for the font.

If you are using a proportionally spaced font like New York or Geneva, the THINK C editor uses the width of the non-breaking space (Option-Space) to figure out the width of a tab.

When you change the font or tab settings, the editor adds `EFNT` and `ETAB` resources to your text files to record the new settings. Other text editors use these resources as well.

Note

To change the default font and tab settings, use ResEdit to modify the `CNFG 0` resource in the THINK C application. The second, third, and fourth words of this resource specify the font number, font size, and tab width, respectively, used for newly created Untitled windows.

Using Markers

THINK C recognizes the markers that MPW puts in files, and MPW recognizes the markers that THINK C puts in files.

Markers are like bookmarks in a book. They help get to a certain point in your program quickly. You can place markers at the beginning of each routine, or anywhere else in your program.

To put a marker in a file, place the insertion point in the line you want to mark or select some text in that line. Choose the **Mark...** command in the **Search** menu. You'll see the dialog in Figure 8-5.

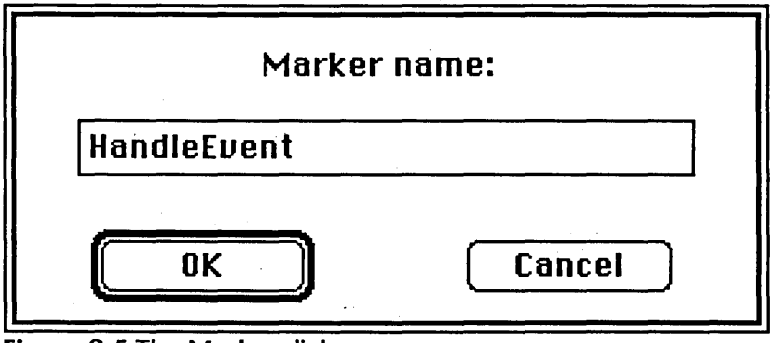


Figure 8-5 The Mark... dialog

If you selected text, the editor uses that text as the default name. If you don't select text, the editor uses the word nearest the insertion point as the default name. Edit the default name, if you want, and click OK. If you don't want to insert a marker after all, click Cancel.

Note

Make sure that your marker has a unique name. THINK C doesn't check whether the name is already used. If several markers have the same name, that name is listed several times in the marker pop-up menu.

Holding down the Option key while clicking in a title bar brings up a menu of the file's header files, described on page 128.

To go to a marker in a file, hold down the Command key and click in the title bar of the file's editing window. A pop-up menu of the file's markers appears, sorted alphabetically, like Figure 8-6. Choose the marker you want to go to, and the editor scrolls to that marker. If you marked a text selection, the editor selects that text. If you marked an insertion point, the editor move the insertion point there.

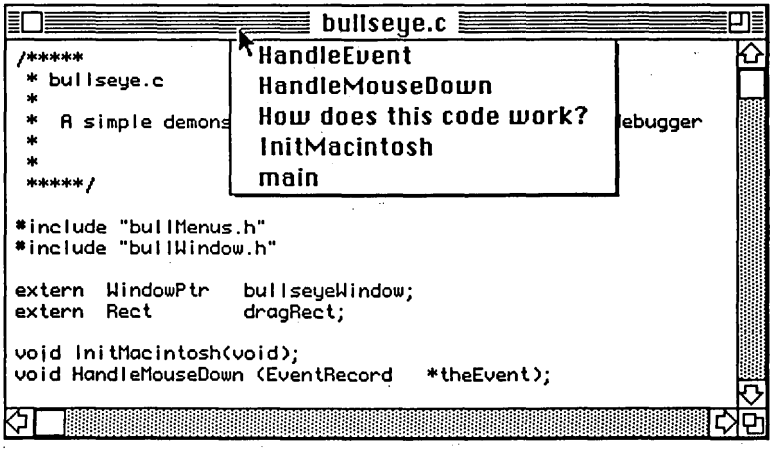


Figure 8-6 The marker pop-up menu

To delete a marker, choose the **Remove Marker...** command in the **Search** menu. You'll see the dialog in Figure 8-7.

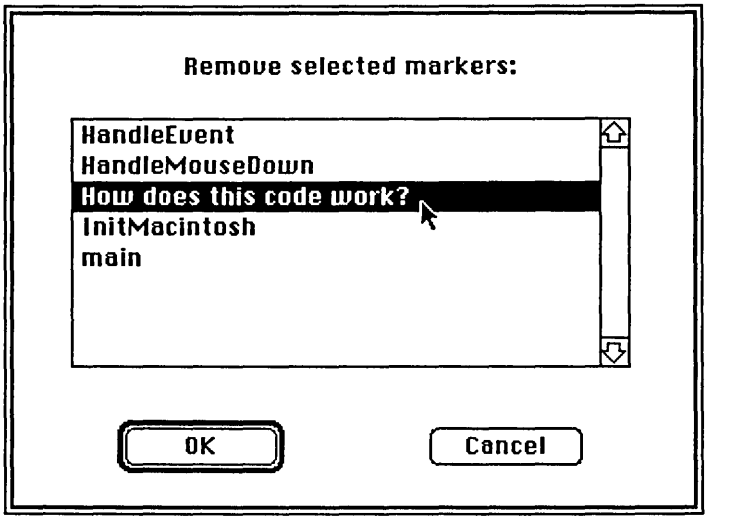


Figure 8-7 The Remove Marker... dialog

Select the markers you want to delete and click OK. If you change your mind, click Cancel. The next time you see the marker pop-up menu, you'll notice that the deleted markers are gone.

Using MPW Projector with THINK C

To learn how to use the Options... dialog, see "Using the Options... Dialog" on page 179.

You can work on a large project with people who use MPW Projector. If the "Projector-aware" option is on in the Preferences section of the **Options...** dialog, THINK C checks for a Projector resource (`CKID` resource) every time it opens a source file. The "Projector-aware" option is one by default. THINK C displays an icon in the lower left corner of the editing window that tells you whether Projector marked the file as read-only. If the file is a read-only file, its window has a crossed-out pencil icon. If it's not a read-only file, its window has a regular icon. If it's a modified read-only file, its window has a pencil icon with a dotted cross-out.

For example, let's say you opened the files in Figure 8-8 with this option on. `FileA.c` doesn't have a Projector resource. `FileB.c` isn't read-only. `FileC.c` is read-only. `FileD.c` is modified read-only.

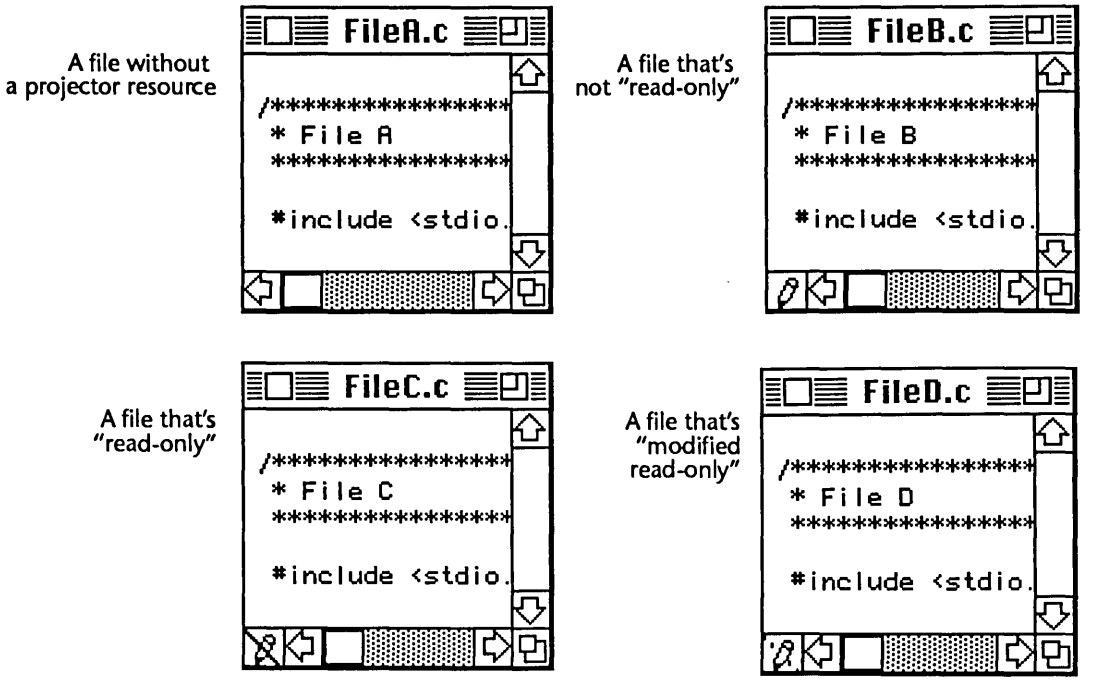


Figure 8-8 Editing files marked with MPW Projector

You can't edit a file marked read-only when the "Projector-aware" option is on. THINK C lets you select and copy text, but ignores your typing and disables the **Cut** and **Paste** commands. If you want to edit a file marked read-only, you have two choices:

- You can make a copy of it. Choose **Save As...** and enter a new file name in the standard file dialog. THINK C copies the file's text, but not its Projector resource (or anything else in the resource fork, for that matter).
- You can make it a modified read-only file. Choose **Modify Read-only** in the **File** menu. Its icon changes to a pencil with a dotted cross-out, and you can edit it.

Printing Files

Use the **Print...** command in the **File** menu to print the file in the frontmost edit window. You'll see the standard print dialog for your printer.

The **File** menu also contains a standard **Page Setup...** command that lets you set the page size and other options before you print.

Closing and Saving Files

It's a good idea to save your work every fifteen minutes or so just in case something horrible happens. Power failures and strange machine crashes occur at the most inopportune times.

Closing a file

To close a file, click on its edit window's close box. If you've made changes, and haven't saved the file, the editor asks you if you want to save it before closing. You can also use the **Close** command in the **Edit** menu to close a file.

Saving a file

To save a file without closing it, use the **Save** command from the **File** menu. If you've never saved the file before (that is, if its edit window is Untitled), you'll get a standard file dialog asking you to name the file.

Saving a file with a different name

The THINK C editor gives you two ways of saving a file under a different name. The **Save As...** command asks you to name a new file and then saves the contents of the edit window under that name. If the file is part of your project (it's in your project window), its name changes there, too.

The **Save a Copy As...** is similar to the **Save As...** command except that it doesn't change the name of the file. This command saves the contents of the current edit window under a new name, but lets you continue editing the original file under its original name. The project window remains unchanged. This is a good way to make backup copies without mistakenly editing the backup.

Saving and closing all open files

To save all the open files, use the **Save All** command in the **Windows** menu. Using this command is the same as using the **Save** command on each window.

The **Close All** command closes all the open files. If a file hasn't been saved, the editor asks you if you want to save it.

Saving files automatically

*For more information on the **Options...** command, see "Using the Options... Dialog" on page 179.*

THINK C saves files for you automatically when you close them if you uncheck the "Confirm saves" check box in the Preferences page of the **Options...** command.

Searching and Replacing

The THINK C editor offers a wide range of search and replace capabilities. You can:

- find a string in a file
- replace one string with another
- find a string in any file in your project
- find the definition of a symbol
- find strings that match a pattern (grep)

Finding a string

Use the **Find...** command in the **Search** menu when you want to find a string. You'll see the dialog box in Figure 8-9.

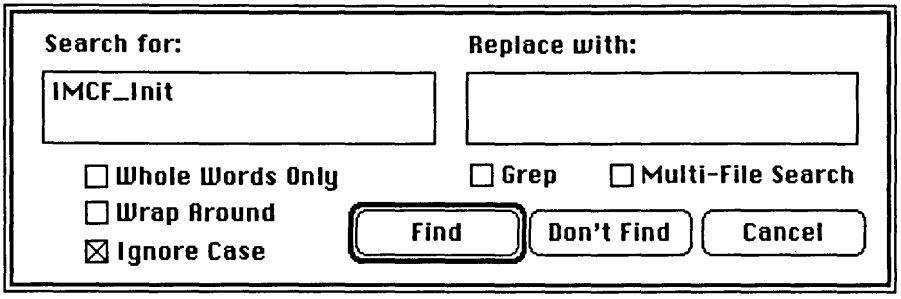


Figure 8-9 The **Find...** dialog

Type the string you're looking for in the "Search for" field and click the Find button. If the string is in the file you're editing, the editor scrolls to where it is and highlights it. If it's not, the editor just beeps.

Since the string you've found is highlighted, you can replace it just by typing in a replacement string.

To find the next instance of the string, use the **Find Again** command in the **Edit** menu.

Search options

The three check boxes on the left side of the **Find...** dialog let you specify how the editor looks for your string.

If you check the "Whole Words Only" option, the editor matches only whole words. This option is useful when you're looking for a one-letter variable name or a word that commonly appears inside other words, like `handle`.

The editor usually searches from the insertion point to the end of the file. If you check the "Wrap around" option, it searches the entire file for your string. The search begins from the insertion point (or the end of the selection). If the editor doesn't find your string by the time it reaches the end of the file, it searches from the beginning to the insertion point.

When the "Ignore case" option is checked, the editor matches the search string regardless of case. If this option is off, the case of the strings must match exactly.

For more information on the Options... command, see "Using the Options... Dialog" on page 179.

You can set the defaults for these options in the Preferences page of the **Options...** dialog in the **Edit** menu. When you set the options in the **Find...** dialog, they apply only to the current session. When you set the options in the **Options...** dialog, they are the permanent defaults for the current project or for all new projects.

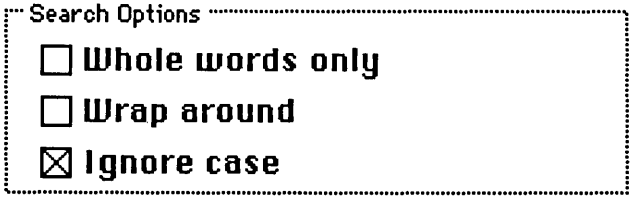


Figure 8-10 The Search Options in the **Options...** dialog

Replacing a string

If you want to replace some but not all instances of the search string, enter a replacement string in the Replace with field of the **Find...** dialog box. When the editor finds the first occurrence, use one of these commands:

- **Find Again** to go on to the next instance
- **Replace** to replace it with the replacement string
- **Replace and Find Again** to replace the current instance and then immediately go on to the next one.

Replace All replaces every instance of the search string in the file with the replacement string. If you don't type in a replacement string, it deletes every instance of the search string (that is, it replaces it with nothing).

Setting things up for searching later

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 replace strings and the option settings without doing the search. To go ahead with the search later, use the **Find Again** command

You might use this button when you realize that the insertion point is not where it should be to start the search. Click on the Don't Find button, move the insertion point to the proper place, then use **Find Again** to find your string.

Note

The **Find Again** command looks for the string you've entered in the search string.

Finding non-printing characters

To look for tab and return characters, hold down the Command key as you type them into the Search for and Replace with fields. To insert other non-printing characters, use the **Copy** command to copy them into the Clipboard, then **Paste** them into the Search for and Replace with fields.

A return signifies the end or beginning of a line in a string search.

Searching through multiple files

The Multi-File Search option lets you look for a string in more than one file in your project. This feature is useful when you're tracking down undefined or multiply defined symbols, or if you change the number of parameters to a function, and you need to fix up all the references to it.

To look for a string in more than one file, check the Multi-File Search check box in the **Find...** dialog box. When you check this box, another dialog box displays all the text files associated with the project.

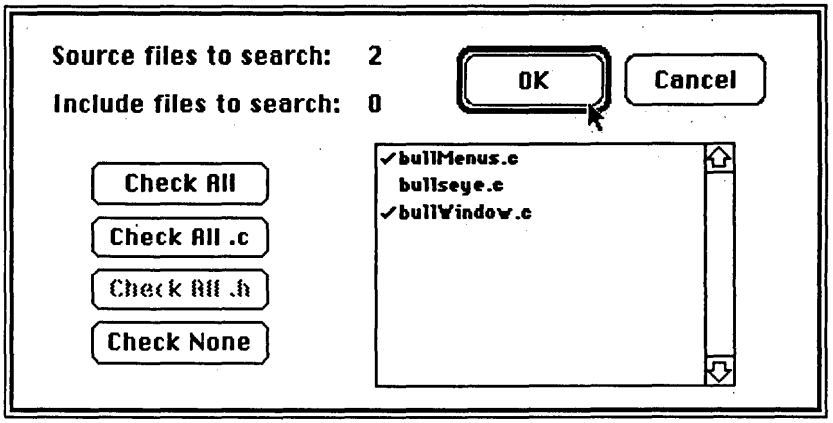


Figure 8-11 The Multi-File Search dialog

Scroll through the list and click on individual files to select them. A small check mark appears next to the file name.If a file is already selected, clicking on its name removes the check mark.You can use the buttons in the dialog box to Check All, Check None, Check All .c, or Check All .h files. You can also use these Command-keys instead of the button:

Button	**Command-key equivalent**
Search All	Command-A
Search All .c	Command-S
Search All .h	Command-H
Search None	Command-N

When you've checked the files you want to search, click OK to return to the **Find...** dialog box., then click Find to start the search. If you didn't select any files, the Multi-File Search option isn't checked.

THINK C looks for the search string in each of the checked files, starting with the first one checked. When it finds a file that contains the search string, it opens the file and selects the search string. At this point, you can edit the file, or, if you want to search further in the current file, you can use the **Find...**, **Find Again**, **Replace**, and **Replace All** commands to work within the current file. When you're ready to go on to the next file, use the **Find in Next File** command.

Note

Multi-file search finds the first instance of the search string in each file. To find more instances of the search string in the file, use the **Find Again** command. Once you issue a **Find in Next File** command, THINK C looks in the next file you checked, even if there are additional instances of the search string in the current file.

Here's another example of when you'd want to use the Don't Find button. Suppose that in a multi-file search, you decide that you really want to set the Match Words option. Open the **Find** dialog, and change the option. If you were to click on the Find button, the search would go on to the next file. So you click on the Don't Find button, and continue using the **Find Again** and **Find in Next File** commands.

Disabling multi-file search

Entering a new search string cancels a multi-file search. To enter a new search string without cancelling a multi-file search, bring up the **Find...** dialog. Click on the Multi-File Search check box. The list of all the text files appears. Click OK to accept all the checked files, and then enter the search string in the "Search for" field.

To cancel a multi-file search without going through the multi-file selection dialog, hold down the Option or Command key as you click the Multi-File Search check box.

Finding the definition of a symbol

There's a shortcut for finding the definition of a function, class, or global variable used in a source file. While holding down the Option or Command key, double-click on the name of function, class, or global variable. THINK C opens the file that defines it and moves the definition. If it doesn't exist or is defined in a library, THINK C beeps.

Note

If THINK C doesn't bring you to the definition of the symbol, use the **Find Again** command until you find it.

You can use this shortcut to find a method, too. Just Option-double-click (or Command-double-click) on a method name. If more than one class defines a method for that message, THINK C displays the Class Browser and highlights the classes that define a method for that message. Double-click on the class you want. THINK C opens the file that defines it and moves to the defi-

nition. For more information on the Browser, see *Object-Oriented Programming Manual*, Chapter 6, "The Browser."

Searching for a Pattern (Grep)

In addition to the search and replace functions described in the previous section, the THINK C editor also provides a powerful pattern search capability called Grep. The Grep search option in THINK C is based on the Grep utility on Unix systems. If you're familiar with this kind of pattern matching you'll find an old friend here. If pattern searching is new to you, experiment with this feature before you use it on a real file.

Warning

The editor looks for patterns only when the Grep option is on.

Patterns

A pattern is a description of a set of strings rather than a specific string. For example, you can build a pattern that means "any word that begins with P." Or a pattern that means "any function call with `&event` as an argument."

Patterns can't span lines. So you can't write a pattern that means "three consecutive lines that begin with a, b, and c."

Simple patterns

The simplest patterns match a single character.

1. Any character, with the exceptions noted below, is a pattern that matches itself.

For example, the pattern 2 matches a character *2*. If you've checked Ignore Case in the **Find...** dialog box, any letter matches both its upper- and lowercase equivalent. So, either `a` or `A` matches both *a* and *A*.

2. The character `.` is a pattern that matches any character.

3. The character `\` followed by any character except `( ) < >` or one of the digits `1-9` is a pattern that matches that character.

For example, `\.` matches a `.` and `\\` matches a `\`.

4. A string of characters `s` surrounded by `[` and `]` is a pattern `[s]` that matches any one of the characters in the string `s`. The pattern `[^s]` matches any character that is *not* in the string `s`. If `s` is

string of three characters in the form a-b, this represents all of the characters from a to b inclusive. All other characters in s are taken literally. To include the character] in s, you must use it as the first character. To include the character - in s, you must use it at the beginning or at the end of s.

For example, the pattern `[A-Za-z0-9]` matches any alphanumeric character. The pattern `[^!-~]` matches any non-printing ASCII character. The Ignore Case option has no effect on what's between brackets.

Complex Patterns

To match strings, not just individual characters, you need patterns that match consecutive sequences of characters. One way of doing this is to append a * to the end of one of the simple patterns.

1. A pattern x followed by a * is a pattern x* that matches zero or more consecutive occurrences of characters matched by x.

For example, the pattern `@*` matches a string containing any number of at-signs. If the string does not begin with an at-sign, or if it contains no at-signs at all, then the pattern matches the empty string at the beginning of the string to be matched. You'll see later on why this is useful.

You can put patterns together to form more complex patterns:

2. A pattern x followed by a pattern y forms a pattern xy that matches any string ab, where a matches x and b matches y.

For example, the pattern `P.` matches any two-letter string consisting of P and any other character.

3. You can concatenate the compound pattern xy with another pattern z, forming the pattern xyz.

To put all of this together, consider the pattern `(.*)`. This pattern matches any string enclosed in parentheses. This includes the string `()`, since the sub-pattern `.*` matches the empty string between the (and the).

Does this pattern match the string `(())`? Since the sub-pattern `.*` matches any number of occurrences of all characters, won't the pattern match just the `(()` and not the very last)? The answer is any sub-pattern of the form x* in a pattern x*y matches the largest number of occurrences of whatever x matches that still allows a match to y. In matching `(())` against the pattern `(.*)`, only the inner pair of parentheses matches the sub-pattern `.*`, so the pattern matches `(())`.

Sub-patterns

Grep has a way of remembering sub-patterns so you can use them again as part of even more complex patterns. Things get a little complicated here.

1. A pattern surrounded by `\(` and `\)` matches whatever the sub-pattern matches.

For example, `\(a[b-y]z\)` matches the same thing as `a[b-y]z`.

2. A `\` followed by `n`, where `n` is one of the digits `1-9`, matches whatever the nth `\(` `\)` sub-pattern matched. You can add a `*` to a `\n` pattern to form a pattern `\n*` that matches zero or more occurrences of whatever `\n` matched.

For example, to find two repeated words (like "the the") you might use a pattern like this: `\([a-z][a-z]*\) \1`. This pattern matches a space, any sequence of letters, a space, and the same sequence of letters. Note that `\1` is not a reapplication of the pattern. Instead it becomes whatever the first `\(` `\)` pair matched.

Constraining patterns

Finally, you can constrain patterns to match only if they meet certain conditions in the context outside the string.

1. A pattern surrounded by `\<` and `\>` matches whatever the pattern matches, provided that the first and last characters of the matched string match `[A-Za-z0-9_]` and that the characters immediately surrounding the matched string don't match `[A-Za-z0-9_]`. In other words, the pattern matches only if the string begins and ends on a word boundary. If you've checked Match Words in the **Find...** dialog box, the entire pattern you enter is treated as though it were surrounded by `\<` and `\>`.

For example, to find occurrences of repeated words even if they're not surrounded by spaces, you would use the pattern
`\(\<[a-z][a-z]*\>\)[^a-z]*\1`

2. A pattern `x` preceded by a `^` forms a pattern `^x`. If `^x` is not preceded by any other pattern, it matches whatever `x` matches as long as the first character `x` matches occurs at the beginning of a line.

3. A pattern `x` followed by a `$` forms a pattern `x$`. If the pattern `x$` is not followed by any other pattern, it matches whatever `x` matches as long as the last character that `x` matches is at the end

of a line. If the pattern `x$` is followed by another pattern, then the `$` is taken literally.

These last two items constrain pattern matches to begin or end at line boundaries, and can be combined to constrain a pattern to match an entire line only.

Replacing with Grep

You can use Grep not only to search for strings, but also to replace them. The following special characters let you alter the replacement string.

1. Each occurrence of the character `&` is replaced with whatever the entire pattern matched.

For example, if you wanted to add a P to the beginning of every word that ended with `ptr`, you would search for `\<.*ptr\>` and replace it with `P&`.

2. Each occurrence of `\n`, where n is one of the digits 1-9, is replaced by whatever the nth occurrence of `\(` matched

For example, to change all strings like `#define FOO 1` to `FOO = 1`, search for:

```
#define \(\<[A-Za-z0-9][A-Za-z0-9]*\>\)
\(\<.*\>\)
```

and replace it with

```
\1 = \2
```

3. Each occurrence of a string `\x`, where `x` is not one of the digits `1-9`, is replaced by `x`.

Grep Examples

Grep is not easy to learn. To give you a hand, here are some typical examples.

Suppose that you've written a Macintosh application, and you've forgotten to put a `\p` at the beginning of your strings to signal to the compiler to make them Pascal strings rather than C strings. You can change all your C strings to Pascal strings by specifying

```
"\([^"]*\)"
```

as the search pattern and

```
"\\p\1"
```

as the replacement string.

To convert

```
symbol  equ  (expression+4); a comment
```

to

```
#define symbol (expression+4)/* ; a comment */
```

search for

```
\(\<.*\>\)[space tab]*\<equ\>\([^;]*\)\(.*\)
```

and replace with

```
#define \1 \2 /* \3 */
```

Explanation:

- `\<.*\>` matches a symbol.
- The surrounding `\(` and `\)` lets you use the symbol in the replacement string as `\1`.
- The `[space tab]*` matches any number of spaces or tabs between the symbol and the key word equ. (The words space and tab stand for the characters here because you can't see them on paper. To enter a Tab, type Command-Tab in the dialog box.)
- `\<equ\>` matches the word equ. It des not match equ if it is part of another word, for example equal. `\<equ\>` is not surrounded by `\(` and `\)` because it's thrown away in the replacement string.
- `[^;]*` matches an expression formed by any number of characters up to but not including a ; (semi-colon).
- The surrounding `\(` and `\)` lets you use the expression in the replacement string as `\2`.
- The `.*` matches the comment which is the rest of the line.
- The surrounding `\(` and `\)` stores the comment as `\3`.
- If there was no ; (semicolon) in the line, then `\2` consists of everything after the equ to the end of the line and `\3` is an empty string.

To convert `$HHHH` to `0xHHHH`, where `H` is a hexadecimal digit, Grep search for

```
$\([0-9A-Fa-f][0-9A-Fa-f]*\)
```

and replace with

```
0x\1
```

Explanation:

- `$` matches a `$`. `[0-9A-Fa-f]` matches one hex digit.
- `[0-9A-Fa-f][0-9A-Fa-f]*` matches one or more hex digits. (The pattern `[0-9A-Fa-f]*` matches zero or more hex digits.)
- The surrounding `\(` and `\)` lest you remember the hex digits in the replacement string as `\1`.

Note

Save complicated Grep search and replace strings in a file so you can copy and paste them into the **Find...** dialog box.

Files & Folders 9

Organizing your files and folders correctly helps THINK C work faster. This chapter tells you how to set up your files and folders, how THINK C looks for #include files, and what happens when you move your files from one folder to another or from one machine to another.

Before you begin

Make sure that you followed the installation instructions in Chapter 2, "Installing THINK C 5.0." If you didn't, go back now, and check to make sure that your disk is set up correctly. This chapter explains why your disk should be set up this way.

Contents

The THINK C and Project Trees

THINK C works with many files at the same time. Most programs have several source files and header files, and virtually every Macintosh application needs to use the `MacTraps` library and some of the standard #include files. THINK C needs to be able to keep track of where these files are on your disk. At the same time, you have to be able to organize the files in your project in a way that makes sense to you.

A full pathname describes exactly where on a disk a file is. For example, the name `Hard Disk:Folder 1:Folder 2:MyFile.c` *says the file* `MyFile.c` *is in the folder called* `Folder 2` *which is in the folder* `Folder 1` *on the disk called* `Hard Disk`.

Instead of using full pathnames to keep track of where a file is, THINK C looks for your files either within the folder that contains the THINK C application or within the folder that your project is in.

THINK C treats all the files in all the folders within the `THINK C Folder` as if they were in the same flat folder. The `THINK C Folder` and all the subfolders in it are called the **THINK C Tree**.

THINK C treats all the files in your project folder as if they were all in the same folder. The folder and subfolders your project is in is called the **project tree**.

This organization lets you move files in and out of folders and even rename folders without having to let THINK C know exactly where the files are. THINK C notes which tree a file belongs to when you add it to your project. If you move files later on, THINK C looks in the appropriate tree to find it.

How THINK C Names Files

When THINK C displays a file name in an edit window's title or in a **Get Info...** dialog, it uses these naming conventions to let you know which tree the file is in:

If the name looks like...	**It's in this tree**
`<filename>`	THINK C Tree
`filename`	project tree

If a file isn't in the THINK C Tree or in the project tree, THINK C displays an absolute name for it:

`vol:filename`	top level folder
`vol:folder:filename`	subfolder of top level folder
`vol:...:folder:filename`	deeper subfolder
`vol:?:filename`	subfolder on unmounted volume

How THINK C Looks for Header Files

These are the rules THINK C uses to find header files:

#include statement	THINK C looks here
`<filename.h>`	THINK C looks only in the THINK C Tree
`"filename.h"`	THINK C looks first in the referencing folder, then in the project tree, and finally in the THINK C Tree.

The referencing folder is the folder that contains the file that has the `#include` preprocessor directive. For example, if a source file references a header file `MyUtils.h`, and that file in turn has the line `#include "MyUtilTypes.h"`, THINK C will look for `MyUtilTypes.h` in the folder that contains `MyUtils.h` first.

Once-only Headers

You may want to create a header file that you want included in several places but which should define its symbols only once in a project. You can use the `#pragma once` directive to do this.

If you have the directive

```
#pragma once
```

in your header file, THINK C will include that file only once. If another file tries to include that header file, THINK C knows that the symbols in that file have already been defined, so it doesn't process the file again.

Note

THINK C 4.0 used a similar mechanism. It would not include a header file if the symbol `_H_`*fname* was defined, where *fname* was the name of the file (in lowercase, without the `.h`). In fact, the `#pragma once` directive does exactly the same thing. The `#pragma once` directive has the advantage that you can rename the header file without changing the directive.

Shielded Folders

To shield a folder from either search tree, enclose its name in parentheses. For example, you might have a folder in the project folder named `(Backups)`. THINK C ignores all the files and sub-folders in shielded folders. You can use shielded folders to store old versions of source or header files or to

keep THINK C from wasting time looking in folders that contain other kinds of documents like development notes.

Project Specific Folders

There is one exception to the shielding rule. If the folder your project is in contains a folder that has exactly the same name as your project surrounded by parentheses, THINK C *will* search that folder.

You can use this feature if you're working on two projects that share files. For instance, suppose you're working on two projects, `INITProject` and `cdevProject`, that share some source files and are in the same folder. You create two folders, `(INITProject)` and `(cdevProject)`, that both contain versions of the header file `config.h` tailored to control conditional compilation of the common source files.

Note

In THINK C 5.0, a better way of accomplishing the same thing is to use the Prefix page of the **Options...** menu.

Using Aliases

THINK C lets you work with the alias of a project file. The project tree begins where the original project is. However, THINK C does not support aliases in these cases:

- Putting aliases in a project
- Including aliases in an `#include` statement
- Using an alias as a project's resource (`.rsrc`) file
- Inserting an alias of a folder in your THINK C tree or project tree.

Using the Trees

The way THINK C keeps track of your files lets you organize your files just the way you like without having to specify full path names. There are a few points you should remember about using the THINK C and project trees.

Don't put project folders in the THINK C Tree

This is the most common mistake. It seems natural to put all your THINK C files in one folder and then toss your project folders in there as well. If you set up your disk like this, THINK C will search *all* your other project trees every time it searches the THINK C Tree. Setting up your project folders this way not only increases search times, it also makes it more likely that you'll duplicate names within trees.

Avoid duplicate file names in trees

Just as you can't have two files with the same name in the same folder, you shouldn't have duplicate file names in different folders within the project or THINK C Tree. If you do, THINK C won't know which file to use. Duplicate file names won't lead to any explicit errors, but you may end up using the wrong file.

It's OK to have the same file name in both the project and THINK C trees. THINK C resolves the conflict deterministically by search order.

Warning

The **Save As...** command copies files, so if you use it to save a copy in another folder, be sure to remove or rename the original file. See section "Moving Files Within a Project" on page 157.

Organizing Your Files

When you're working on a small program, it makes sense to keep all the source files and all the header files in the same folder like this:

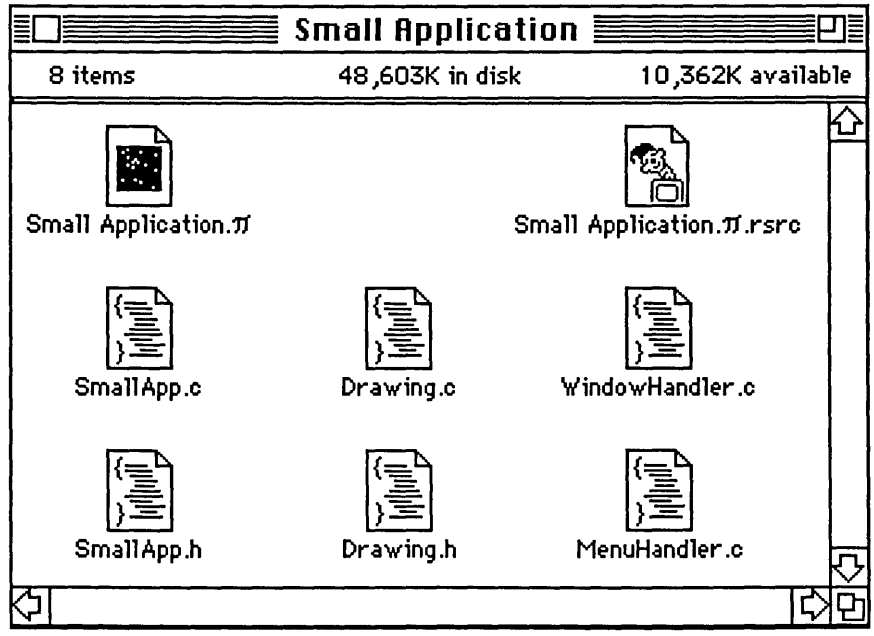


Figure 9-1 Organizing files for a small application

In the project in Figure 9-1, THINK C looks for all the project files in the folder called `Small Application`.

As your application grows, you might want to distribute your source and header files among several folders like this:

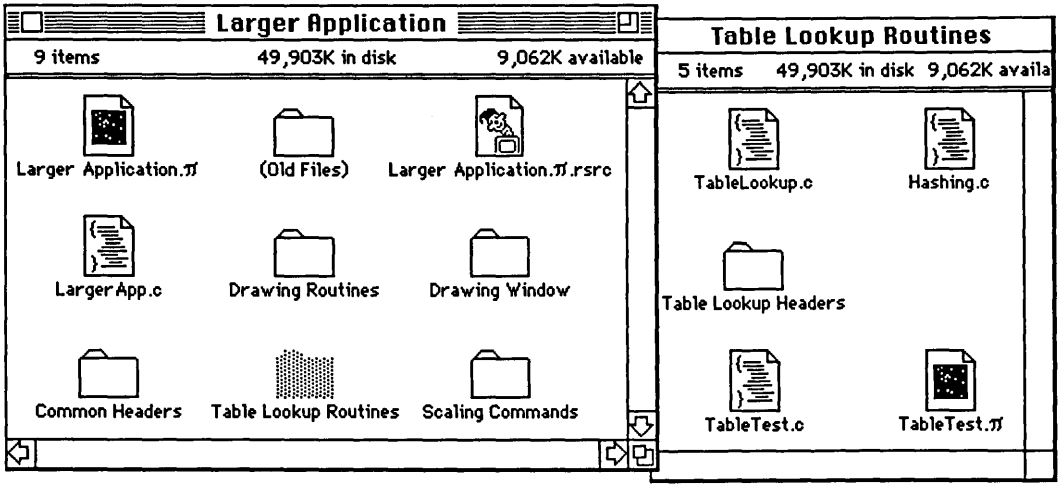


Figure 9-2 Organizing files for a larger application

In Figure 9-2 most of the source and header files are in folders. Since THINK C treats all the files and folders within the `Larger Application` folder as if they were in the same folder, you can put your header files where they make the most sense. The folder called `(Old Files)` is a shielded folder, so THINK C will never look for source or header files there. You can use shielded folder this way to store old versions of files.

Moving Files Within a Project

You can move files freely within a tree. If THINK C can't find a file that's already in your project, it assumes that you've just moved it somewhere within the tree, and tries to find it there.

Moving a source file

To move a file from one tree to the other tree, it's best to use the **Save As...** command in the **File** menu. This command takes care of updating the references to the file in your project document without losing the object code, so you won't have to recompile it.

This is how you move a file from one tree to the other:

1. Open the file you want to move. Double clicking on its name in the project window is the fastest way.
2. Choose **Save As...** from the **File** menu. When the standard file dialog box appears, go to the folder within the tree you want to save the file in, and click on Save.
3. Make sure you delete the file from the original tree. Since **Save As...** makes a copy, the original file is still in the old tree.

Moving a library

Moving a library is a little trickier since you can't open libraries with the editor.

1. Move the library to a folder in the other tree. Use the Finder or a file-moving desk accessory.
2. Choose **Remove** from the **Project** menu to remove references to it from the project.
3. Use the **Add...** command in the **Source** menu to put it back into the project.

Moving other files

From time to time your project may refer to files and libraries outside the THINK C or project trees. To move these files, use the same procedure as for libraries above.

Moving files to another machine

When you move a project to another Macintosh, use the Use Disk button in the **Make...** dialog to let THINK C find all the files. Files don't need to be in exactly the same place on the two machines as long as they're within the project or THINK C Trees. Files outside either tree must have the same absolute pathname on the two machines.

A note about search times

Searching the trees after you've moved files around can take a little time. Once THINK C finds a file, though, it remembers where it is and looks there first the next time. So if the compiler seems slower than usual, don't worry. Once THINK C learns where your files are, it will speed up again.

Disk Layout Diagram

This diagram shows the recommended disk layout. You don't have to set up your disk this way, but the important thing to remember is that your project folders should not be in the THINK C folder.

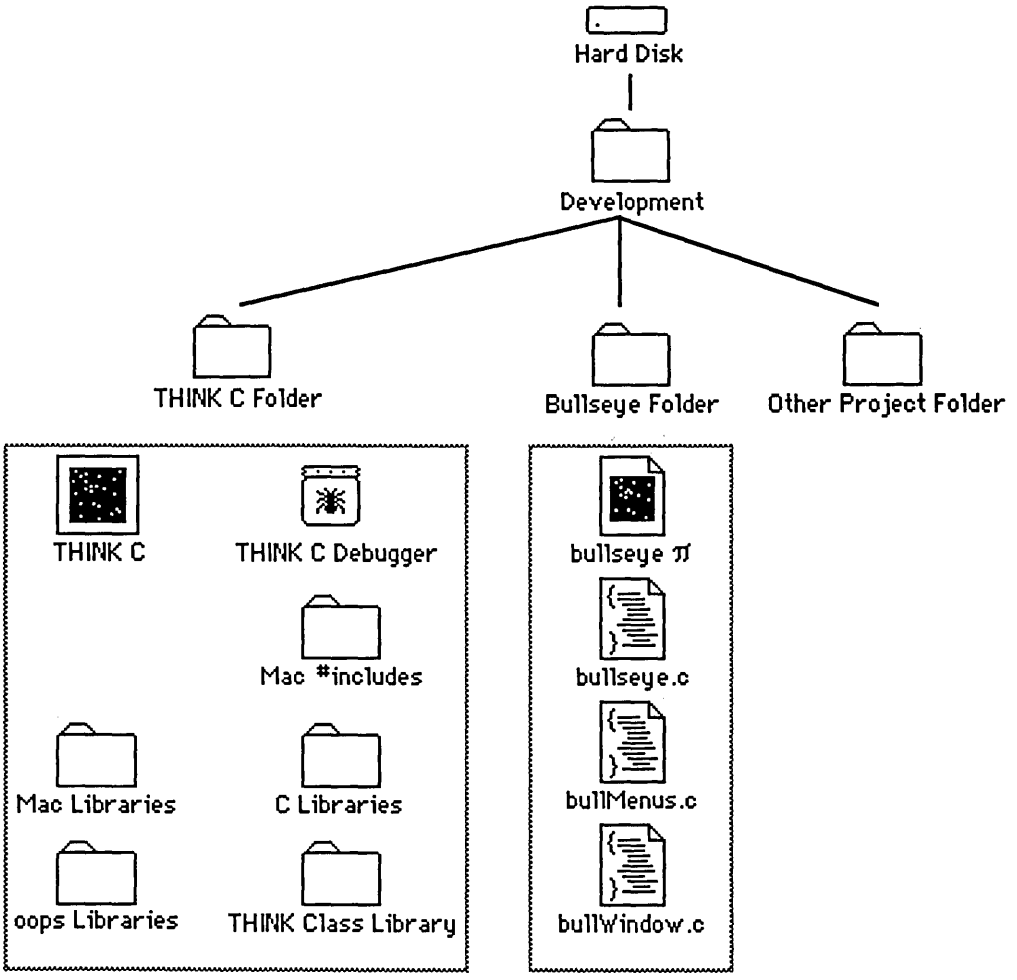


Figure 9-3 Recommended disk layout for THINK C

The Compiler 10

Different compilers implement C differently, even if they conform to the ANSI standard. This chapter explains the unique features of THINK C: how to compile source files, how to use precompiled headers, and how to set options that affect the way THINK C compiles your source files. This also chapter lightly touches on how THINK C complies with the ANSI standard for the C language. For more detailed information, Chapter 22, "Language Reference."

Contents

Compiling Source Files

Unlike traditional compilers, THINK C doesn't generate separate **object files** from your **source files**. Instead, THINK C puts all the object code into the project document. Although you can compile files yourself, most of the time you'll be using the Auto-Make facility to compile your files.

Note

Source files are your program files. Object code is the machine language that the THINK C compiler generates from your source files.

Compiling files not in the project

You can add a source file to your project and compile it in one step. First, create your source file with the THINK C editor. Save your file in the same folder as the project document. Make sure that the file name ends in `.c`. THINK C will only compile files that end in `.c`.

Next, choose **Compile** from the **Source** menu. A dialog box shows you how many lines THINK C has compiled. If there were no errors in the source file, THINK C adds the file and its object code to the project.

Compiling files already in the project

If you want to compile a file that is already in the project, just click on its name in the project window and choose **Compile** from the **Source** menu. Once a file is in the project, you don't need to open it to compile it.

Checking files without compiling

Sometimes you just want to make sure that your source file will compile without actually compiling it. The **Check Syntax** command in the **Source** menu checks the syntax of the contents of the frontmost edit window without generating code or adding the file to the project window. In fact, you don't even have to save the file first.

Fixing errors in source files

When THINK C detects an error in your source file, it opens the source file and displays a bug alert.

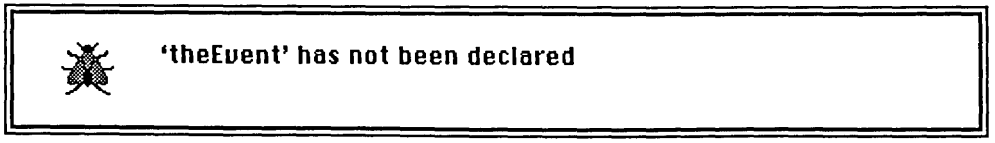


Figure 10-1 A bug alert

To get rid of this alert box, click anywhere in it or press the Return or Enter key.

The source file that contains the error will be in an editor window with the line that contains the error highlighted.

Precompiled Headers

THINK C lets you **precompile** header (#include) files. Precompiled headers contain only declarations and preprocessor symbols. Since precompiled headers are in a format THINK C can use readily, they load faster than text header files.

Note

If you're using the source level debugger, you should use precompiled headers. Precompiled headers help make the debugger tables smaller.

THINK C comes with one precompiled header file, `MacHeaders`, which contains the most common declarations you use for writing Macintosh programs. If you #include `MacHeaders` in the project prefix, THINK C automatically includes `MacHeaders` in all the files in your project, so you never have to explicitly #include common header files like `QuickDraw.h`. (It doesn't hurt if you do.) To edit the project prefix, go to the Prefix page of the **Options...** dialog, described on page 181.

Note

The project prefix contains the line `#include <MacHeaders>` by default.

The `MacHeaders` file contains these files:

```
Controls.h          Desk.h
Devices.h           Dialogs.h
DiskInit.h          Errors.h
Events.h            Files.h
Fonts.h             Lists.h
Memory.h            Menus.h
Notification.h      OSEvents.h
OSUtils.h           Processes.h
Quickdraw.h         Resources.h
Scrap.h             SegLoad.h
```

```
StandardFile.h          TextEdit.h
Timer.h                 ToolUtils.h
Types.h                 Windows.h
pascal.h                asm.h
LoMem.h                 THINK.h
```

These files aren't used as often, so they're not included in MacHeaders. You'll have to include them yourself.

```
ADSP.h                  AIFF.h
Aliases.h               AppleEvents.h
AppleTalk.h             Balloons.h
CommResources.h         Connections.h
ConnectionTools.h       CRMSerialDevices.h
CTBUtilities.h          DatabaseAccess.h
DeskBus.h               Disks.h
Editions.h              ENET.h
EPPC.h                  FileTransers.h
FileTrasferTools.h      Finder.h
FixMath.h               Folders.h
GestaltEqu.h            Graf3D.h
HyperXCmd.h             Icons.h
Language.h              MIDI.h
Packages.h              Palettes.h
Picker.h                PictUtil.h
Power.h                 PPCToolBox.h
Printing.h              PrintTraps.h
QDOffScreen.h           Retrace.h
ROMDefs.h               SANE.h
Script.h                SCSI.h
Serial.h                ShutDown.h
Slots.h                 Sound.h
SoundInput.h            Start.h
SysEqu.h                Terminals.h
TerminalTools.h         Values.h
Video.h                 Traps.h
```

Usually, you'll just use the built-in MacHeaders. You can, however, edit the default MacHeaders file or make your own precompiled headers.

Editing the MacHeaders file

You might find that in the kinds of program you write, you frequently refer to a header file that is not already in MacHeaders. Or, MacHeaders might

include some files you never use. You can edit the MacHeaders file to suit the kinds of programs you write.

1. Find the file Mac #includes.c in the Mac #includes folder. Duplicate it and give it a new name, such as My #includes.c. Open the duplicate with THINK C.

2. If you want to change the compile options, edit the #pragma options directive. Otherwise, it uses the settings in the **Options...** dialog and uses different options depending on the project that's open. For more information, see "Accessing Option Settings in Your Code" on page 195.

3. Search for the files you want to add or remove. All the #include statements are enclosed in conditional compilation directives. To add a file, change the #if 0 directive to #if 1. To remove a file, change the #if 1 directive to #if 0. Some files can't be used together. For more information, see the list below.

4. Choose **Precompile...** from the **Source** menu. After THINK C precompiles the file, save it as MacHeaders. (Precompiled files don't go into the project.) The best place for MacHeaders is in the Mac #includes folder, but you can save it anywhere in the THINK C tree.

The auto-make facility marks the files in the current project for recompilation if you change MacHeaders. To let other projects know that MacHeaders changed, use the **Make...** command in the **Source** menu. Click on the Use Disk button to mark all the files affected, then click on the Make button to recompile them.

When you add and remove files from MacHeaders, keep these dependencies in mind:

- You cannot #include both Printing.h and PrintTraps.h.
- You cannot #include both LoMem.h and SysEqu.h.
- If you #include both asm.h and Traps.h, you must #include asm.h before Traps.h. If you #include Traps.h, do *not* prefix traps with underscores in inline assembly.

Creating your own precompiled headers

If you want to use your own precompiled headers, follow these steps:

1. Remove the line `#include <MacHeaders>` from the project prefix, described on page 181.

2. Create a file containing the desired series of `#include` statements. You can #include `MacHeaders` or any other precompiled header in the first line of your precompiled header.

3. If you need to change the compile options, use the `#pragma options` directive. Otherwise, it uses the settings in the **Options...** dialog and uses different options depending on the project that's open. For more information, see "Accessing Option Settings in Your Code" on page 195.

4. Choose the **Precompile...** command from the **Source** menu. When THINK C is through precompiling, it asks you to name the file.

You use a precompiled header the same way you use any other header file. Use the `#include` statement to load it into your source file. The `#include` statement must be the first non-comment line of your source file. You can use only one precompiled header per source file. (If you #include `MacHeaders` in the project prefix, you can't explicitly include any other precompiled header.)

If you don't #include any precompiled headers in the project prefix, you can use several different precompiled headers for different parts of your program, and you can still explicitly include `MacHeaders` if you want to use it in certain files.

Note

You can use only one precompiled header per source file.

THINK C Reports

THINK C lets you look at your source code and finished applications in three different ways. You can see the preprocessor output of a source file, the assembly code a source file produces, and a link map of a finished application.

Viewing the preprocessor output

If you think you have a bug in one of your macros, use the **Preprocess** command in the **Source** menu. It runs the code in the frontmost window through the THINK C preprocessor and displays the result in a new window. The preprocessor expands your macro, includes the contents of your #include files, and evaluates your `#ifdef` statements. You can save and print the contents of this window like you would any other file.

Disassembling your code

Looking at the assembly code that the compiler produces helps you debug your code and figure out how efficient it is. The **Disassemble** command in the **Source** menu disassembles the code in the front-most window and displays the result in a new window. You can save and print the contents of this window like you would any other file

---

Note

The **Disassemble** command shows all references to global variables and subroutine calls as being at `0x0000(A5)`. This value is merely a place holder for the disassembler. When THINK C compiles your program it uses the correct offset.

Generating a link map

THINK C can write a link map for your application. The link map lists all your project's segments, including one for global data. For each function (or global variable) in a segment, the map lists its name, its position in the segment, and the file it's defined in. To generate a link map, turn on the "Generate link map" option in the Preferences page of the **Options...** dialog.

THINK C creates the link map only when you use the **Build...** command. The name of the map is the name of the project with `.map` appended. For example, the link map for the `Bullseye.π` project is `Bullseye.π.map`. THINK C places the link map in the project folder and erases any link map that was there.

This is an excerpt from the global data section of a link map:

```
Segment "%GlobalData" size=$00084E
   appleMenu       -$00084E(A5)    file="bullMenus.c"
   fileMenu        -$00084A(A5)
   editMenu        -$000846(A5)
   widthMenu       -$000842(A5)
   windowBounds    -$0007A8(A5)    file="bullWindow.c"
   circleStart     -$0007A0(A5)
```

Here is how to read it:

- The top line gives the segment's name and size. This segment is named `%GlobalData` and is `0x00084E` bytes large.
- Each of the other lines lists a variable and its address as a negative offset from `A5`. The variable `appleMenu` is at `0x00084E` offset from `A5`.
- At the far right is the name of the file that defines the variable. The variables `appleMenu`, `fileMenu`, `editMenu`, and `widthMenu` are in the file `bullMenus.c`. The variables `windowBounds` and `circleStart` are in `bullWindows.c`.

And this is an excerpt from a code segment:

```
Segment "Seg2" size=$0004FC rsrcid=2
   SetUpMenus          $000004     file="bullMenus.c"
   AdjustMenus         $000092
   HandleMenu          $000186
   InitMacintosh       $000282     file="bullseye.c"
   HandleMouseDown     $0002A6
   HandleEvent         $00036E
   main                $000414  JT=$000072(A5)
```

Here is how to read it:

- The top line gives the segment's name, size, and resource ID. The segment `Seg2` is `0x0004FC` bytes large and is in code resource 2.
- Each of the other lines lists a function name and its offset within the segment. The function `SetUpMenus()` is at offset `0x000004`.
- If a function has a jump table entry, the address of the entry is listed as an offset from `A5`. The function `main()` has a jump table entry at `0x000072` offset from `A5`.

- At the far right is the name of the file that defines the function. The functions `SetUpMenus()`, `AdjustMenus()`, and

`HandleMenus()` are in the file `bullMenus.c`. The functions `InitMacintosh()`, `HandleMouseDown()`, `HandleEvent()`, and `main()` are in `bullWindows.c`.

How THINK C Implements C

This section describes how THINK C implements certain parts of the C language, including how it assigns variables to registers and how it represents integers and floating-point numbers

Identifier length and capitalization

In THINK C, every character in an identifier and its case is significant. For external identifiers, the number of significant characters is 255. For all other identifiers, all characters are significant.

Using register variables

In THINK C, you can decide which variables are placed into registers or let THINK C decide for you. The "Automatic Register Assignment" option in the Code Optimization page of the **Options...** menu controls what happens. If that option is on, THINK C automatically assigns variables to registers to make your code as small as possible. If that option is off, THINK C puts only those variables declared `register` into registers.

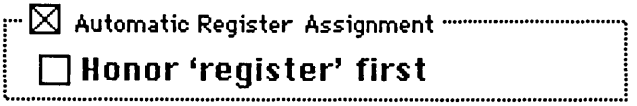


Figure 10-2 The "Automatic Register Assignment" option

You can test and change the settings of these options from your code. See "Accessing Option Settings in Your Code" on page 195 and "Other optimization options" on page 202.

If you select this option, the "Honor 'register' first" option below appears. This table explains what that option does:

If "Honor register first is"...	THINK C...
On	Puts the variables declared `register` in registers and puts the variables it wants in the remaining registers.
Off	Ignores your `register` declarations and puts the variables it wants in registers

When you ask THINK C to honor your register declarations, the following registers are available for register allocation:

Registers	Used for
`A2-A4`	Pointer types
`D3-D7`	Integral types, pointer types, and floats if "Generate 68881 instructions" is on
`FP4-FP7`	12-byte floating-point types if "Generate 68881 instructions" is on

Note

If you're building a desk accessory, device driver, or code resource, register `A4` is not available for register variables. These project types use `A4` to access their globals.

You should not rely on the order in which THINK C allocates variables to registers. THINK C never places a struct or a union in a register, even if it would fit into one.

When you ask THINK C to choose variables to place in registers, THINK C tries to make your code as small as possible. Notice that it doesn't try to save time. For example, it will not make a special effort to place a variable incremented in a loop in a register.

This list explains which option is best for your project:

- If you're porting old code with many `register` declarations, turn on "Automatic Register Assignment" and turn off "Honor 'register' first." THINK C will ignore the declarations and will produce the smallest code possible.
- If you're writing new code, turn on both "Automatic Register Assignment" and "Honor 'register' first." As you write, don't use 'register' declarations unless you're sure you want to override THINK C's assignments. For example, you may want to put a variable incremented in a loop in a register. Since this option optimizes for space and not speed, THINK C may not put that variable in a register. You must declare it `register` to ensure that it is.
- If you need to be absolutely sure what's in each register, turn off "Automatic Register Assignment."

Note

When THINK C compiles a function that contains in-line assembly, it ignores this option's setting and does not automatically assign variables to registers.

Integer representation

This section describes how THINK C implements integers. The limits for the integer types are declared in the header file `<limits.h>` and documented in the *Standard Libraries Reference.*

Integers are represented as 2's complement binary numbers. These are the sizes of integer types:

Type	Bytes
`char`	1
`short`	2
`long`	4
`int`	2 or 4

You can test and change the settings of this option from your code. See "Accessing Option Settings in Your Code" on page 195 and "Type options" on page 197.

You choose the size of `int` with the "4-byte int" option in the Compiler Settings page of the **Options...** dialog. If the option is off, `int`s are two bytes long. If it's on, `int`s are four bytes long.

Use four byte `int`s if you're porting code from a compiler that uses four byte integers, like MPW C, or you're porting code that assumes integers and pointers are the same size. Otherwise, use two byte integers, since the MC68000 family handles them more efficiently.

Note

If you turn on the "4-byte int" option, you must recompile the `ANSI` library if your project uses it.

Floating point representation

This section describes how THINK C implements floating-point numbers. The limits for the floating-point types are declared in the header file `<float.h>` and documented in the *Standard Libraries Reference.*

THINK C gives you these three types of floating-point numbers.

- `long double` is the fastest floating-point type. Use this type most of the time.
- `float` is smaller than double, but also slower. Use this type when you need to save space and can sacrifice some speed and accuracy.
- `short double` is the medium sized float but, strangely enough, is the slowest floating point type. You'll rarely need to use this type.

The THINK C type `double` is usually the same size as a `long double`. However, if the "8-byte doubles" option is on, `double` uses the same representation of `short double`. This option is described in "Porting code from other compilers" on page 193.

Note

The `short double` type is a THINK C extension and is not part of the ANSI standard.

THINK C uses these five different representations for floating point values:

These formats are documented in detail in the Apple Numerics Manual, Second Edition *(Addison-Wesley) by Apple Computer.*

- 4 byte IEEE single precision
- 8 byte IEEE double precision
- 10 byte SANE extended precision
- 12 byte MC68881 extended precision
- 12 byte universal extended precision

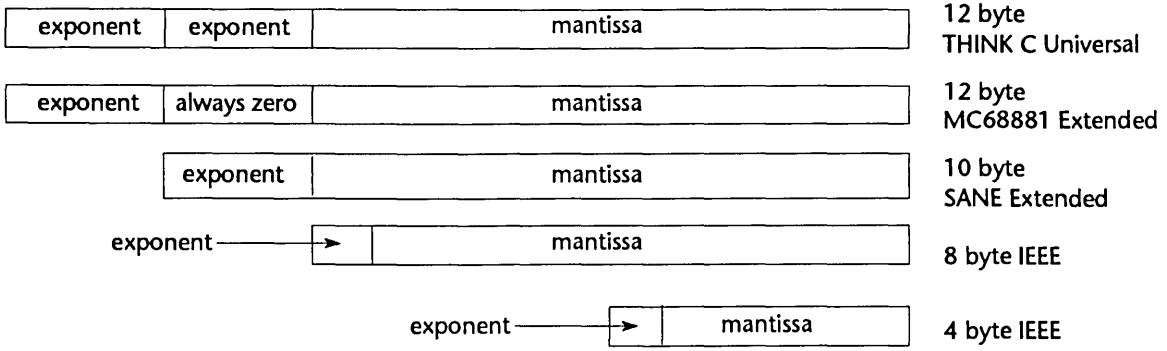


Figure 10-3 Floating-point formats

The Universal format is a THINK C extension that lets THINK C use the same format whether the program uses MC68881 code generation or SANE. The

universal format is the same as the MC68881 extended format, but it duplicates the exponent part in the second word.

The types `float` and `short double` always use the same representation: `float` uses the 4-byte IEEE representation, and `short double` use the 8-byte IEEE representation. However, the representation for `long double` depends on your option settings.

*For more information on the **Options...** dialog, see "Using the Options... Dialog" on page 179.*

To choose a format for `long double`, use these two options in the Compiler Settings page of the **Options...** dialog: "Native floating-point format" and "Generate 68881 instructions." If "Native floating-point format" is off, THINK C uses the Universal format. If "Native floating-point format" is on, THINK C chooses a format according to the setting of "Generate 68881 instructions." This table shows which format THINK C uses:

You can test and change the settings of these options from your code. See "Accessing Option Settings in Your Code" on page 195, "Type options" on page 197, and "Generating processor-specific code options" on page 200

If Native Floating-point is	**and Generate 68881 is**	**Then doubles use this format...**
Off	On or Off	Universal
On	Off	SANE Extended
On	On	MC68881 Extended

If your project contains any libraries or projects that were compiled with the Universal format, use the Universal format. These THINK C libraries use the Universal format:

- `ANSI`
- `profile`

If you're writing a library for general use, use the Universal format. Projects can use your library no matter how they set the "Generate 68881 instructions" option, as long as they turn off the "Native floating-point format" option.

If you want the fastest code possible and use no libraries compiled in the Universal format, use the native floating-point format. The conversions to and from the Universal format are eliminated.

THINK C Extensions

If you're writing ANSI-conformant code, you must disable all the THINK C extensions. For more information, see "Compiling ANSI-Conformant Code" on page 182.

THINK C defines several extensions to the C language that make programming on the Macintosh easier and let you take advantage of object-oriented programming.

Most of the extensions are controlled with the "Language Extensions" option in the Language Settings page of the **Options...** dialog. When you turn that

option on, you can choose between two sets of extensions: "THINK C" or "THINK C + Objects."

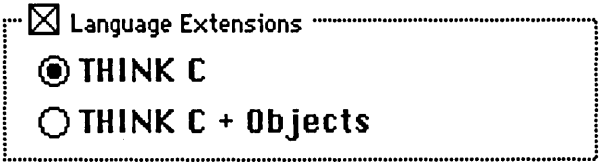


Figure 10-4 The "Language Extensions" option

You can test and change the settings of these options from your code. See "Accessing Option Settings in Your Code" on page 195 and "Language extension options" on page 198.

The "THINK C" setting enables these extensions, which are described below:

- Inline assembly
- `pascal` keyword
- C++ style comments
- `short double` type
- Comments after `#else` and `#endif`
- Inline function definitions
- Low memory global definitions
- MC68881 unary inline functions
- Function prototypes with `(...)`
- Dimensionless arrays
- Using `void *` with function pointers
- The `THINK_C` predefined symbol

The "THINK C + Objects" setting enables all the extensions above plus it lets you write object-oriented code. The object extensions are described in *Object-Oriented Programming*, Chapter 4, "Using Objects in THINK C." If you use the object extensions, these words become keywords, and you can't use them as identifiers:

```
class           delete
new             virtual
```

These words are interpreted specially in context. They are not keywords, but you should avoid using them:

```
direct      indirect      inherited
operator    private       protected
public      this
```

Two other extensions have their own option settings in the Language Settings page of the **Options...** dialog.

This extension...	**is controlled by this option...**
Disabling trigraphs	Recognize trigraphs
enums of any size	enums are always ints

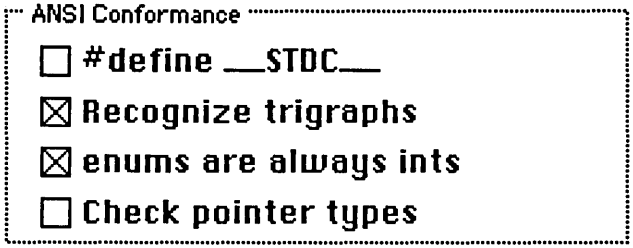


Figure 10-5 Two more THINK C extensions

Disabling trigraphs

You can test and change the settings of this option from your code. See "Accessing Option Settings in Your Code" on page 195 and "Language extension options" on page 198.

Trigraphs are sequences of three letters that are treated as one. The sequence contains `??` and an additional character. Trigraphs let computers without such characters as braces ({, }), tilde (~), and caret (^) use C. However, many Macintosh programs use character literals that resemble trigraphs. For example, the file type `'????'` is interpreted as `'?^`. If you don't want THINK C to recognize trigraphs, turn off the "Recognize trigraphs" option in the Language Settings page of the **Options...** dialog.

Note

To write the file type `'????'` with the "Recognize trigraphs" option on, use `'???\?'`.

---

enums of any size

You can test and change the settings of this option from your code. See "Accessing Option Settings in Your Code" on page 195 and "Enumerated type option" on page 199.

In THINK C, enumeration types can be the same size as a `char`, `short int`, `int`, or `long int`.

THINK C makes an enumeration type as small as possible or as large as necessary. For example, this type will only be as large as a `char`:

```
enum { red=1, yellow, green };
```

And this type will be as large as a `long int`:

```
enum { million=1000000, billion=1000000000 };
```

If you want enumeration types to be any size, turn off the "enums are always ints" option in the Language Settings page of the **Options...** dialog. If you are writing ANSI-conformant code, turn the option on.

*The rest of the extensions in this section are available when you turn on the "Language Extensions" option in the Language Settings page of the **Options...** dialog*

Inline assembly

The identifier `asm` is reserved for the inline assembler. For more information about the inline assembler, see Chapter 13, "Assembly Language."

pascal keyword

The identifier `pascal` is reserved to define functions that follow Pascal calling conventions.

C++ style comments

Two slashes (`//`) introduce a comment. The comment ends at the new-line character.

short double type

The type `short double` is allowed. It is an 8-byte IEEE floating point number.

Comments after #else and #endif

An identifier after an `#endif` or `#else` preprocessing directive is ignored.

```
#ifdef DEBUG
   printf( "oops!\n");
#endif DEBUG
```

Inline function definitions

You can define inline functions using this form:

```
returnType functionName (arguments) =
   { instr1, instr2, ... };
```

THINK C replaces a call to the function *functionName* with the machine instructions *instr1, instr2, ...* instead of generating a function call. For example, the function `PrOpen()` is defined like this:

```
pascal void PrOpen(void)
   = {0x2F3C,0xC800,0x0000,0xA8FD};
```

When you call `PrOpenPage()` THINK C generates these instructions:

```
move.l  #0x10000808, -(SP)
_PrGlue
```

For a detailed description of the #pragma parameter directive see page 194.

You can use the `#pragma parameter` directive be used to assign registers to parameters:

```
#pragma parameter __A0 NewHandleClear(__D0)
pascal Handle NewHandleClear(Size byteCount)
 = 0xA322;
```

Most of the Macintosh Toolbox routines are defined in the header files as inline functions.

Low memory global definitions

Global variables may be defined to refer to absolute memory addresses.

```
extern short MemErr : 0x220;
```

This construct is only legal in global scope. The `extern` storage-class specifier is optional. It was required in earlier versions of THINK C.

MC68881 unary inline functions

When the "Generate 68881 instructions" option is on, you can declare unary inline functions for the MC68881 like this:

returnType functionName (*argument*) : *instr;*

where *instr* is the low seven bits of the second word of the desired MC68881 instruction. This syntax works with unary functions—functions with only one argument.

For example, this is the inline function definition for `atanh()`:

```
double atanh (double) : 0x0D;
```

The header file `<math.h>` defines some unary functions in this way. You may want to define others yourself.

Function prototypes with (...)

The parameter list `(...)` is allowed in a function prototype.

Dimensionless arrays

Dimensionless arrays are allowed as the last member of `struct` definitions. The member does not contribute to the size of the struct. For example:

```
struct {
   short count;
   char data[];
} CountData;        /* sizeof(CountData) is 2 */
```

Using void * with function pointers

You can use the type `void *` with function pointers, in addition to data pointers.

The THINK_C predefined symbol

THINK C predefines a preprocessor symbol, `THINK_C`, so you can test for THINK C and for the version when doing conditional compilation. In this version of THINK C, it is set to 5. In THINK C 4.0, it's set to 1. In previous versions, it's not defined.

Using the Options... Dialog

The **Options...** dialog in the **Edit** menu gives you a great deal of control over how the THINK C environment behaves and how the THINK C compiler compiles your code.

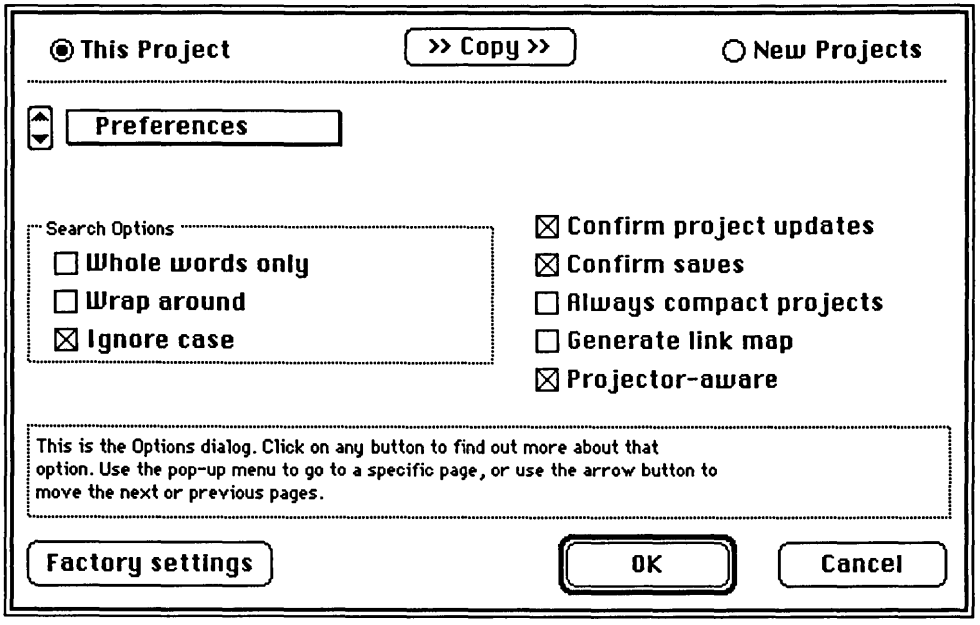


Figure 10-6 The Options... dialog

The help box is near the bottom of dialog, just above the buttons. It displays information about an option when you click on the option. To get help on an option without changing its setting, click on it without releasing the mouse button, move the mouse off the option, and then release the mouse button.

The **Options...** dialog has six pages. To go to a certain page, click on the pop-up menu in the upper left corner and select the page name. You can also scroll through the different pages with the arrow keys or the arrow buttons to the left of the pop-up menu. (In the Prefix page, you have to press the Command key with the arrow keys.)

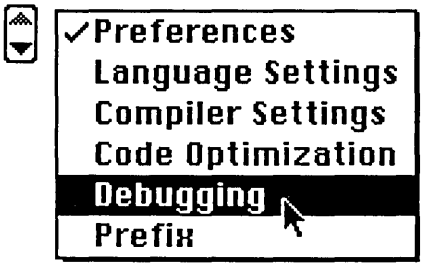


Figure 10-7 The **Options...** dialog pop-up menu

You can set the options for the current project, or you can set the defaults that THINK C will use when you create a new project. Use the Copy button at the top of the dialog box to copy the THINK C defaults to the current project, or if you want the options you've set for a particular project to be the THINK C options. Note that even though only one page of the options shows up in the dialog at one time, the Copy button copies *all* of the options, even the ones in the pages you can't see.

When you click on the Factory Settings button, THINK C sets all the options in all the pages back to their factory settings, the settings they had when you first took THINK C out of the box.

When you click on the OK button, the changes for all pages of the dialog are saved. When you click on the Cancel button, the changes for all the pages of the dialog are discarded. It's like you never selected the **Options...** command.

The following sections show you how to set the options for special needs:

To set up the options to...	Read this...
Create a link map, listing every function in your application	"Generating a link map" on page 168
Use THINK C's extensions to C	"THINK C Extensions" on page 174
Include code in every file in your project	"Using the Project Prefix" on page 181
Compile code that strictly follows the ANSI standard	"Compiling ANSI-Conformant Code" on page 182
Use the global optimizer to make your code smaller and faster	"Using the Global Optimizer" on page 184
Use other optimizations to improve your code	"Using Other Optimizations" on page 186
Strictly to type-check your code.	"Type Checking" on page 188
Port code easier by changing how THINK C interprets your declarations and Pascal string literals	"Porting Code to THINK C" on page 193
Write code for a Macintosh with a MC68020 or MC68881	"Writing Processor-Specific Code" on page 191

Some options are explained in other parts of the THINK C documentation:

To set up the options to...	Read this...
Select your debugging options	"Debugging Options" on page 243
Use the code profiler	Chapter 15, "The Profiler
Select your search options.	"Search options" on page 140
Select your object-oriented programming options.	*Object-Oriented Programming*, Chapter 4, "Using Objects in THINK C"

Using the Project Prefix

The project prefix lets you include some text in all the source files in your project. It's as if you put the project prefix at the beginning of all your source

files. You set in this prefix in the Prefix page of the **Options...** dialog, shown in Figure 10-8.

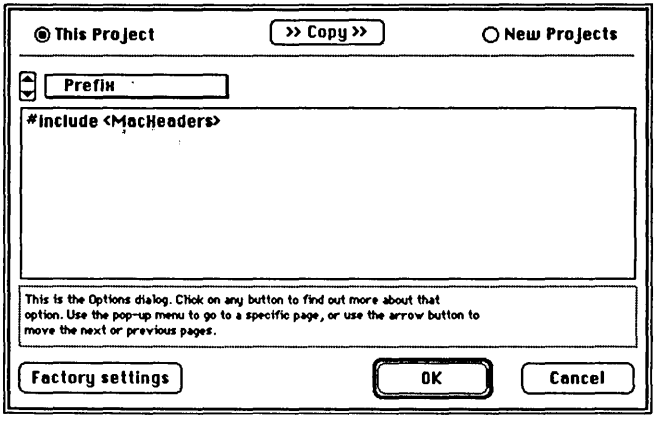


Figure 10-8 The Prefix page of the **Options...** dialog

If you use a precompiled header file, like `MacHeaders`, #include it here. By default, this page contains the line `#include <MacHeaders>`.

Note

THINK C does not precompile what's in the project prefix. To precompile something, you must use the **Precompile...** command, described in "Precompiled Headers" on page 164.

If you need to define a macro in all your files, define it here. For example, you may have some debugging code in your files that's compiled only if the macro `DEBUG` is defined. To include that code, include this line here:

```
#define DEBUG
```

When you don't need to include the debugging code anymore, just delete that line from this page. You don't need to edit every file in your project.

Compiling ANSI-Conformant Code

For more information on how THINK C complies with the ANSI standard, see Chapter 22, "Language Reference."

To compile ANSI-conformant code, you have to change some settings in the Languages Settings page of the **Options...** dialog, shown in Figure 10-9. These settings turn on ANSI features and turn off the THINK C extensions.

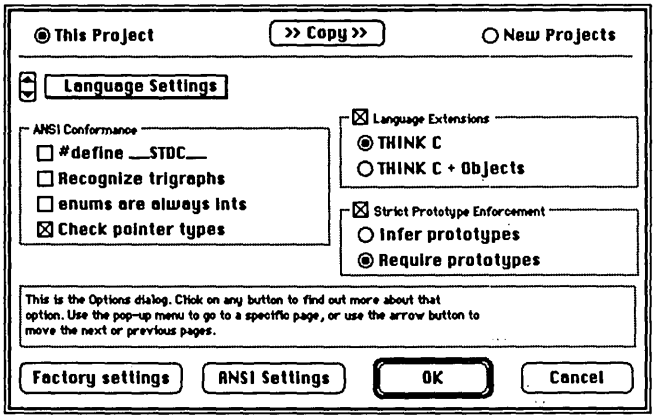


Figure 10-9 The Language Settings page of the **Options...** dialog

You can test and change the settings of these options from your code. See "Accessing Option Settings in Your Code" on page 195. and "Language extension options" on page 198

The ANSI Settings button on this page sets all the options on this page to their ANSI-conforming settings, as shown in this table:

ANSI Settings sets this...	**to this...**
#define __STDC__	On
Recognize trigraphs	On
enums are always ints	On
Check pointer types	On
Language Extensions	Off
Strict Prototype Enforcement	On & "Infer prototypes" selected

This table explains what the options do when they have their ANSI-compatible settings:

The ANSI-conformant setting of this...	**does this...**
#define __STDC__	#Defines `__STDC__`. If THINK C is ANSI-conformant, `__STDC__` is 1. If THINK C isn't ANSI-conformant, `__STDC__` is 0. If this option is off, THINK C doesn't define `__STDC__`.
Recognize trigraphs	Recognizes trigraphs. For example, `??'` is recognized as `^`. Described in "Disabling trigraphs" on page 176.

The ANSI-conformant setting of this...	does this...
enums are always ints	Makes enumerated constants the same size as an `int`. Described in "enums of any size" on page 176.
Check pointer types	Checks whether pointers match when you assign pointers or perform pointer arithmetic. Described in "Checking pointer types" on page 188.
Language Extensions	Disables all THINK C extensions, described in "THINK C Extensions" on page 174.
Strict Prototype Enforcement	Infers a prototype when you first use a function. Described in "Enforcing prototype use" on page 189.

Using the Global Optimizer

You can test and change the settings of these options from your code. See "Accessing Option Settings in Your Code" on page 195. and "Global optimizer options" on page 201

THINK C includes a global optimizer than can speed up your code. To use the global optimizer, turn on the "Use Global Optimizer" option in the Code Optimization page. The options below "Use Global Optimizer" let you choose which optimizations you want THINK C to apply. The rest of this section describes each optimization in detail.

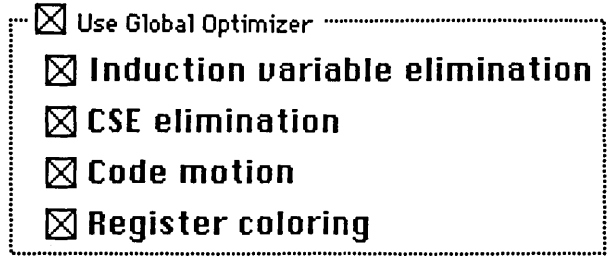


Figure 10-10 The "Use Global Optimizer" option

For more information, see "Debugging Optimized Code" on page 242.

You may not want to use the global optimizer while you're debugging. It generates machine code that is significantly different from your source code. For more information, Also, the global optimizer adds an additional pass over your compiled code and may double your compilation time.

The section "Using Other Optimizations" on page 186 describes other optimizations that don't require the global optimizer.

Induction variable elimination

This optimization makes loops faster, especially those that cycle through an array. It may make your code slightly larger. For example, take this loop:

```
int a[ARRAY_SIZE], i;

for (i=0; i<ARRAY_SIZE; i++)
   a[i] = GetNextElelment();
```

Without this optimization, your code has to perform a multiplication each time it figures the address for the next array element (`i * sizeof(int)`). With this optimization, it remembers the address of the last element and just adds the size of an element to that address.

Use this optimization if your code contains many loops. It's especially useful on a Macintosh with a MC68000, like a Macintosh Classic or Plus, which performs 32-bit multiplication in software.

CSE elimination

This optimization makes your code smaller and faster. It replaces subexpressions that are used more than once with a temporary variable set to the subexpression's value. For example, look at this code:

```
a = i*2 + 3;
b = sqrt(i*2);
```

With this optimization, your code assigns `i*2` to a temporary variable and computes it only once. It's as if the code were written like this:

```
temp = i*2;
a = temp + 3;
b = sqrt(temp);
```

Use this optimization on all your code. It's especially useful if you also use the "Automatic Register Assignment" option on page 186.

Code motion

This optimization makes your loops faster. It moves expressions out of a loop that remain constant in each iteration. For example, take this loop:

```
while ( !feof(fp) ) {
   i = x*5;
   DoSomething( fp, i );
}
```

With this optimization on, your code moves `i = x*5` outside the loop and computes it only once. It's as if the code were written like this:

```
i = x*5;
while ( !feof(fp) )
   DoSomething( fp, i );
```

Use this optimization if your code has a lot of loops.

Register coloring

This optimization makes your code smaller and faster. It searches your code for variables of the same type that are never used at the same time and has them share a register, if one is free. For example take this function:

```
void foo(void)
{
   int i, j, k;

   for (i=0; i<10; i++);
      printf( "%d\n", i );

   for (j=2; j<1024; j+=2);
      printf( "%d\n", j );

   for (k=10; k>0; k--);
      printf( "%d\n", k );
}
```

Notice that `i`, `j`, and `k` are never used at the same time. With this optimization on, THINK C has them share a register.

You can test and change the settings of these options from your code. See "Accessing Option Settings in Your Code" on page 195. and "Other optimization options" on page 202

Using Other Optimizations

This section describes optimizations that the compiler performs. You turn them on in the left column of the Code Optimization page of the **Options...** dialog.

To let THINK C optimize register allocation, see "Using register variables" on page 170

To learn about the global optimizer, see "Using the Global Optimizer" on page 184

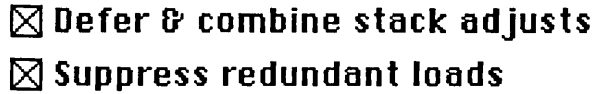


Figure 10-11 The other optimization options

Defer & combine stack adjusts

If this option is on, THINK C saves time and space in code that calls many functions in a row.

If this option is off, THINK C follows this function-calling rule: When a statement calls a C function, that statement pushes the arguments on the stack before calling the function and pops them off after it returns.

If this option is on, THINK C changes that rule. If your code contains a run of C functions, it doesn't pop a function's arguments off the stack immediately after one of the functions return. It continues to call the other functions and to push their arguments on the stack. It lets the arguments accumulate and keeps track of how many are on the stack. When the run of functions is done, it pops the arguments off all at once.

Note

THINK C can't use this optimization with Pascal functions such as Macintosh Toolbox functions or functions declared `pascal`. A Pascal function pops its arguments off the stack itself, before it returns.

If you're debugging and you use the **Skip to Here** command, you may want to turn this option off. When this option is on, THINK C must keep track of how much is currently on the stack. If you skip to a statement where that amount is different from where you are, THINK C will pop off the wrong number of arguments and may crash.

Suppress redundant loads

If this option is on, THINK C doesn't load data into a register when that data is already in a register. This optimization makes your code smaller and faster. The factory setting is on.

To understand how this optimization works, look at this example:

```
int j=0, i=1, k;

j = i + 1;
k = j;
```

When you reach the last statement (`k = j`), the value of `j` is in two places: a register and in memory. If this option is off, `k` gets the value from memory, requiring you to compute the memory address. If this option is on, `k` gets the value from the register, saving you the time and space the computation takes.

THINK C loads `j` from memory into a register twice, once for each time it appears. If this option is on, THINK C loads it from memory only once.

If you're debugging, you may want to turn this option off. When you set a variable in a data window, the debugger puts the new value into memory. However, when this option is on, your program may be using a value in a register and not in memory.

For example, look at the code above. After `j = i + 1` executes, there are two copies of `j`, one in memory and one in a register. Your program is using the register copy. The value in the register and in memory is 2. If you examine `j`, the data window shows 2.

Now, say you change the value of `j` in the data window to 10. When your program continues, `k` is set to 2, anyway. The data window changed the value of `j` in memory, but your program used the value of `j` in the register.

Type Checking

You can test and change the settings of these options from your code. See "Accessing Option Settings in Your Code" on page 195. and "Type-checking options" on page 199

THINK C lets you choose how strictly it enforces type checking. You can have THINK C make sure that pointer types are assignment-compatible, and you can enforce strict type checking by requiring a function prototype for every function. These options are in the Language Settings page of the **Options...** dialog.

If you want to compile ANSI-conformant code, you must set these options a certain way. For more information, see "Compiling ANSI-Conformant Code" on page 182.

Checking pointer types

When the "Check pointer types" option is on, THINK C makes sure that pointer types match when you assign one pointer to another or when you do pointer arithmetic. If this option is off, THINK C treats all pointers as equivalent types, and won't display the "pointer types do not match" error message. When subtracting two pointers, however, the two types must be pointers to objects of the same size.

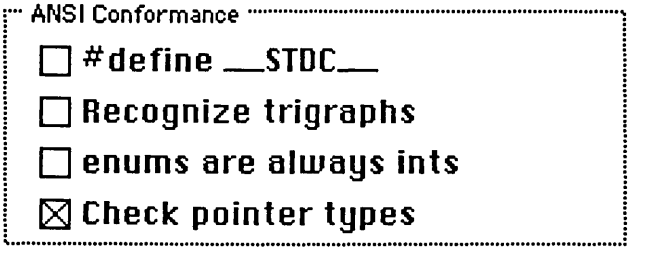


Figure 10-12 The "Check pointer types" option

When you're precompiling a header file, turning this option off can do something else. If this option is off and the "Require prototypes" option is on, the pointer types in the argument lists of the header's prototypes are changed to `void *`. For example, it's as if this prototype:

```
WindowPtr GetNewWindow( int windowID,
   Ptr wStorage, WindowPtr behind );
```

were defined like this:

```
WindowPtr GetNewWindow( int windowID,
   void *wStorage, void *behind );
```

Notice that the return type of the function isn't changed. The `MacHeaders` file is compiled like this, so you don't need to use the rarely used pointer types required in some Macintosh Toolbox calls.

In a precompiled header, you set options with the `#pragma options` directive, described in "Accessing Option Settings in Your Code" on page 195. To turn off the "Check pointer types" option and turn on the "Require prototypes" option, include this line in your precompiled header:

```
#pragma options(!check_ptrs, require_protos)
```

Enforcing prototype use

To relax prototype checking when you precompile a header file, see "Checking pointer types" on page 188.

The "Strict Prototype Enforcement" option lets you choose how strictly THINK C enforces the use of prototypes. If this option is on, you can choose between two enforcement levels: "Infer prototypes" and "Require prototypes." If this option is off, THINK C does nothing when you use or define a function without a prototype.

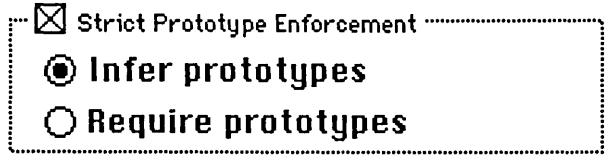


Figure 10-13 The "Strict Prototype Enforcement" option

This table explains the two enforcement levels:

If you choose...	THINK C does this when you use a function without a prototype...
Infer prototypes	Infers a prototype from the first appearance of a function. That appearance can be a function call or an old-style declaration. If a subsequent call, declaration, or prototype doesn't match the inferred prototype, it's an error.
Require prototypes	Raises an error. You can't use or define a function unless it has a prototype. New-style function do not satisfy this requirement.

Note

There is one exception to the "Require prototypes" requirement. A static function does not need a prototype. Just define it before using it.

When you use the "Require prototypes" option, THINK C does not perform argument promotion on an old-style function definition, if there is a prototype for it. Take this prototype and old-style definition for the same function:

```
int AFunction(char);

AFunction(b)
    char b;
{
    ...
}
```

If "Require prototypes" is not selected, THINK C promotes b to an `int`, and the prototype and definition don't match. If "Require prototypes" is selected, THINK C treats the old-style definition as a new-style definition, and the prototype and definition match.

These examples show you how THINK C infers prototypes. If the first appearance of `PrintFloat()` is this call:

```
PrintFloat("Pi is ", 3.14159);
```

THINK C infers this prototype:

```
int PrintFloat( char *, double );
```

And if the first appearance of `PrintInt()` is this old-style declaration:

```
PrintInt( string, value )
   char *string;
   int value;
{
   printf( "%s%d\n", string, value );
}
```

THINK C infers this prototype:

```
int PrintInt( char *, int );
```

THINK C follows the rules of argument promotion when it infers prototypes. For example, it promotes parameters of type `char` and `short` to `int`, and parameters of type `float` to `double`. For example, from this function call:

```
PrintShort("This answer is: ", (short) 32);
```

THINK C infers this prototype: (Note that THINK C promotes the `short int` to `int`.)

```
int PrintChar( char *, int );
```

You can test and change the settings of these options from your code. See "Accessing Option Settings in Your Code" on page 195. and "Generating processor-specific code options" on page 200

Writing Processor-Specific Code

THINK C lets you write code for a specific processor. You can write code for the MC68020 CPU or the MC68881 floating-point unit. To write processor-specific code, use these three options in the Compiler Settings page of the **Options...** dialog, shown in Figure 10-14: "Generate MC68020 instructions," "Generate MC68881 instructions," and "Native floating-point format."

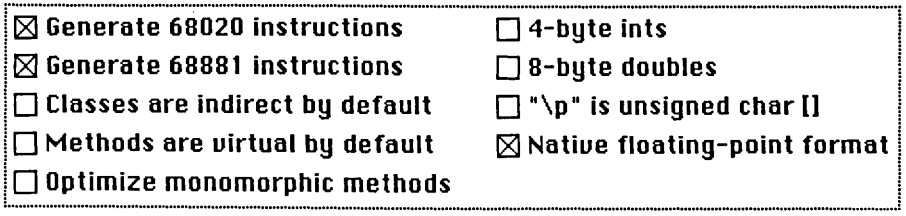


Figure 10-14 The options for writing processor-specific code

Note

Before your program uses code compiled for a specific processor, make sure that processor is present, or your program may crash. Use the Gestalt Manager, described in *Inside Macintosh VI*, Chapter 3, "Compatibility Guidelines," or `SysEnvirons()`, described in *Inside Macintosh V*, Chapter 1, "Compatibility Guidelines."

Writing code for the MC68020

To write code for a Macintosh with a MC68020, turn on the "Generate MC68020 instructions" option. THINK C generates code that is optimized for Macintoshes with the MC68020, MC68030, or MC68040 (like the Macintosh LC, II, or IIfx), and that will not run on Macintoshes with the MC68000 (like the Macintosh Classic or Plus).

If the option is on, THINK C generates MC68020 instructions for bit-field operations, addressing, and long word multiplication, division, and modulo operations. Also, the inline assembler accepts all MC68020 instructions and addressing modes for assembly in an `asm { ... }` construct.

Writing code for the MC68881

To write code for a Macintosh with a MC68881, turn on the "Generate MC68881 instructions" option. THINK C generates code that is optimized for Macintoshes with the MC68881 or MC68882 floating-point unit (like the Macintosh II or IIfx) or the MC68040, and that will not run on Macintoshes without it (like the Macintosh Classic, LC, or Plus).

For more information on using register variables, see "Using register variables" on page 170.

If this option is on, THINK C generates code for the floating point coprocessor. You may declare up to four local floating-point variables as `register` variables, and THINK C will placed them into MC68881 registers.

For more information on how THINK C represents floating-point numbers, see "Floating point representation" on page 172.

If your code contains a lot of intensive floating-point computation, you may want to turn on the "Native floating-point" option. If both the "Generate MC68881 instructions" and the "Native floating-point format" options are on, `long doubles` use the MC68881 Extended format. If you call any SANE functions, you'll have to convert from the MC68881 Extended format and the SANE Extended format. The `float` and `short double` types are identical for SANE and for IEEE.

Note

If you turn on the "Native floating-point" option, you need to recompile the `ANSI` library if your project uses it.

Porting Code to THINK C

You can test and change the settings of these options from your code. See "Accessing Option Settings in Your Code" on page 195 and "Type options" on page 197.

This section describes a few options that make porting code to THINK C easier. The first part describes how to port code from other compilers and the second part describes how to port code from previous versions of THINK C. To port code easily, you use three options in the right column of the Compiler Settings page of the **Options...** dialog, shown in Figure 10-15.

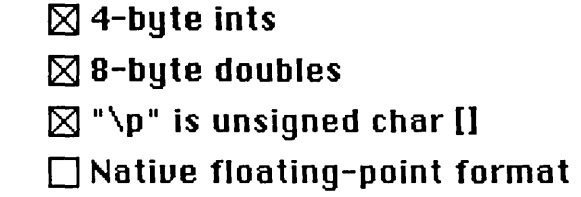


Figure 10-15 The options for porting code

Note

The ANSI-compatibility options help your port ANSI-compatible programs. Those options are described in "Compiling ANSI-Conformant Code" on page 182.

Porting code from other compilers

A common problem programmers have when porting code is that the compilers define `int` to be different sizes. By default, THINK C defines `int` to be two bytes long, smaller than a pointer. If you're porting code that assumes that `int` is four bytes long or the same size as a pointer, you many want to turn on the "4-byte ints" option in the Compiler Settings page of the **Options...** dialog. For more information on how THINK C represents integers, see "Integer representation" on page 172.

Note

No matter how this option is set, `long int` is always four bytes long and `short int` is always two bytes long.

A less common problem is that compilers define `double` to be different sizes. If you're porting code that assumes that `doubles` are eight bytes long, turn on the "8-byte doubles" option in the Compiler Settings page of the **Options...** dialog. For more information on how THINK C implements floating-point numbers, see "Floating point representation" on page 172.

Note

If the "8-byte doubles" option is off and the "Language Extensions" option is off, no type in THINK C gives you eight-byte floating-point numbers.

Porting code from other versions of THINK C

Pascal string literals are string literals that start with `\p` *or* `\P`*, like* `"\pTitle"`

Previous versions of THINK C assumed Pascal string literals to be of type `char[]`. If you have code that took advantage of this assumption, turn off the ""\p" is unsigned char []" option in the Compiler Settings page of the **Options...** dialog.

By default, THINK C now assumes Pascal string literals to be of type `unsigned char []`, which is compatible with the Macintosh Toolbox types `Str255` and `StringPtr`. When you assign a Pascal string literal to a variable of type `Str255` or `StringPtr`, you don't need to cast the string.

Using #pragma Directives

THINK C implements the following `#pragma` directives:

```
once                    parameter
options                 nooptimize
```

The `#pragma options` directive is described in "Accessing Option Settings in Your Code" on page 195. The rest are described below.

The pragma once directive

When this directive appears in a header file, THINK C includes it only once even if there is more than one `#include` directive that includes the header file.

```
#pragma once
```

The pragma parameter directive

This directive applies to a subsequent inline function definition and allows parameters to be passed in registers instead of on the stack. It specifies which register holds the return value which registers parameters are passed in. The `#pragma parameter` directive must appear before the inline declaration.

`#pragma parameter` *return-reg*$_{opt}$ *function-name* (*param-reglist*$_{opt}$)

Function-name is the name of a function that is subsequently defined inline. If the definition is not an inline definition, never defined, or already defined, the directive is ignored.

The optional *return-reg* can be `__A0`, `__A1`, `__D0`, or `__D1`.

The optional *param-reglist* is a parameter list made up of `__A0`, `__A1`, `__D0`, `__D1`, or `__D2`.

The inline definition must have a prototype and the prototype must not end with `...`. The return type must be an integer type or a pointer type. The return type must be 4 bytes long if the return is in an address register. The argument types may not exceed 4 bytes, except when the inline definition is declared `pascal`, in which case the address of the parameter is used. An address register can hold a 2-byte value for arguments only. In no case can an address register hold a 1-byte value.

Examples:

```
#pragma parameter __A0 NewHandleClear(__D0)
pascal Handle NewHandleClear(Size byteCount)
 = 0xA322;

#pragma parameter Delay(__A0,__A1)
pascal void Delay(long numTicks,long *finalTicks)
 = {0xA03B,0x2280};
```

The pragma nooptimize directive

The `nooptimize` pragma directive allows the appearance of a call to a function to disable the global optimizer for the function that makes the call.

`#pragma nooptimize (`*function-name*`)`

Function-name must have already been declared as a function, otherwise it is an error.

Any function that calls a function that has been the subject of the `nooptimize` directive will not be optimized by the global optimizer, even if the option is on. For an example, look at the implementation of `setjmp`.

This directive should only be used for functions that have atypical flow-of-control properties.

Accessing Option Settings in Your Code

You can use preprocessor directives to change and test most of the options in the **Options...** and **Set Project Type...** dialogs. The `__option` directive

lets you query option settings and the `options` pragma directive lets you change them. Both directives take option names as arguments, which are described later in this section.

Note

There are two underscores in `__option`.

To see if an option is enabled, use the `__option` directive in an `#if` directive. For example, this code fragment sets the macro `INT_SIZE` depending on how you set the "4-byte ints" option:

```
#if __option(int_4)
   #define INT_SIZE 4
#else
   #define INT_SIZE 2
#endif
```

To change the setting of an option, use the `options` pragma directive. It takes any number of option names, separated by commas. To turn an option on, include its name in the list. To turn an option off, put an `!` in front of its name. For example, this code fragment turns on the "Generate 68020 instructions" option and turns off the "Generate 68881 instructions" option:

```
#if MACINTOSH_LC
   #pragma options(mc68020, !mc68881)
#elif
   ...
#endif
```

You can place an `options` pragma anywhere in your file. If it appears outside a function, it applies for the remainder of the file. If it appears inside a function, it applies to that entire function only. The previous settings are restored when THINK C finishes compiling that function. You can change only certain options inside a function.

Internal compiler options

THINK C sets these options automatically when it compiles your project. You can't change them with the `options` pragma directive, and you can't

set them from any dialog. But you can test them with the `__option` directive.

If this is set...	Then THINK C ...
`a4_globals`	Uses A4 offsets (instead of A5) for globals. This option is set if your project is not an application.
`pcrel_strings`	Generates strings in code. This option is set if your project is a single segment code resource.
`jump_table`	Generates jump table. This option is set if your project is an application or a multi-segment desk accessory, device driver, or code resource.

Set Project Type... options

These are options in the **Set Project Type...** dialog. You can't change them with the `options` pragma directive, but you can test them with the `__option` directive.

If this is set...	Then THINK C...
`far_code`	Generates 32-bit code references. This option is like the "Far Code" option in the **Set Project Type...** dialog.
`far_data`	Generates 32-bit data references. This option is like the "Far Data" option in the **Set Project Type...** dialog.
`separate_strs`	Places strings in `STRS` section. This option is like the "Separate STRS" option in the **Set Project Type...** dialog.

Type options

These options control how THINK C interprets floating-point and integer declarations. They correspond to options in the Compiler Settings page of

the **Options...** dialog. You can't change them with the `options` pragma directive, but you can test them with the `__option` directive.

If this is set...	Then...
`int_4`	Integers are 4 bytes long, not 2. This option is like turning on the "4-byte ints" option.
`double_8`	Doubles are 8 bytes long. This option is like turning on the "8-byte doubles" option.
`native_fp`	Floating-point numbers use the native format, instead of the universal format. This option is like turning on the "Native floating-point" option.

Language extension options

These options control whether you can use some THINK C extensions to the C language. They correspond to options in the Languages Settings page of the **Options...** dialog. You can't change them with the `options` pragma directive, but you can test them with the `__option` directive.

If this is set...	Then THINK C...
`stdc`	Defines the macro `__STDC__`. This option is like turning on the "#define __STDC__" option.
`trigraphs`	Recognizes trigraph symbols. This option is like turning on the "Recognize trigraphs" option.
`thinkc`	Uses the THINK C extensions to the C language.
`objectc`	Uses the object-oriented programming extensions to the C language.

Depending how `thinkc` and `objectc` are set, you get one of the three possible settings for the "Language extension level."

If thinkc is	and objectc is	You have this setting...
Off	Off	ANSI C
On	Off	THINK C
On	On	THINK C + Objects

The **Options...** dialog doesn't let you set `thinkc` off and `objectc` on.

Enumerated type option

This option lets you choose what size enumerated types can be. It correspond to the "enums are always ints" option in the Compiler Settings page of the **Options...** dialog. You can change it outside a function, but not inside one.

If this is set...	Then...
`pack_enums`	enums can be the size of any integral type.

Pascal string option

This option lets you choose the type of Pascal string literals. It correspond to the ""\p" is unsigned char []" option in the Compiler Settings page of the **Options...** dialog. You can change it outside a function, but not inside one.

If this is set...	Then...
`signed_pstrs`	Pascal string literals (`"\p"`) are of type `char []`.

Object-oriented programming options

These options let you compile object-oriented code written for a previous version of THINK C. They correspond to options in the Compiler Settings page of the **Options...** dialog. You can change them outside a function, but not inside one.

If this is set...	Then...
`virtual`	Methods are virtual by default. This option is like turning on the "Methods are virtual by default" option.
`indirect`	Classes are indirect by default. This option is like turning on the "Classes are indirect by default" option.

Type-checking options

These options let you control how strictly THINK C type-checks your code. They correspond to options in the Languages Settings page of the **Options...** dialog. You can change them outside a function, but not inside one.

If this is set...	Then THINK C...
`check_ptrs`	Checks pointer types. This option is like turning on the "Check pointer types" option.
`require_protos`	Requires function prototypes.
`infer_protos`	Infers a prototype when a function is first used.

With `require_protos` and `infer_protos`, you can get one of the three possible settings for the "Strict Prototype Enforcement" option.

If require_ protos is	and infer_ protos is	You get this setting...
Off	Off	"Strict Prototype Enforcement" off.
Off	On	"Strict Prototype Enforcement" on, and "Infer Prototypes" selected.
On	On or Off	"Strict Prototype Enforcement" on, and "Require prototypes" selected.

Generating processor-specific code options

These options let you generate code for a specific processor. They correspond to options in the Compiler Settings page of the **Options...** dialog. You can change them anywhere.

If this is set...	Then THINK C...
`mc68020`	Generates code for the MC68020. This option is like turning on the "Generate 68020 code" option.
`mc68881`	Generates code for the MC68881. This option is like turning on the "Generate 68881 code" option.

Note

If `native_fp` is on, you can't change the setting of the `mc68881` option.

Debugging options

These options help you debug your code. They correspond to options in the Debugging page of the **Options...** dialog. You can change them anywhere.

If this is set...	Then THINK C...
`profile`	Generates calls to the profiler. This option is like turning on the "Generate profiler calls" option.
`force_frame`	Generates a stack frame for each function call. This is option like

	turning on the "Always generate stack frames" option.
`macsbug_names`	Generates Macsbug names.
`long_macsbug_names`	Uses long format for macsbug names.

With `macsbug_names` and `long_macsbug_names`, you can get one of the three possible settings for the "Macsbugs Names" option.

If macsbug_ names is	**and long_macs- bug_names is**	**Your get this setting...**
Off	On or Off	"Macsbugs Names" off.
On	Off	"Macsbugs Names" off, and "Short format" selected.
On	On	"Macsbugs Names" off, and "Long format" selected.

Global optimizer options

These options control the global optimizer, described in "Using the Global Optimizer" on page 184. They correspond to options in the Debugging page of the **Options...** dialog. You can change them anywhere.

If this is set...	**Then...**
`global_optimizer`	Use the global optimizer on this code. This option is like turning on the "Use Global Optimizer" option.
`gopt_induction`	If the global optimizer is on, simplify induction expressions. This option is like turning on the "Induction variable elimination" option.
`gopt_cse`	If the global optimizer is on, eliminate common expressions. This option is like turning on the "CSE elimination" option.
`gopt_loop`	If the global optimizer is on, move loop invariants. This option is like turning on the "Code motion" option.
`gopt_coloring`	If the global optimizer is on, optimize register usage. This option is like turning on the "Register coloring" option.

Note

Inside a function, you can turn off `global_optimizer`, but you cannot turn it on.

Other optimization options

These options control other optimizations. They correspond to options in the Code Optimization page of the **Options...** dialog. You can change them anywhere.

If this is set...	Then THINK C...
`defer_adjust`	Defers and combine stack adjusts. This option is like turning on the "Defer & combine stack adjusts" option.
`redundant_loads`	Suppresses redundant loads. This option is like turning on the "Suppress redundant loads" option.
`honor_register`	Honors register declarations.
`assign_registers`	Lets the compiler assign variables to registers.

With `honor_register` and `assign_registers`, you can get one of the three possible settings for the "Automatic Register Assignment" option, plus an extra one that wasn't included.

If honor_ register is	If assign_ registers is	You get this setting...
On	Off	"Automatic Register Assignment" off.
On	On	"Automatic Register Assignment" on, and "Honor 'register' first" on.
Off	On	"Automatic Register Assignment" on, and "Honor 'register' first" off.
Off	Off	Put nothing in registers.

Working with the Toolbox

11

The Macintosh Toolbox is the set of over 1,000 routines that give Macintosh applications the consistent interface. This chapter tells you what you have to do to call the routines described in Inside Macintosh. It also describes the few differences between the Apple interfaces to the Toolbox and the THINK C interfaces to the Toolbox.

Contents

Calling Toolbox Routines

With THINK C, you can use all of the routines described in *Inside Macintosh I-VI* including the ones marked `[Not in ROM]`. To use the Toolbox routines, call them exactly as they appear in *Inside Macintosh*. The only thing you need to know is how to convert the Pascal declarations into C declarations.

Most Macintosh routines are implemented as traps that use Pascal calling conventions. THINK C generates these traps inline instead of generating a subroutine call. For routines marked `[Not In ROM]`, THINK C generates calls to library functions in `MacTraps`. Calls to these routines also follow Pascal calling conventions.

To use a Macintosh Toolbox routine, you must be using the right library and the right headers. If you're using `MacHeaders`, you won't have to include a header file for the most common Toolbox routines.

This library...	Contains...
`MacTraps`	Glue for routines in *Inside Macintosh I-V*, definitions for QuickDraw globals, and the Gestalt glue.
`MacTraps2`	Glue for routines in *Inside Macintosh VI* and less common routines.
`AppleTalk`	The alternate AppleTalk library.
`nAppleTalk`	The preferred AppleTalk library.
`HyperXLib`	Glue for HyperCard XCMDs.
`CommToolbox`	Library for the Communications Toolbox.
`PrGlue`	Printing glue for earlier System Software.
`SANE`	SANE routines

For documentation on HyperCard XCMDs *and the Communications Toolbox, contact APDA. See "Apple Programmer's and Developer's Association (APDA)" on page 14.*

Almost every Macintosh program needs to use `MacTraps`. If you use routines described in *Inside Macintosh VI*, you should also use `MacTraps2`.

Passing arguments to Toolbox routines

You don't need to know the details of C and Pascal calling conventions, though you might find it useful. See Chapter 13, "Assembly Language."

Since the argument declarations in *Inside Macintosh* are given in Pascal, you need to know how to convert them to C. This table gives you the general rule for converting argument declarations from Pascal to C:

If the object is...	Pass...
a `var` parameter	a pointer to the object
4 bytes or smaller	the object
larger than 4 bytes	a pointer to the object

Here are some examples of Pascal declarations and their C counterparts:

Pascal Type	C Type
`integer`	`short`
`longint`	`long`
`char`	`short`
`Boolean`	`Boolean or char`
`Byte`	`Byte` in `struct` declarations, `short` when passed as an argument.
`var Byte`	`short *`
`ProcPtr`	See "Working with Pascal Routines" on page 212.
`Handle`	`Handle`
`var Handle`	`Handle *`
`Ptr`	`Ptr`
`var Ptr`	`Ptr *`
`OSType, ResType`	`OSType, ResType, long`
`packed array [1..4] of char`	`long`
`Str255`	`Str255` or `unsigned char *`
`var Str255`	`Str255` or `unsigned char *`
`StringPtr`	`StringPtr` or `unsigned char *`
`var StringPtr`	`StringPtr *` or `unsigned char **`
`Rect`	`Rect *`
`var Rect`	`Rect *`
`Point`	`Point`
`var Point`	`Point *`
`Extended`	`extended`, or in certain cases, `double` See "Working with Floating-point" on page 214.

Working with Pascal strings

Toolbox routines that take strings as arguments expect them to be Pascal strings. Unlike null-terminated C strings, Pascal strings begin with a length byte. To write a Pascal string constant, start the string with `"\P"` or `"\p"`. This is how you would call the QuickDraw routine `DrawString()`:

```
DrawString("\pThis is a Pascal string");
```

Because Pascal strings start with a length byte, the longest a Pascal string can be is 255 bytes. Pascal strings are *not* null terminated. Pascal string literals are of type `unsigned char []` by default to agree with the definition of `Str255`. Older versions of THINK C treated Pascal string literals as

`char []`. You can change this behavior in the "Compiler Settings" page of the **Options...** dialog.

You can use these routines to convert strings from one form to the other:

To convert...	use this function...
A C string to a Pascal string	`CtoPstr()`
A Pascal string to a C string	`PtoCstr()`

These routines convert the strings in place, and return the converted string. These are their function prototypes:

```
unsigned char *CtoPstr(char *s);
char *PtoCstr(unsigned char *s);
```

Note

If you're not using `MacHeaders`, include `pascal.h` to use these functions.

Calling AppleTalk routines

If your application uses AppleTalk, you should be aware that there are two sets of AppleTalk interfaces. Apple calls the old interface (described in *Inside Macintosh Volume II*) the **alternate** set. The new interface (described in *Inside Macintosh Volume V)* is called the **preferred** set. You can use either or both sets of interfaces.

To use the alternate AppleTalk routines, use the library `AppleTalk`. To use the preferred routines, use `nAppleTalk` and `AppleTalk`. In either case, you should include the header file `AppleTalk.h`.

To use this interface...	Use these libraries
preferred	`nAppleTalk and AppleTalk`
alternate	`AppleTalk`

Earlier versions of THINK C used two header files, `AppleTalk.h` and `nAppleTalk.h`. This version uses only one header file, `AppleTalk.h`, supplied by Apple. The data structure definitions in the earlier version were adapted from the Pascal definitions in *Inside Macintosh*. The data structure definitions in the current header files are slightly different, so you may have to make changes to your source code.

Calling Print Manager routines

There are two ways to use the Print Manager. If you want your project to run under System 4.1 or earlier, include `Printing.h` and add the library `PrGlue`. 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 `PrintTraps.h` so all calls to the Print Manager routines use the traps directly.

Warning

You can use either `Printing.h` or `PrintTraps.h`. You cannot include both header files in the same source file, or you will get compiler errors.

`MacHeaders` does not include either `Printing.h` or `PrintTraps.h`.

To run on System...	Use this library / header
4.1 or earlier	`PrGlue` / `Printing.h`
later than 4.1	*no library* / `PrintTraps.h`

The Macintosh Header Files

THINK C gets its information about the Toolbox routines from the header files in the `Mac #includes` folder. The header files contain C prototypes for all the Toolbox routines. In many cases, the Toolbox routines are defined as inline functions.

Note

Earlier versions of THINK C had a built-in list of all the Toolbox routines and information about the arguments that they took. THINK C 5.0 does not have a built-in list. Instead, it uses the information in the header files.

Beginning with this version, THINK C uses virtually the same header files that Apple provides with its MPW C compiler. In some cases, where the compilers differ, THINK C has its own header files or uses modified versions of the header files.

For the most common Toolbox routines, you won't have to include any header files explicitly if you use the `MacHeaders` precompiled header.

The Mac #includes folder

The `Mac #includes` folder contains the following folders and files:

Item	Description
`Apple #includes`	This folder contains all of the Macintosh header files from Apple. Some of them have been modified for THINK C. The next section describes the modifications.
`Mac #includes.c`	This file is the "source file" for MacHeaders, the default precompiled header. Instructions in this file tell you how you can modify it to create a customized precompiled header.
`MacHeaders`	This is the default precompiled header. It contains the symbols from the most commonly used Toolbox manager.
`THINK #includes`	This folder contains header files unique to THINK C. The files are described in more detail below.

The Apple #includes folder

This folder contains the header files for the Macintosh Toolbox. Most of these files come from Apple, and they are the official interfaces to the Macintosh Toolbox. The following files have been modified for THINK C. All of the changes are marked in the file with conditional compilation directives, except for `SANE.h` which is completely replaced with a THINK C version.

File	Modification
`Connections.h`	Change in `enum` definitions to accommodate different size of `int` in THINK C.
`Files.h`	Change type of bit-field to take into account differences in bit-packing.
`Packages.h`	Add `#include <BDC.h>` directive. See description of `BDC.h` below.
`Quickdraw.h`	Change in `enum` definitions to accommodate different size of `int` in THINK C.
`SANE.h`	Replace with THINK C version. See description of `SANE.h` below.
`Sound.h`	Don't define type `Time` to avoid conflict with low memory global named Time.
`Types.h`	Add special definition of `extended` that is always 80 bits wide and THINK C's definition of `NULL, ((void *) 0)`.

If you want to examine the changes, open the header file and search for the symbol `THINK_C`.

The THINK #includes folder

This folder contains header files unique to THINK C. They contain definitions for the inline assembler, routines for working with globals in non-application projects, definitions of low memory globals, and so on.

Most of these files are included in `MacHeaders`:

File	Included in MacHeaders?
`asm.h`	yes
`BDC.h`	yes
`LoMem.h`	yes
`pascal.h`	yes
`SANE.h`	no
`SetUpA4.h`	no
`THINK.h`	yes

asm.h

This header file contains definitions for the inline assembler. It contains definitions for the trap modifier bits and inline function definitions that are used differently in assembly language than they are in C. To learn about the inline assembler, see Chapter 13, "Assembly Language."

BDC.h

This header file contains the function prototypes for the routines in the Binary-Decimal Conversion Package, `StringToNum()` and `NumToString()`. These routines are declared in the Apple header file, `Packages.h`.

Since programmers use these functions often, and since `Packages.h` brings in `Script.h`, which is rather large, the THINK C version breaks out these functions into a smaller header file so it can be included in `MacHeaders`.

LoMem.h

This file contains definitions of the low memory globals that take advantage of THINK C's absolute address extension. The corresponding Apple header file is `SysEqu.h`. `LoMem.h` does not define every single low memory global, just the most common ones. If you want to use a low memory global that's not defined in `LoMem.h`, you can use THINK C's absolute address extension to define it in your source file.

Warning

You can use either LoMem.h or SysEqu.h. You cannot include both header files in the same source file, or you will get compiler errors.

For example, this is the way the low memory global WindowList is defined in LoMem.h and in SysEqu.h:

```
WindowPeek  WindowList : 0x9D6;  // LoMem.h

enum {
   WindowList = 0x9D6            // SysEqu.h
};
```

If you use LoMem.h, you can treat low memory globals the same way you would any other global variable:

```
#include <LoMem.h>    // or MacHeaders
void F(void)
{
   WindowPeek  w;

   w = WindowList;    // Get first window
   ...
}
```

If you use SysEqu.h, you have to cast the value to use it:

```
#include <SysEqu.h>  // but not MacHeaders
void G(void)
{
   WindowPeek  w;

   w = * (WindowPeek *) WindowList;
}
```

If you're programming exclusively in THINK C, you should use LoMem.h. If you want to be able to compile your program with MPW C, use SysEqu.h.

Note that MacHeaders includes LoMem.h by default. If you want to use MacHeaders, and you want to use SysEqu.h, you should build a new version of MacHeaders that includes SysEqu.h instead of LoMem.h. See "Precompiled Headers" on page 164 and the comments in Mac #includes.c for instructions.

pascal.h

This header file contains the declarations for the functions `CtoPstr()` and `PtoCstr()` that translate C strings into Pascal strings and vice versa. See "Working with Pascal strings" on page 206 for a full description.

SANE.h

This is the header file for the THINK C implementation of SANE, the Standard Apple Numerics Environment. This header file contains the definitions of SANE types and the declarations of SANE functions. See "Working with SANE" on page 214 for more information.

If you are using MPW C, be sure to use the `SANE.h` that Apple provides instead of the `SANE.h` provided with THINK C. The two versions of `SANE.h` are not compatible.

SetUpA4.h

This header file contains the utility routines that let you use global variables in desk accessories, device drivers, and code resources. See "Global data in drivers" on page 108 and "Global data in code resources" on page 118 for more information.

THINK.h

This header file contains definitions and declarations that have been part of previous versions of the THINK C environment but that have no counterparts in the Apple header files. For instance, the upper case versions of the Boolean values, `TRUE` and `FALSE`, are defined here, as are the macros `topLeft` and `botRight` for extracting points from rectangles.

Working with Pascal Routines

Because the Macintosh Toolbox routines expect to be called with Pascal calling conventions, you will probably need to write functions in C that behave as though they were written in Pascal.

Pascal callback routines

Some Macintosh Toolbox routines take a pointer to another function as an argument. Those routines then call the function you passed in. The function you provide is called a **callback** routine. The Toolbox routines expect the callback routines to follow Pascal calling conventions. (To learn about the difference between Pascal calling conventions and C calling conventions, see Chapter 13, "Assembly Language.")

THINK C gives you a way to write functions that behave as though they were written in Pascal. The function definition must begin with the `pascal`

keyword. Make sure you provide a return type. If your function needs to behave like a Pascal procedure, the return type is `void`.

For the parameter declarations, follow the same rules as for calling Toolbox functions. Remember that non-`var` parameters are passed by value, not by reference. If the size of a non-`var` parameter is greater than 4 bytes, pass its address, but do not modify the parameter in your function.

For example, `ModalDialog()` lets you provide a `filterProc` to handle events in your dialog. This is how *Inside Macintosh* declares `ModalDialog()`:

```
PROCEDURE ModalDialog (filterProc: ProcPtr;
   VAR itemHit: INTEGER);
```

`ModalDialog()` expects the `filterProc` to have this declaration:

```
FUNCTION MyFilter (theDialog: DialogPtr;
   VAR theEvent: EventRecord;
   VAR itemHit: INTEGER) : BOOLEAN;
```

In THINK C, the declaration for `MyFilter()` would look like this:

```
pascal Boolean MyFilter (DialogPtr theDialog,
   EventRecord *theEvent, short *itemHit)
```

The call to `ModalDialog()`, then, looks like this:

```
void F(void)
{
   extern pascal Boolean MyFilter();
   short theItem;

   ...
   ModalDialog(MyFilter, &theItem);
   ...
}
```

Keeping C and Pascal on speaking terms can be tricky, but THINK C tries to make it as painless as possible.

Calling Pascal routines indirectly

Only Macintosh Toolbox routines and functions that are declared `pascal` are called using Pascal conventions. If you have a pointer to a Pascal routine, you can call it indirectly through a pointer the same way you would a routine in C, but the pointer must be of type pointer-to-pascal-function.

For instance, this type definition defines PFunc as a pointer to a Pascal function that takes one short argument and returns short:

```
typedef pascal short (*PFunc)(short);
```

This function takes a Pascal routine as an argument and calls it indirectly:

```
short DoPascal(PFunc theFn, short i)
{

// theFn is a pointer to a pascal function that
// returns short

   i = (*theFn)(i);  // call it indirectly

   i = theFn(i);     // this works too, but you
                     // might not be able to
                     // tell that it was called
                     // indirectly
   return (i);
}
```

Now suppose you have a function declared pascal like this:

```
pascal short Add1(short v)
{
   return v + 1;
}
```

You could call the DoPascal function defined above with it like this:

```
j = DoPascal(Add1, 10);
```

Working with Floating-point

For more information about floating-point types in THINK C see Chapter 10, "The Compiler," and Chapter 22, "Language Reference."

THINK C lets you use different floating-point formats, so you can take advantage of any special floating-point hardware or write libraries that will work regardless of the client project's settings. The main issues surrounding floating point and the Macintosh Toolbox involve SANE, the Standard Apple Numerics Environment and using the extended type.

Working with SANE

To use SANE, add the SANE library to your project, and include the file SANE.h in your source file. This library implements the SANE functions as well as some utility functions like x80tox96(), x96tox80(), str2dec(), cstr2dec(), dec2str(), and more. For more information these routines and on SANE, read *Apple Numerics Manual, Second Edition* (Addison-Wesley).

These eight functions are defined in both the SANE and ANSI libraries:

```
atan()      cos()       exp()
fabs()      log()       sin()
sqrt()      tan()
```

To use the SANE versions, include the file `SANE.h` before `math.h`. Similarly, to use the ANSI versions, include the file `math.h` before the file `SANE.h`.

The SANE library uses 80-bit `extended` values, so if your Compiler Settings are other than SANE-native ("Generate 68881 instructions" off and "Native floating-point" on), be sure to use the `x80tox96()` and `x96tox80()` functions to convert between formats as in the example below.

The extended type

Some Toolbox routines, and most SANE routines, use the type `extended` for floating-point numbers. In THINK C, the type `extended` is always an 80-bit SANE extended floating point number, regardless of the "Generate 68881 instructions" and "Native floating-point" settings. When you're using SANE-Native mode ("Generate 68881 instructions" off and "Native floating-point" on), the type `extended` is equivalent to `long double`, and you can use arithmetic and assignment operators on `extended` values.

When "Generate 68881 instructions" is on, or when "Native floating-point" is off, the type `extended` is a `struct`. You can still use it to hold `extended` values, but you'll need to convert them to 96-bit values before you can use the arithmetic or assignment operators with them.

For example, the Toolbox routine `Frac2X` takes a `Fract` value and converts it to an `extended` value. The following THINK C function converts a `Fract` value to a `long double` value regardless of the compiler settings:

```
long double Frac2Dbl (Fract f)
{
   long double result;

#if !__option(mc68881) && __option(native_fp)
   result = Frac2X(f);
#else
   {
      extended temp;

      temp = Frac2X(f);
      x80tox96(&temp, &result);
   }
#endif
   return (result);
}
```

For a description of the `__option` directive see "Accessing Option Settings in Your Code" on page 195.

To use the `x80tox96()` and `x96tox80()` functions, you need to add the SANE library to your project. These are the function prototypes for `x80tox96()` and for `x96tox80()`:

```
void x80tox96(extended *x80, long double *x96);
void x96tox80(long double *x96, extended *x80);
```

The 96-bit value that `x80tox96()` stores in its second argument is in the THINK C universal floating-point format, so it's compatible with the MC68881.

A note on SANE implementations

The SANE library provided with THINK C is not identical to the SANE library provided by Apple with MPW C, though it has the same functionality.

MPW uses two SANE libraries—on for MC68881 code generation and one for straight MC680x0 code generation—because the size of an MPW `extended` changes if the compiler is generating code for the MC68881. The THINK C extended is always the same size, 80 bits, regardless of code generation options.

If you're curious, the sources for the THINK C SANE library are included in your THINK C package.

The Debugger 12

This chapter describes THINK C's source level debugger. The source level debugger lets you debug your application the way you wrote it: in C. The debugger lets you step through your program line by line, set breakpoints, examine and set the values of your variables.

This chapter describes some of the more advanced features of the THINK C debugger. Chapter 5, "Tutorial: Bullseye," is a tutorial that teaches you how to use the debugger. You might want to work through that tutorial before you read this chapter.

Before you begin

Make sure the THINK C Debugger is in the same folder as THINK C. To use the source debugger you need at least 2 megabytes of memory, and you must be running THINK C under System 7.0 or MultiFinder. To debug large projects, you need about 4 megabytes. The debugger works only with application projects. It won't work with code resources or device drivers.

Contents

Running with the Debugger

The debugger runs as a subordinate application with THINK C. Although there is a debugger document icon, you can't launch it from the Finder by itself. When you set the option to use the source debugger, THINK C takes care of launching it. Make sure that the THINK C Debugger file is in the same folder as THINK C.

Turning the debugger on

*Another way to turn the debugger on is to check the Use Debugger check box in the Debugging section of the **Options...** dialog.*

To run your application with the debugger, choose the **Use Debugger** command in the **Project** menu. This command turns the "Use Debugger" option on and off. When the option is on, THINK C adds a "bug" column to the project window.

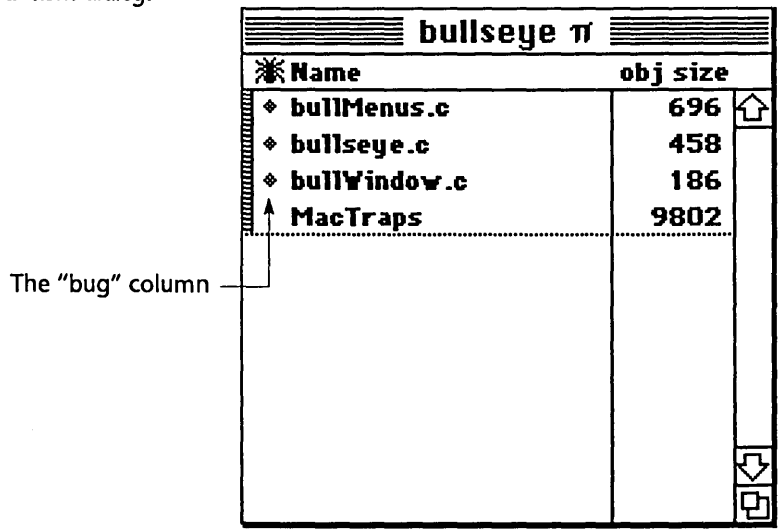


Figure 12-1 A project window with a "bug" column

THINK C generates debugging information for files that have gray diamonds next to them. Initially, all files get gray diamonds. THINK C never generates debugging information for libraries or projects used as libraries.

Clicking in the "bug" column next to a file name turns the gray diamonds on and off. If you hold down the Option key as you click on a gray diamond, THINK C removes the gray diamonds from every file. If you hold down the Option key as you click in the "bug" column where there isn't a gray diamond, THINK C turns the diamonds on for every file in the project.

Running the project

When you run your program, THINK C launches the source debugger instead of launching your program. The debugger then gets ready to run your program.

The debugger displays its own menu bar and two windows at the bottom of your screen. If you're using a Macintosh with more than one screen, and you have the "Use second screen" option on, the two debugger windows appear in the second screen.

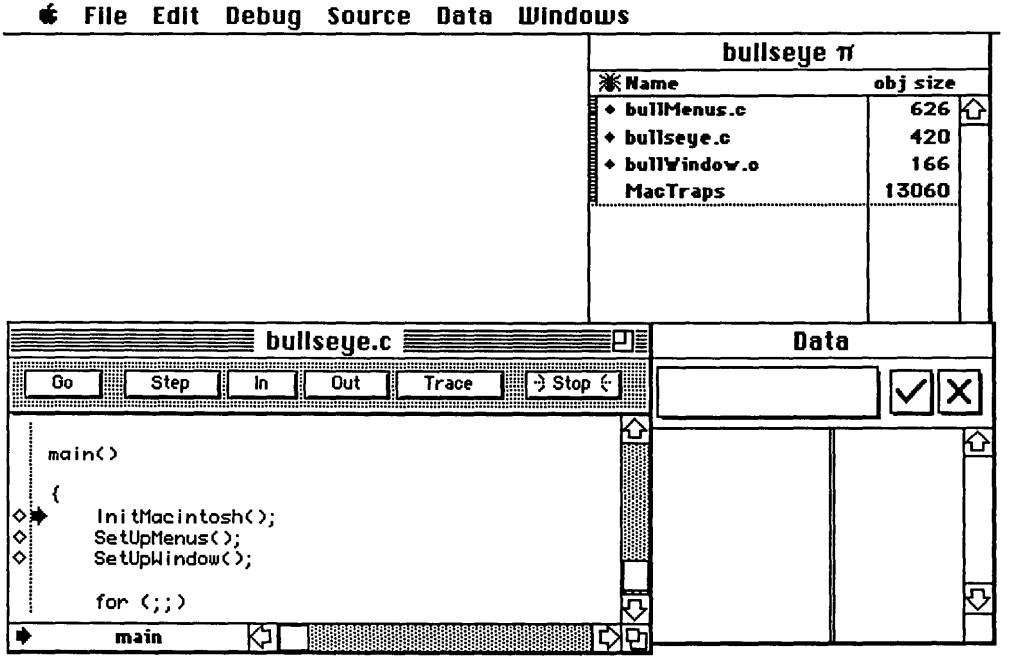


Figure 12-2 Running a program with the THINK C Debugger

The larger window on the left is the **Source window**. It contains the debugger status panel and the source text of your program. The window on the right is the **Data window**. Use the Data window to display and change the values of your variables.

When your application is running, the screen shows the application's menu bar, and its windows are brought forward. When your application stops (at a breakpoint, for example), the debugger's menu bar appears, and its windows are brought forward.

The Debugger Windows

The debugger windows show you the source of your program and the values of your variables. Since the debugger works like any other application running under MultiFinder, you need to be aware whether it is your application or the source debugger in the foreground. Just because the debugger windows are frontmost doesn't mean that your program isn't running. In fact, if your application can run in the background under MultiFinder, it will continue doing its background processing even if the debugger is the active application.

The Source window

The Source window contains the source text of your program, the debugger's status panel, statement markers, the current statement arrow, and the current function indicator. The title of the source window is the name of the source file.

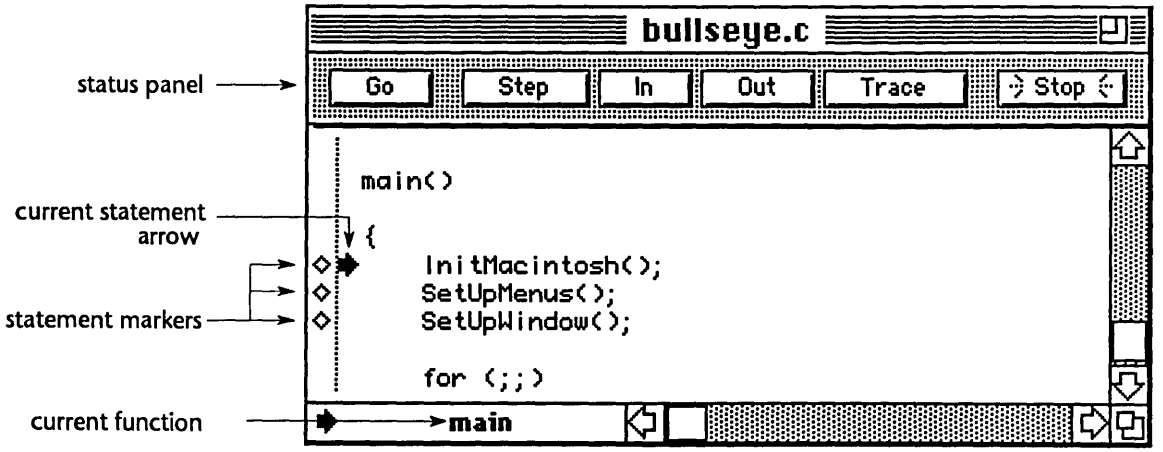


Figure 12-3 The source window of the THINK C debugger

See "Working with the Source Window" on page 223.

The Source window shows the **source text** of your program. When you start the debugger, this window shows the file that contains the `main()` routine of your application

See "Controlling Execution" on page 231.

The top of the Source window has a six button **status panel**. These buttons control the execution of your program. The names of these buttons match the commands in the debugger's **Debug** menu

See "Setting Breakpoints" on page 227.

The column of diamonds running along the left side of the source text are **statement markers**. Every line of your program that generates code gets a statement marker. You can set breakpoints only at statement markers.

See "The current statement and the selected statement" on page 223.

The black arrow to the right of the statement markers is the **current statement arrow**. This indicator shows you the **current statement**, the one that the debugger is about to execute. When you first start your program, the current statement arrow is at the first executable line of your program.

See "The current function indicator" on page 224.

The source debugger uses the space at the lower left of the source window for the name of the **current function**. When you click here and hold the mouse button down, the debugger displays a pop-up menu that shows the call chain—the names of the functions that were called to get to the current function.

The Data window

The other debugger window is the Data window. In this window you can examine and set the values of your variables.

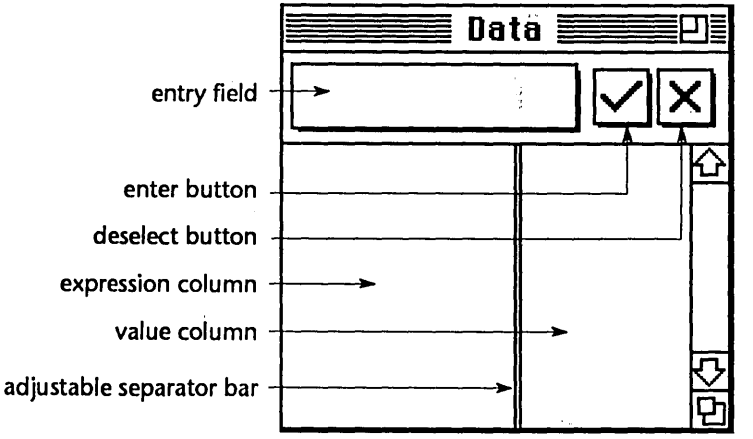


Figure 12-4 The data window of the THINK C debugger

The Data window is modeled after a spreadsheet. Expressions you type into the **entry field** appear in the left column when you press the **enter button** (check mark) or when you press the Return or Enter key. Pressing the Enter key leaves the expression selected. Pressing the Return key leaves the entry field empty so you can type in the next expression. The enter button works just like the Enter key.

If you change your mind and don't want to enter an expression, press the **deselect button** (X mark). The expressions you enter appear in the left column and their values are displayed in the right column.

You can select items in either the expression column or in the value column. If it's legal to edit the item, you can use the entry field to edit the item.

You can drag the center bar to make a column wider.

To clear an expression in the Data window, select it and choose **Clear** from the **Edit** menu or press the Clear key. To clear all the expressions, choose **Clear All Expressions**.

Working with the Source Window

The debugger gets the text for the source window directly from your source file, so you see the file exactly the way you wrote it. As you step through your code and move from file to file, the debugger gets the text for the files it needs. The Source window only displays source text. You can select text to copy, but you can't change it.

The debugger displays the source text only if THINK C generated debugging information for it. If you step into a function that's in a file that didn't have a gray diamond next to it in the project window, the debugger displays the message "Debugging information not available."

See "Editing files while debugging" on page 226.

If a source file is missing, or if you've edited a file while you're debugging, the debugger displays the message "Source text not available."

The current statement and the selected statement

The **current statement** is the one that the debugger is about to execute. The current statement arrow always points to the current statement.

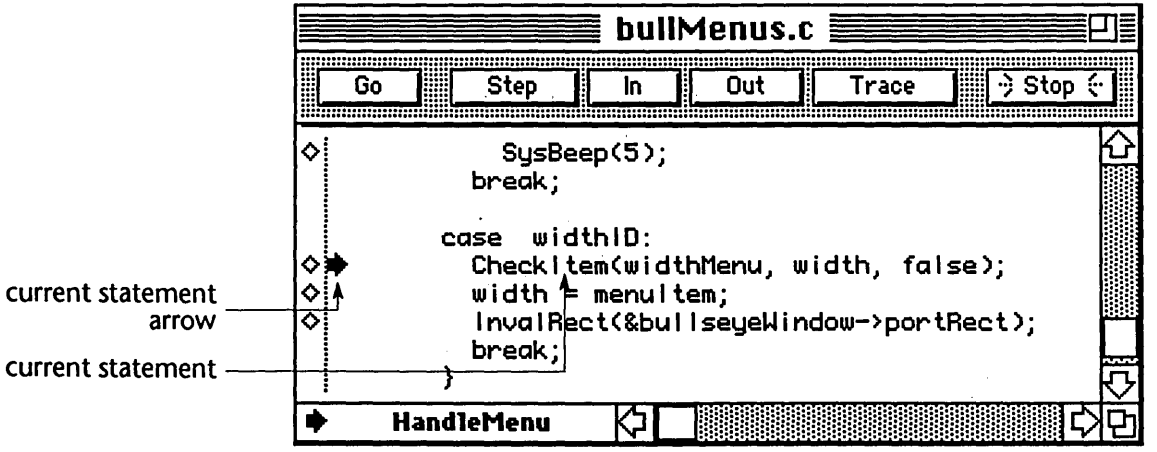


Figure 12-5 The current statement

Some of the debugger commands require you to select a statement in the Source window. To select a statement, just click once anywhere on its line. This line is called the **selected statement**.

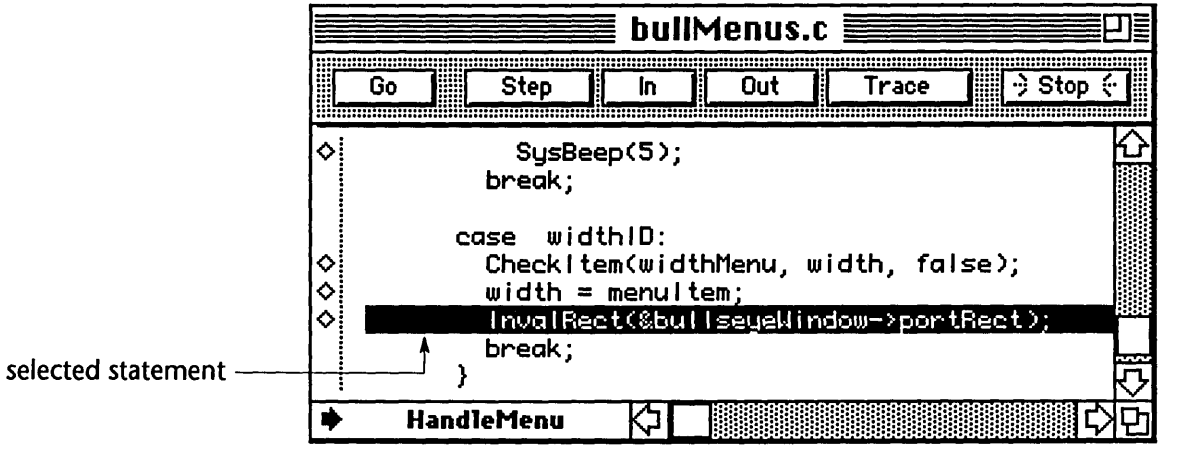


Figure 12-6 The selected statement

All commands that work on the selected statement work on the current statement if there isn't a selected statement.

If the current statement arrow isn't visible (because you've scrolled away or because you're viewing another source file in the Source window), click on the current function indicator in the lower left of the source window, or press the Enter key. The source file that contains the current statement will appear in the Source window.

The current function indicator

The current function indicator at the lower left of the Source window displays the name of the function that the current statement is in. If the current statement is in a library, the current function indicator displays the name of the file. If the current statement is in ROM, the current file indicator displays the address of the program counter.

When you click and hold the mouse button down on the current statement indicator, you'll see a pop-up menu that shows you the **call chain**. The call chain follows stack frames through register `A6`, so if a function doesn't generate a stack frame, you won't see it in the call chain. You can turn on the

"Always generate stack frames" option to have THINK C generate a stack frame for almost every function. See page 244 for more information.

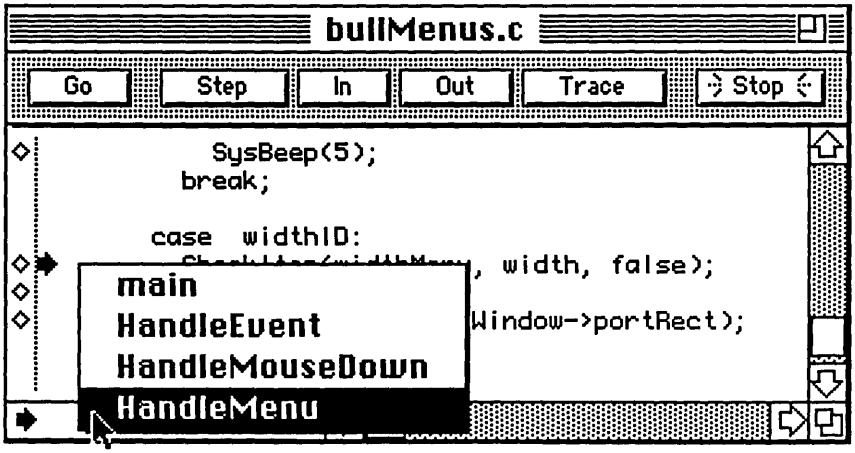


Figure 12-7 The call chain pop-up menu

If you choose an item in the call chain pop-up menu, the debugger opens the file that the selected function is in, and selects the line that the called function would return to.

Viewing other files in the source window

The Source window usually shows the file that contains the current statement. To look at another file in the Source window (to set breakpoints in it, for example) you tell THINK C to send the text to the debugger:

1. Click on the THINK C project window
2. Click on the name of the file you want to look at
3. Choose **Debug** from THINK C's **Source** menu.
4. The file that you chose appears in the Source window.

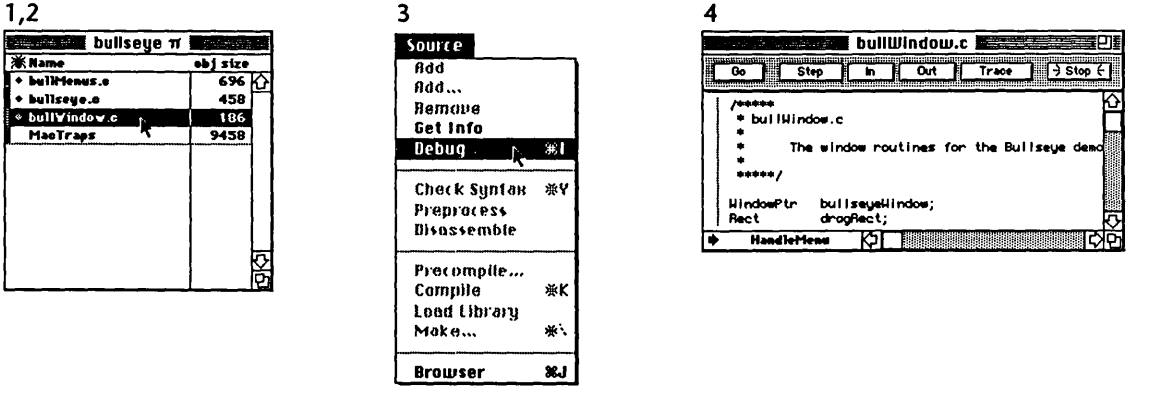


Figure 12-8 Viewing another file in the source window

Now you can examine the file and set breakpoints in it. To get back to the current file, click on the current function indicator at the lower left of the Source window or press the Enter key.

Note

If you select text in the source file before you choose **Debug** from the **Source** menu, the THINK C debugger scrolls the file so the selected text is visible in the Source window.

Editing files while debugging

Because THINK C is still running, you can edit your source files while you're debugging. To edit the file that's in the source window, choose the **Edit *'filename.c'*** command in the **Source** menu.

To edit any other file in your project, click on the project window (or choose your project name from the **Windows** menu), and edit your files normally.

Naturally, the changes you make to source files won't take effect until you recompile.

Whenever the debugger needs the source text for a file, it looks for it on disk. The debugger caches the contents of the file as long as it can, but if it needs the memory for something else, it may reclaim the cache space. If you edit and save a file that the debugger has displayed in the Source window, it will continue to display the unedited version of the file as long as it's still cached. If the debugger sees that the source file on disk and the object code in the project don't match, it displays the message "Source text not available." in the Source window.

Searching in the Source window

To search for something in the source window, you use the THINK C editor.

1. Choose **Edit *'filename.c'*** from the **Source** menu (or press Command-E) to open an edit window for the file in the Source window.
2. Use the THINK C **Find...** command to find what you're looking for.
3. Choose **Debug** from the THINK C **Source** menu (or press Command-I) to get back the source debugger.
4. The line containing the selection in the editor window is displayed in Source window.

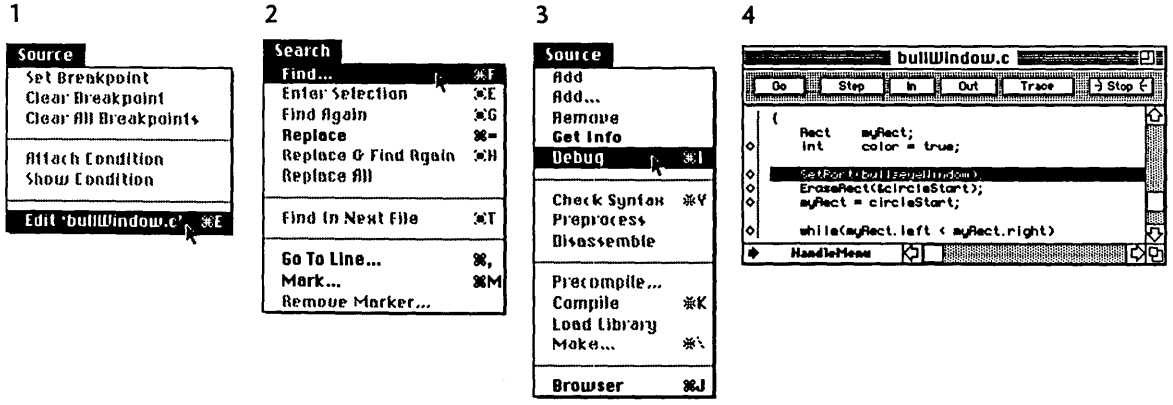


Figure 12-9 Searching from the source window

*See "Controlling Execution" on page 231 to learn about the **Go Until Here** command*

When the Source window displays what you're looking for, you'll usually want to use the **Go Until Here** command.

Setting Breakpoints

The THINK C debugger lets you set breakpoints at any diamond statement marker. When your program is running, the debugger stops execution just before a breakpoint.

You can set three kinds of breakpoints: simple breakpoints, conditional breakpoints, and temporary breakpoints. Execution always stops at a simple breakpoint. Conditional breakpoints let you use the value of an expression in the Data window to determine whether execution should stop. Temporary breakpoints let you set a breakpoint and start execution in a single step. When your program reaches a temporary breakpoint, execution stops, and the debugger automatically clears the temporary breakpoint.

You can set breakpoints while your program is running, not just when it's stopped.

You can also use the functions `Debugger()` and `DebugStr()` to force your program to stop and drop into the debugger.

Simple breakpoints

To set a breakpoint, click on a statement marker diamond. The diamond will turn from hollow to filled, indicating that the breakpoint is set.

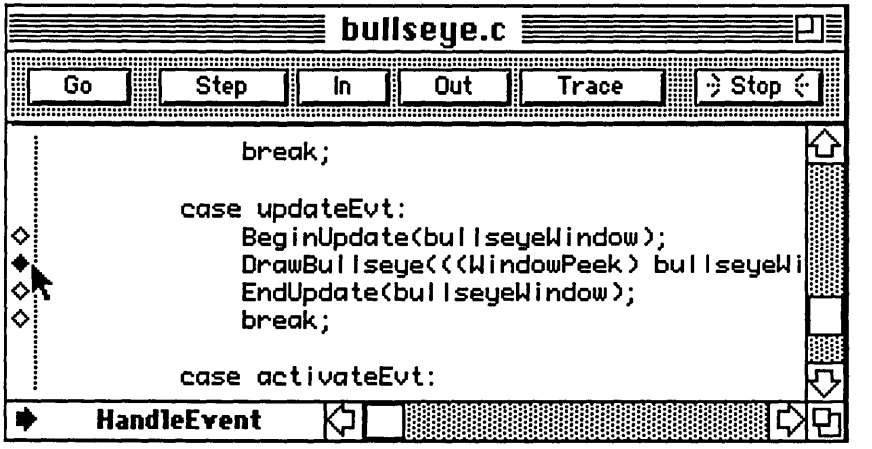


Figure 12-10 Setting a breakpoint

To clear a breakpoint, click on the filled diamond. The statement marker changes to a hollow diamond to show you that the breakpoint is clear. The **Clear All Breakpoints** command in the **Source** menu clears every breakpoint in every file.

You can also use the **Set Breakpoint** and **Clear Breakpoint** commands in the **Source** menu to set and clear breakpoints.

1. Click on a statement in the source window to select it
2. Choose **Set Breakpoint** or **Clear Breakpoint** from the **Source** menu to set or clear a breakpoint

Setting breakpoints in another file

The Source window usually shows the file that contains the current statement. To set breakpoints in another file, you must first tell THINK C to send the file to the debugger:

1. Click on the THINK C project window (Or choose your project name from the **Windows** menu)
2. Click on the name of the file you want to set the breakpoint in.
3. Select **Debug** (Command-I) from THINK C's **Source** menu. The source debugger's Source window will display the text of the file you chose.
4. Click on statement marker diamonds to set breakpoints.

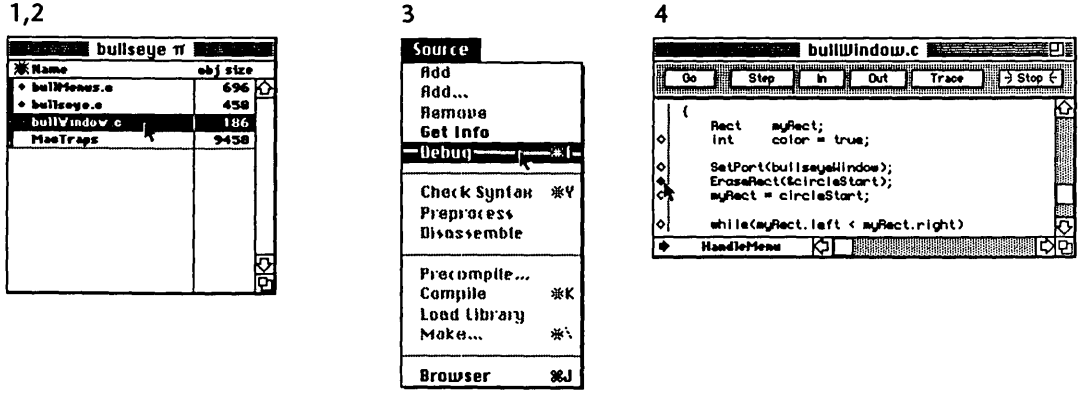


Figure 12-11 Setting a breakpoint in another file

To get back to the current file, click on the current function indicator at the lower left of the Source window or press the Enter key.

Setting temporary breakpoints

Temporary breakpoints are useful if you want to go quickly to a particular line of your program. For instance, suppose you wanted to examine how your program handled menu selections. You'd set a temporary breakpoint at the first line of your menu handling routine. The program would run normally until you chose a command from one of your menus.

To set a temporary breakpoint, hold down the Command or Option key as you click on a statement marker. When you release the mouse button, the debugger starts your program, and execution continues until you hit any

breakpoint, not just the temporary breakpoint. The source debugger clears all the temporary breakpoints when you stop for any reason.

Note

The **Go Until Here** command sets a temporary breakpoint at the selected line.

Setting conditional breakpoints

The THINK C debugger lets you set conditional breakpoints. A conditional breakpoint is a breakpoint that has a condition associated with it. The debugger stops execution at conditional breakpoints only when the condition evaluates to non-zero.

To set a conditional breakpoint:

1. Set a breakpoint by clicking on the statement marker diamond
2. Click on the line to select it
3. Click on an expression in the Data window
4. Choose **Attach Condition** from the **Source** menu.
5. The breakpoint marker turns into a gray diamond to indicate that it's a conditional breakpoint.

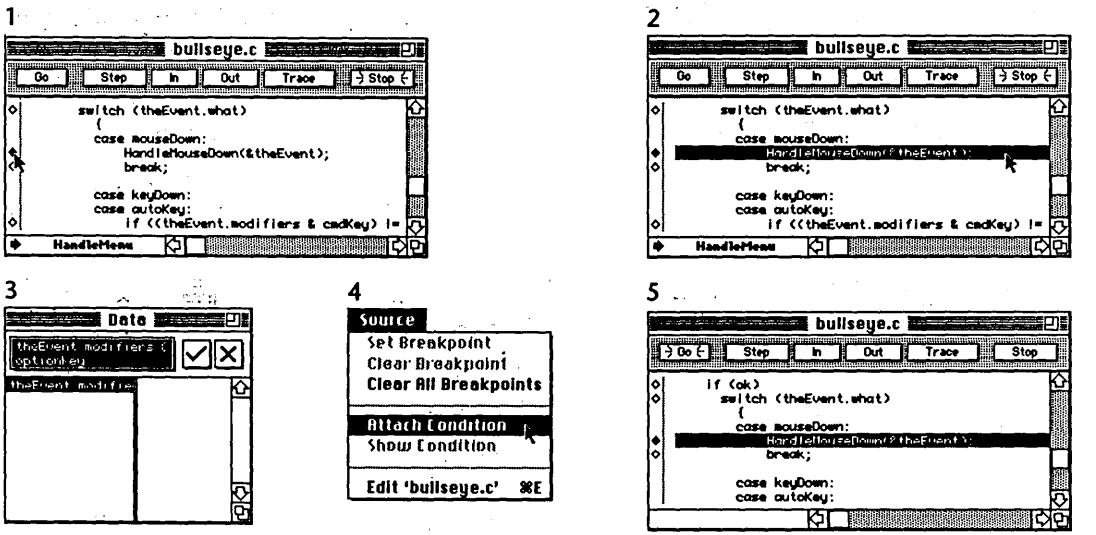


Figure 12-12 Setting a conditional breakpoint

In this example, the debugger will stop at the breakpoint only if you hold down the Option key as you click the mouse (`theEvent.modifiers & optionKey == 0x0800`).

If you're already stopped at a simple breakpoint that you want to turn into a conditional breakpoint, just select an expression in the Data window, and choose **Attach Condition**. Since the debugger uses the current statement if there isn't a selected line, the condition will be attached to the current statement.

The **Attach Condition** command is dimmed if:

- the expression can't be evaluated in the context of the breakpoint
- a breakpoint isn't set
- an expression in the Data window isn't selected
- the expression is a floating point expression

As you debug your program, you may forget the condition that's associated with the breakpoint. To see the condition associated with a conditional breakpoint:

1. Click on the line to select it
2. Choose **Show Condition** from the **Source** menu.

The condition associated with the conditional breakpoint will be selected in the Data window.

To learn how to use the Data window, see "Working with the Data Window" on page 234.

Using Debugger() and DebugStr()

The functions `Debugger()` and `DebugStr()` cause your program to stop and drop into the debugger just as if you had set a breakpoint. The `DebugStr()` function takes a Pascal string as an argument. The string appears in place of the status panel. For an example of `DebugStr()` see page 241.

If you're not using the THINK C debugger, these functions cause your program to drop into a low level debugger if one is installed.

Controlling Execution

The debugger uses six commands to control execution. To make it easier to debug your programs, there are three ways to use them: you can choose

them from the **Debug** menu, you can use the Command key equivalents, or you can use the buttons in the status bar of the Source window.

The buttons in the status panel do double duty as status indicators. When your program is running, the Go button is lit. When your program is stopped, the Stop button is lit. Remember that your program can still be running even if the debugger windows are frontmost.

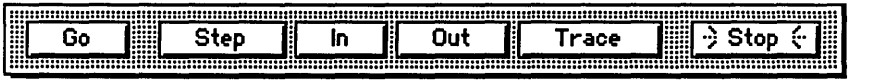


Figure 12-13 The status panel

Go

The Go command starts your program if it was stopped. Your program runs until you stop it (with the Stop button, for example) or until it's about to execute a line with a breakpoint or until it hits an exception (dividing by zero, for instance). If your application is already running, the Go command brings it to the foreground.

Trace

The Trace command executes the current statement. In most cases, the statement indicator will go on to the next statement marker, even if the next statement marker is in another function. The only time it won't is when the program counter steps into some code that the debugger doesn't have the source text for. This usually happens when you step into a trap that's not generated inline. So, for a brief period, the current statement arrow isn't really anywhere in your program, but somewhere in `MacTraps` instead. Though you can't see the current statement arrow, the current function indicator at the lower left tells you which file it's in.

Step

The Step command is like the Trace command except that it goes on to the next statement marker in the current function. If you're at the end of a function, Step returns to the calling function. Use Step when you want to follow the execution within a function without falling into another function. (Technically speaking, Step skips over JSRs.)

The first time you Step through a `for` loop, the current statement arrow jumps to the end of the loop because in the generated code, that's where the test is done.

Step In

The Step In command executes Trace commands until the current statement arrow falls into a function. Step In is useful when you want to skip over a set of assignments or Toolbox traps, to fall into the next function call. If Step In reaches the last statement of the current function without falling into another function, it will stop immediately after the function returns.

Step Out

The Step Out command executes Step commands until the current statement arrow falls out of the current routine. This operation can be slow if there's a lot left to do, but it's a sure way of leaving the current routine. A faster way of leaving the current routine is to use the **Go Until Here** command or to set a temporary breakpoint at the last diamond in the function.

Stop

The Stop command stops execution of your program. The Stop command works when any debugger window is active, and the Stop button only works when the source window is the active window. When you press the Stop button you'll usually be coming out of your call to `GetNextEvent()` or `WaitNextEvent()`.

If your program is not in its event loop, you might not be able to make the debugger the frontmost application. In this case, Command-Shift-Period is the **panic button**. Use Command-Shift-Period to stop execution when one of your application's windows is frontmost or when you think it's stuck in a loop.

Warning

Command-Shift-Period won't work if you're stuck in an infinite loop in ROM.

Go Until Here

The **Go Until Here** starts execution and stops at the selected line. This command is exactly the same as setting a temporary breakpoint (see "Setting Breakpoints" on page 227) at the selected line. Use this command when you want to move through a block of code quickly.

See "Searching in the Source window" on page 227 to learn how to look for text from the debugger.

This command is more convenient than setting a temporary breakpoint when the line you want to go to is already selected. For instance, it's easier to press Command-H after you've found a string you're looking for.

Skip To Here

The **Skip To Here** command changes the program counter to the selected line without executing any intervening code.Use it when you want to skip over code you know to be buggy but not crucial to the rest of the program's operation.

Warning

This command is potentially dangerous. Make sure the code you're skipping to doesn't depend on anything the skipped code does. For instance, it is a very bad idea to skip over initialization routines.

Stepping continuously

If you hold down the Option or Command key as you click on one of the status panel buttons (except Stop) you'll enter **auto mode**. In auto mode the debugger updates the Source and Data windows and repeats the command. To trace through every line of your program automatically, for example, hold down the Option key as you click on the Trace button.

One useful technique is to set breakpoints at spots where you'd like to look at some variables and then do an Auto-Go. When your program hits the breakpoint, the debugger will update the Data window and start the program up again.

To cancel auto mode, type Command-Shift-Period. The debugger will finish the command and then stop. Note that if you're in Auto-Go mode, the Stop button just cancels auto mode. You'll have to click on the Stop button again to stop your program.

Working with the Data Window

The Data window lets you examine and modify the values of your variables as you debug your programs. You can type any legal C expression in the entry field. The expression you type appears in the expression column, and its value appears in the value column. You can display values in several formats

to make your debugging easier. You can also display structs and arrays in their own windows.

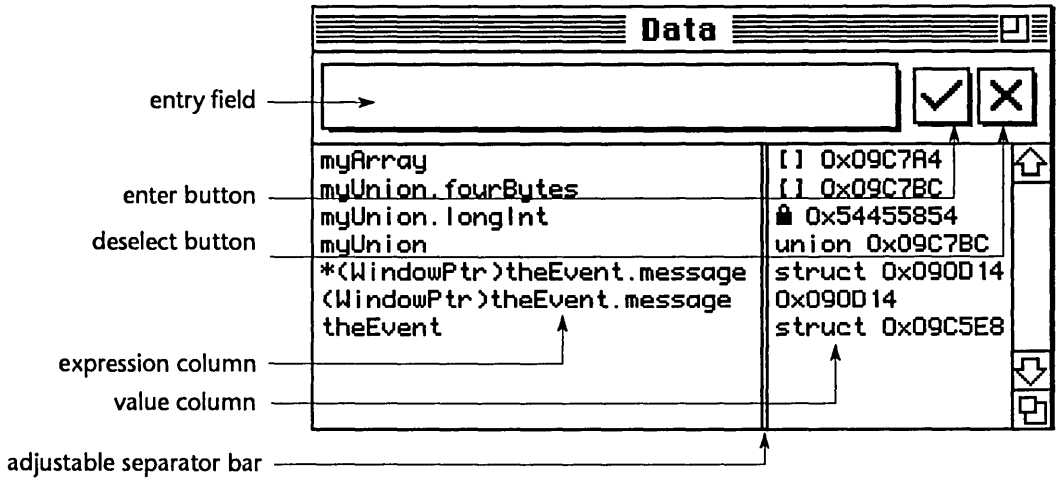


Figure 12-14 The data window

Entering an expression

You can enter any expression that does not have a potential side effect. That means you cannot enter assignment statements, function calls, or expressions involving ++, --, +=, etc.

The debugger compiles the expressions you enter in the context of the selected line in the Source window, or, if you haven't selected a line, in the context of the current statement. If the expression you enter is not defined within the context, the debugger displays an error message as the value of the expression.

The quickest way to enter an expression into the Data window is to select it in the Source window and choose the **Copy to Data** command in the debugger's **Edit** menu.

Expressions in the Data window have either local scope or global scope. An expression has local scope if it refers to variables with dynamic storage; in other words if it refers to non-static variables local to a function. All other expressions have global scope.

If you use the Enter key to enter an expression, the expression remains selected. If you use the Return key to enter an expression, the entry field is ready to accept the next expression. Clicking on the enter button is the same as pressing the Enter key.

Editing expressions

To edit an expression in the Data window, select it in the left column. The expression will appear in the entry. Use standard Macintosh editing techniques to edit the text of the expression.

When you edit an expression in the data window, the context is the same as when you entered the expression. To change the context of an expression to the current context after you edit it, hold down the Option or Command key as you click on the enter button.

Removing expressions

To remove an expression from the Data window, select the expression you want to remove, and choose the **Clear** command from the **Edit** menu or press the Clear key. To remove all the expressions from the Data window, use the **Clear All Expressions** command in the **Data** menu.

Formatting values

When you enter an expression, it's added to the left column. The value of the expression appears in the right column. You can use the formats in the **Data** menu to change how the debugger displays the variables.

To change a format, select an expression in the Data window. Then choose a format from the **Data** menu. Some of the formats won't be available. The formats available depend on the type of the expression. The default formats are shown in italics.

Type	**Formats Available**
integers	*Decimal*, Hex, Char
unsigned	*Hex*, Decimal, Char
pointers	*Pointer*, Address, Hex, C String, Pascal String
arrays	*Address*, C String, Pascal String
structs	*Address*
unions	*Address*
functions	*Address*
floats	*Floating Point*

This is what the display formats look like:

Format	Example
Decimal	`4523345, -23576`
Hex	`0xA09E1487`
Char	`'c', 'TEXT'`
C String	`"abcdef\nghi\33"`
Pascal String	`"\pabcdef\nghi\33"`
Pointer	`0x7A7000 or 0x007A7000`
Address	`[] 0x09FE44, struct 0x08FC14`
Floating Point	`1961.0102`

The C string and Pascal string formats display non-printing characters in backslash form. Whenever it can, the debugger uses the built-in escape characters (`\n`, `\r`, `\b`); otherwise it uses `\nn`, where nn is an octal value.

Of course, you can use type casting to use formats that aren't normally available. For example, if you really wanted to see an integer, `i`, as a C string, you would type this expression: `(char *) i`.

To see any pointer as an array, just change its format to Address. This way, when you double-click on its value, you'll see an array window instead of the value of what the pointer points to.

Displaying and changing contexts

If you forget the context of an expression in the Data window, use the **Show Context** command in the **Data** menu. The Source window will display the context for the expression.

When you select an expression, you can edit it in the entry field. When you press the enter button (or the Return or Enter keys), the debugger will recompile the expression in its original context.

To change the context of an expression you've already entered or edited, select the context in the Source window, select the expression, then choose **Set Context** from the debugger's **Data** menu. You can do the same thing by holding down the Option or Command key as you click on the enter button (or press the Return or Enter key).

Evaluating expressions

The debugger re-evaluates the expressions in the data window every time your program stops. Expressions whose context isn't in the current function are not re-evaluated unless they have global scope. To keep the Data window from getting too cluttered, the debugger clears their values.

Sometimes, you don't want an expression to be re-evaluated. For instance, you might want to compare the values of the same expressions at different times. Select an expression and choose **Lock** from the **Data** menu.

Setting values

The debugger lets you change the values of your expressions as long as the expression would be legal in the left side of an assignment statement.

Working with expressions

Double-clicking in the left column of the data window makes a copy of the expression. This is useful if you want to lock one copy down while you let another be evaluated every time the debugger stops.

Double-clicking on the value of struct, union or array opens up a new window.

Double-clicking on a value formatted as Pointer enters a new, dereferenced, expression in the Data window. If you hold down the Shift key, the new expression will be dereferenced twice.

If you hold down the Option key as you double-click, the new entry replaces the original entry.

Examining structs and arrays

You can examine fields of structs and arrays displayed in the Data window. (In this section, whatever you read about structs applies to unions as well.)

When you double-click in the right column on an expression whose value is struct, the debugger opens up a struct window. The struct window looks like the data window. The names of the fields appear in the left column, and their values appear in the right column.

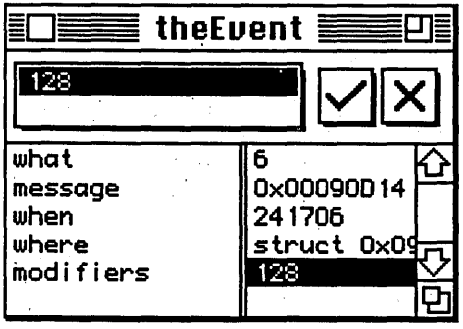


Figure 12-15 A struct window

You can change the values of the fields of the records the same way you change any variable. Of course, you can't change the names of the fields.

Double-clicking on a value opens other windows the same way as double-clicking in the main data window. A new expression appears in the main Data window for the new window.

Note

All struct and array windows "belong" to the main Data window. For every struct or array window there is an entry in the main Data window.

Double-clicking on a field name enters a new expression in the main Data window.

As mentioned above, Option-double-clicking replaces the original expression with the new one. For example: Suppose you had an struct window for an `EventRecord`, `theEvent`. Option-double-clicking on the `what` field would make the struct window disappear (the original expression is `theEvent`), and `theEvent.what` would appear in its place in the Data window.

Array windows are similar to struct windows. The indices appear in the left column and values appear in the right column. Unlike the main Data window or struct windows, every element in an array is displayed in the same format.

Because C compilers don't enforce array bounds, array windows have "infinite" scroll bars.

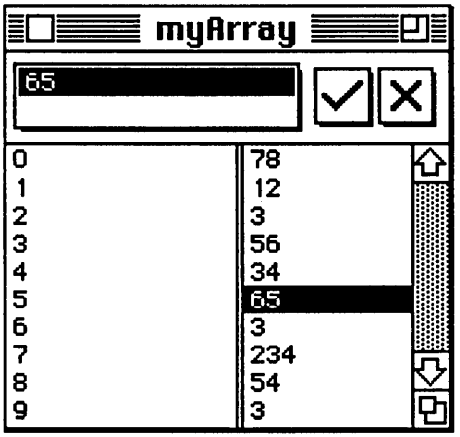


Figure 12-16 An array window

If you select an index in the left column and change its value, the debugger will display the array from that index.

To see a pointer as an array, set its format to Address and double-click on its value.

Error Messages in the Debugger

If executing a statement in the Source window or evaluating an expression in the Data window would cause a run time error, the THINK C debugger displays an error message in the status panel or in the value column of the Data window.

In Figure 12-17, the Source window contains a small program that tries to access illegal memory (a negative memory address, in this case). The data win-

dow has an expression that tries to divide by zero and one that tries to dereference the illegal pointer.

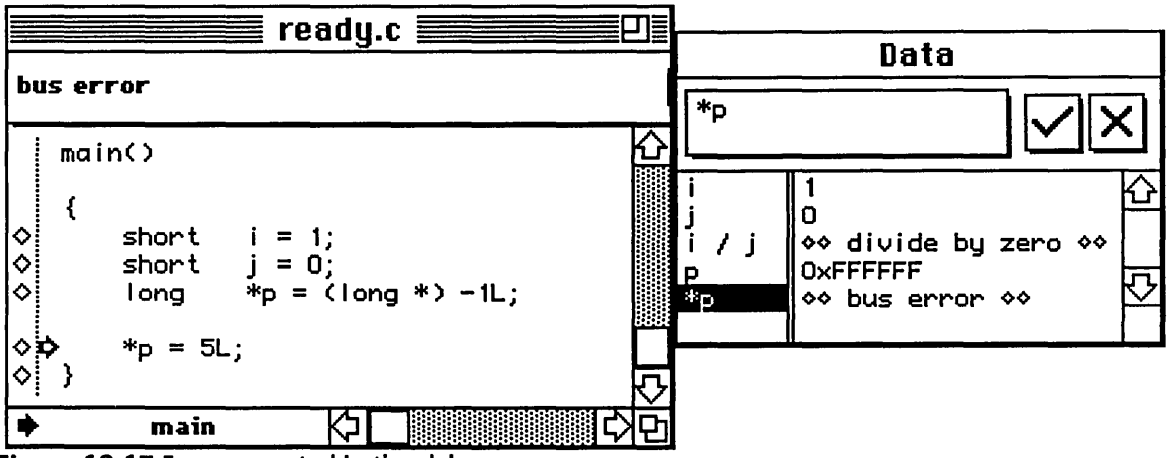


Figure 12-17 Errors reported in the debugger

In most cases, the safest thing to do when you get an error in the Source window is to choose the **ExitToShell** command from the **Debug** menu. If you want to continue from the error, click on the error message in the Source window, and the status panel will reappear.

If you use the `DebugStr()` function in your program your program stops, the string appears the same way an error message would:

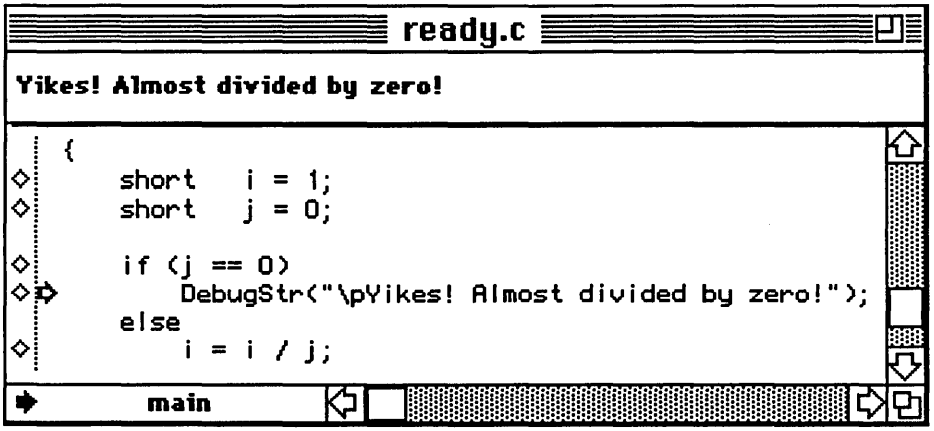


Figure 12-18 Using DebugStr() to report anomalous behavior

Saving the Debugger State

Use the **Save** command in the debugger's **File** menu to save all of your breakpoints, the expressions in the Data window, and the positions of all your debugger windows. Whenever you start the debugger again, it restores the debugger state. If you don't want the debugger to restore the state for a particular session, hold down the Option key as the debugger starts up.

To prevent the debugger from saving your state, hold down the Option key as you quit the debugger.

By default, the debugger does an automatic save when the "Always save session" option is on in the Debugging page of the **Options...** dialog. If this options is off, you can still use the **Save** command to save the debugger state. No matter how that option is set, the debugger always restores the previously saved state.

To put the debugger windows into their default positions, hold down the Shift key as you start the debugger.

If you want to make sure that the THINK C debugger doesn't save any session information, use the **Clear All Breakpoints** command in the **Source** menu and the **Clear All Expressions** command in the **Data** menu. Then choose **Save** from the **File** menu.

THINK C stores some debugger state information directly in your source files. As you edit your files, the THINK C editor maintains breakpoints and the contexts of data window expressions. However, other editors do not update the debugger state information. If you edit files with debugger information in other editors, your breakpoints will be set at the wrong lines, and the data window expressions will be evaluated in the wrong contexts.

Debugging Optimized Code

The code optimizer changes the way that THINK C compiles your programs. You should be aware that optimized programs might behave differently under the debugger than they would if they were not optimized.

For more information about the global optimizer and other optimizations see "Using the Global Optimizer" on page 184 and "Using Other Optimizations" on page 186.

If you use the global optimizer, certain statements might never get executed, so the compiler doesn't generate code for them. That means that you would not be able to set a breakpoint there. If you have register coloring on, and the optimizer uses the same memory location for two separate variables, changing one variable in the Data window might change another one as well. If you use common subexpression elimination, changing the value of a variable in the data window might not make a difference if the subexpression it belongs to has already been evaluated.

If you use the "Defer & combine stack adjusts" option, you should not use the **Skip To Here** command in the debugger to skip over a function call. If you use the "Suppress redundant loads" option, changes that you make to

values in the Data window might not "take" if the value was stored temporarily in a register.

In general, you can use the THINK C debugger to debug optimized code, but you need to be careful, and you need to be aware of how the optimizer changes your code.

Debugging Options

This section describes all the options that control the THINK C debugger. To set these options, choose the **Options...** command from the THINK C Edit menu, and turn to the debugging page.

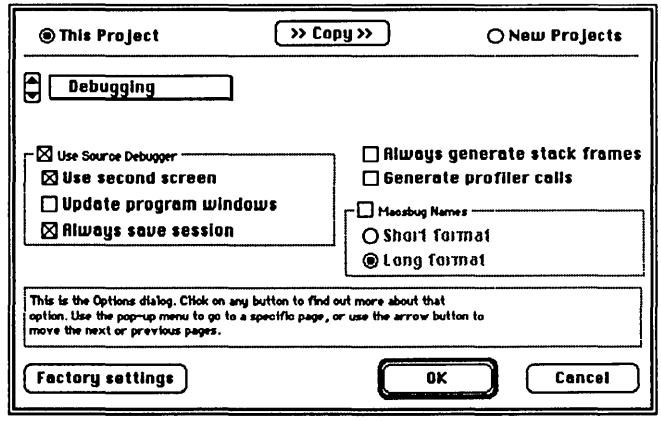


Figure 12-19 The debugging options

Use Source Debugger

This check box is the master switch for the THINK C debugger. If it's on, THINK C launches the source level debugger when you choose the **Run** command. Clicking in this check box is exactly the same as choosing **Use Debugger** from the **Project** menu.

Use second screen

If you have more than one monitor, the debugger use the second one to display the source and data windows. This option is available only when the "Use Source Debugger" option is on.

Update program windows

When this option is on, the debugger tries to update your windows for you when your project is stopped. If this option is off, your program must wait until control returns to it, so it can handle the update event itself. Since it

can't do so until it gets back to its event loop, an update may remain pending for some time.

This option is useful if you're trying to step through code that draws in a window, and the window is partially obscured by a THINK C or THINK C debugger window.

Since this window uses memory to save the window image, you might need to increase the THINK C debugger's memory partition if you're working on a large screen or in color. See "Memory Considerations" on page 246 for more information.

Always save session

If this option is on, the THINK C debugger saves the state of your debugging session before the debugger quits. It is as if you had chosen the **Save** command from the THINK C debugger's **File** menu. If this option is off, the debugger does not save the debugger state automatically at the end of each session. If you do a manual save, though, the debugger will restore that state when you start the debugger again. See "Saving the Debugger State" on page 242 for more information about saving sessions.

Always generate stack frames

If this option is on, THINK C always generates a stack frame (`LINK/UNLK` instructions), even for functions that normally wouldn't have one. If a function has no local variables and takes no parameters, THINK C does not generate a stack frame for it unless this option is on. Since the call chain pop-up menu relies on stack frames, you may want to turn on this option while you're debugging.

Note

If a function has no local variables, takes no parameters, and uses inline assembly, THINK C will not generate a stack frame for it, even if this option is on.

Generate profiler calls

When this option is on, THINK C generates code for the profiler. To learn about the profiler, see Chapter 15, "The Profiler." This option doesn't affect the operation of the THINK C debugger.

Macsbug Names

If this option is on, THINK C generates names for low level debuggers like Macsbug so they can find the name of a function. There are two varieties of Macsbug names:

Short format	Use only the first eight characters of the function name. All characters are translated to upper case.
Long format	Use all the characters of the function name. Case is preserved.

THINK C generates Macsbug symbols only for functions that have stack frames. If you have functions in your program that normally wouldn't require a stack frame, you might want to turn on the "Always generate stack frames" options above.

Both kinds of Macsbug names appear right after the code for a function, so turning this option on will increase the size of your code. You should turn this option off when you build your application, desk accessory, device driver, or code resource.

Using Low Level Debuggers

The THINK C source level debugger is great for figuring out what your application is doing. But sometimes you need to get closer to the machine. The **Monitor** command in the **Debug** menu invokes a low level debugger like TMON or Macsbug.

When you use the **Monitor** command, all registers and low memory globals contain the proper values for your program. The PC (program counter), however will not be pointing to the next instruction. Instead, it will be pointing somewhere in the debugger.

Warning

If you don't have a low level debugger installed, don't use the **Monitor** command.

Using the Monitor command with TMON 2.8.x

If you're using TMON 2.8.x, the value of the PC will be in TMON's V register. It's a good idea to keep a TMON disassembly window anchored to V so you can see your program's code when you use the Monitor command.

If there was an expression or value selected when you dropped into TMON, the N register contains that value.

To return to the THINK C debugger, use TMON's Exit command.

Using the Monitor command with other debuggers

If you're using another low-level debugger, like Macsbug, the correct value of the PC is right before the current PC. For instance, to display your program's code in Macsbug, type:

```
DM PC-4
IL @.
```

The value of the current Data window selection is not available in other low level debuggers.

Leaving the low level debugger

If you entered the low level debugger through the **Monitor** command, use your debugger's standard exit command.

If you got into your low level debugger any other way, there is no simple way to return to the source debugger. You can use your low level debugger's ExitToShell to abort your program as long as it's the foreground application. Check the low memory global `CurApName (0x0910)`. If it contains the name of your program, go ahead.

Quitting the Debugger

The best way to quit the debugger is to quit your application. You should use the **ExitToShell** command in the debugger's **Debug** menu only when you can't use your application's **Quit** command.

Memory Considerations

THINK C, the source debugger, and your project each run in their own MultiFinder partition. The default partition sizes are:

Application	Partition Size
THINK C	1000K
THINK C Debugger	250K
project	384K

If you want to run other applications, or if you find you're running out of memory, you can change the partition sizes. Change your project partition first. If you still don't have enough memory, change the size of the THINK C partition. Finally, change the debugger's partition.

The default partition size for your project is 384K. Moderately sized projects don't usually need this much memory. You might want to try smaller values u can give more memory to other applications.

THINK C uses the most memory when it's compiling. If THINK C complains that it's running out of memory when you compile, close any open windows before rebuilding your project. The THINK C partition should be no smaller than 700K.

If you have the "Update program windows" debugger option on, the debugger partition might need to be bigger than 250K. You will definitely need to increase it if you're running with either large or color monitors. If this option is off, and you're running a small project, you can make the debugger partition as small as 200K.

To change the size of your project's partition, use the **Set Project Type...** command in the **Project** menu. You can change the partition sizes for THINK C and the debugger from their **Get Info...** boxes in the Finder. You can't change the partition size of an active application, so you'll have to quit THINK C first to change its partition size.

Assembly Language 13

Assembly language lets you write the most efficient code possible. Using it with THINK C is a powerful combination. This chapter tells you how to use assembly language in your THINK C programs. You can use THINK C's built in inline assembler, or you can use object files generated by other assemblers. This chapter also describes C and Pascal calling conventions so you can write well-behaved assembly code.

What you should know

You should know the assembly language for the MC68000 family of microprocessors. Motorola publishes the *M68000 Family Programmer's Reference Manual* which contains descriptions for the MC680x0 processors, and the MC6888x floating-point coprocessor.

Contents

Using the Inline Assembler

THINK C has in integrated **inline assembler** that lets you write assembly language in your C routines. To use the inline assembler, you must set the "Language Extensions" option on the "Language Settings" page of the **Options...** dialog to "THINK C" or "THINK C + Objects".

You can use instructions for the Motorola MC68000 and MC68020 processors and for the MC68881 floating point coprocessor. The inline assembler lets you refer to C variables and functions within assembly language routines. Your C routines can `goto` labels in the assembly routines and vice versa.

Note

The inline assembler doesn't support any instructions that are unique to the MC68030 or the MC68040.

Writing assembly language code

The `asm` keyword invokes the inline assembler. The syntax for mixing assembly language statements with C code is simple:

```
asm {
    ...     /* assembly instructions   */
    ...     /* one per line            */
}
```

The compiler treats this construct as a C statement. It can appear anywhere a C statement can appear, which means that it must appear within a function.

Use only one assembly language instruction per line. You can use semicolon style comments, but since you're still writing in C, you can use C style comments as well. Preprocessor symbols and macros are expanded within assembly language, just as they are within C. You can use a C constant expression wherever a constant would be legal in assembly language.

The MC68881 and MC68882 are also called "floating point units".

The inline assembler supports all the standard MC68000, MC68020, and MC68881 instructions and addressing modes. THINK C does not support the assembly language extensions for the MC68851 memory-management unit, so you can not use the memory management instructions available on the MC68030 or the MC68040.

Note

Using only MC68000 assembly language ensures that your code will run on all Macintosh computers.

The inline assembler follows assembly language syntax conventions with a few exceptions. The DC (define constant) directive, which places literal values in the code stream, is the only assembly language directive the inline assembler recognizes. Use C to declare data, to define symbolic constants, and to import and export symbols.

The following identifiers are reserved in an assembly block.

d0	d1	d2	d3
d4	d5	d6	d7
a0	a1	a2	a3
a4	a5	a6	a7
fp0	fp1	fp2	fp3
fp4	fp5	fp6	fp7
sp	pc	zpc	ccr
sr	usp	isp	msp
vbr	sfc	dfc	cacr
fpcr	fpsr	fpiar	dc

The assembler is not case sensitive with respect to instruction mnemonics and size specifications (`.B`, `.W`, `.S`, `.L`, `.P`, `.X`, and `.D`). Register names can be either all uppercase or all lowercase, but not mixed case. For example, the inline assembler recognizes `fp1` and `FP1`, but not `Fp1`.

When THINK C compiles a function that contains an assembly block, it automatically turns off some optimization and debugging options for that function. It returns the options to their original settings when it compiles the next function. The following options are turned off:

- Automatic Register Assignment
- Defer & combine stack adjusts
- Always generate stack frames
- All global optimizer options

Writing processor-specific assembly code

The inline assembler supports all the standard MC68000, MC68020, and MC68881 instructions and addressing modes. However, to make sure your code runs on a certain Macintosh, you may need to specify which instructions and addressing modes are allowed in your assembly block. There are two ways to do this.

Warning

It's up to you to make sure that a specific processor is available before you allow your program to execute code for it. Use the Gestalt Manager to find out what processors a machine has.

To learn about the Options... dialog, see "Using the Options... Dialog" on page 179.

Usually, you'll use **Options...** dialog to make your selection. The options that affect assembly code are the "Generate 68020 instructions" and "Generate 68881 instructions" options in the Compiler Settings page of the **Options...** dialog. These options specify the type of assembly you can use in every `asm { ... }` block.

This table shows how to set these options for a certain type of assembly code:

If the 68020 option is	and the 68881 option is	You can write this type of assembly code...
Off	Off	MC68000 assembly.
On	Off	MC68020 assembly.
Off	On	MC68000 and MC68881 assembly.
On	On	MC68020 and MC68881 assembly.

Note

You cannot use MC68020 instructions or addressing modes when the "Generate 68020 instructions" option is off.

Sometimes, you need to override the settings in the **Options...** dialog. For instance, your program may be written for the MC68000, but if there is a MC68020 available, you want certain routines to take advantage of it. You can specify which processor an assembly block is for by putting the processor's name between the `asm` keyword and the opening brace. For example, this block is for the MC68000:

```
asm 68000 {
    ...      ; MC68000 instructions
}
```

You can specify that a block is for more than one processor, usually a specific CPU and the FPU (MC68881). Put the processor names, separated with

commas, between the asm keyword and the opening brace. For example, this block is for the MC68020 and the MC68881:

```
asm 68020, 68881 {
   ...      ; MC68020 and MC68881 instructions
}
```

This table shows the processor names that the inline assembler recognizes and what code you can write for that chip:

With this name...	You can write this code...
68000	MC68000 assembly.
68020	MC68020 assembly.
68030	MC68020 assembly. PMMU assembly is *not* supported.
68040	MC68020 and MC68881 assembly. PMMU assembly is *not* supported.
68881	MC68881 assembly.
68882	MC68881 assembly.

Note that the inline assembler doesn't support any instructions unique to the MC68030, MC68040, or MC68882. In fact, the names 68030, 68040 and 68882 are just synonyms for the other names.

Using C identifiers in assembly language

The inline assembler lets you use C identifiers directly. The base register (A6 for locals, A5 or A4 for globals, PC for labels) is optional, but must be correct if supplied.

If you've declared register variables, you can use their names wherever a register would be legal. The inline assembler is case sensitive with respect to C identifiers, just like the C compiler.

If you declare a variable to be stored in a register and THINK C can't store it in one, the assembler will signal an error when you use the variable. For example, this code fragment might signal an error:

```
void foo (void)
{
   register long  *a, *b, *c, *d;
   ...
   asm {
      move.l   (d), d0
      ...
   }
}
```

You can reference fields of a `struct` directly. For example, if you have a variable of type `WindowRecord`, you can get the `refCon` field like this:

```
long myRefcon(void)
{
   extern WindowRecord myWindow;

   asm {
      move.l   myWindow.refCon,d0
   }
}
```

You can get the offset of a field of a structure in two ways. The simplest is to use the name of the structure, followed by a dot, followed by the name of the field. For example, this function returns the `refCon` field of a `WindowRecord` that you have a pointer to:

```
long GetRefcon (WindowPtr wp)
{
   asm {
      movea.l      wp,a0
      move.l       WindowRecord.refCon(a0),d0
   }
}
```

The other way is to use the `offsetof()` macro defined in the file `stddef.h`. This function also returns the `refCon` field of a `WindowRecord` that you have a pointer to:

```
#include <stddef.h>  // for the offsetof() macro

long GetRefcon2 (WindowPtr wp)
{
   asm {
      movea.l  wp, a0
      move.l   offsetof(WindowRecord,refCon)(a0),d0
   }
}
```

Note

The macro `OFFSET()` defined in `asm.h` works the same way as `offsetof()` and is available for compatibility with previous versions of THINK C.

Using floating point literals

You may use valid C floating point literals as immediate-mode operands to MC68881 instructions. When you need to specify an immediate operand that

is accepted by MC68881 but that is not a valid C floating point literal, enclose the operand in quotes. For example, in the following statement `"INF"` stands for the MC68881 representation of infinity:

```
fmove.x  #"INF",fp2
```

In the following statement, if you don't use the quotes, the assembler would attempt to store the number as an integer and cause an overflow. The quotes tell the assembler to treat it as a double float:

```
fmove.x  #"123456678901234567890",fp1
```

With `.0` appended, the number is a valid C floating-point literal and doesn't need quotes:

```
fmove.x  #12345667890123456789O.0,fp1
```

Labels and branching

You can label assembly language instructions with C or assembler labels. An assembler label consists of an at sign (@) followed by one or more alphanumeric characters. Colons are optional following assembler labels, but must appear after C labels.

```
void foo (void)
{
   asm {
            …
@123        …  /* A legal asm label */
@2_px:      …  /* A legal asm label */
here:       …  /* A legal C label   */
   }
}
```

The scope of an assembler label is the enclosing function, not just the sequence of inline assembly in which it appears. This is the way C labels work, too.

You can use these C branching statements within assembly language:

Statement	Action
`break`	exits the surrounding loop or switch
`continue`	skips to next iteration of the surrounding loop
`return`	exits function
`goto label`	same as `bra @label`

You can `goto` C labels in assembly language from C code.

You can also refer to C labels from inline assembly, whether the label appears in assembly code or in C code. The label must be preceded by `@` to indicate to the assembler that it is a label. This avoids ambiguity in statements such as: `lea foo, A1`.

The assembler optimizes branch instructions without size qualifications (such as `BNE`), choosing whether to generate a long or a short branch. This optimization does not apply to instructions with size qualifications, such as the `BNE.S` or `BNE.W` instructions.

The assembler also optimizes branches around C statements appearing within inline assembly. For example,

```
while (...) {
    ...
    asm {
        bne    @1
        continue
@1      ...
        ...
    }
}
```

generates the same code as:

```
while (...) {
    ...
    asm {
        beq    @cont
@1 ...
        ...
    }
cont:
}
```

As above, this optimization applies only to instructions without size qualifications; for example, it apples to the `BNE` instruction, but not the `BNE.S` or `BNE.W` instructions.

If you JSR to a function from within assembly language, the function must be already declared. For example, to call the function MyFun() from assembly:

```
extern int MyFun(void);

int OtherFun(void)
{
    ...
    asm {
        ...
        jsr    MyFun
        ...
    }
}
```

It's up to you to make sure that you pass the correct arguments for a function you call from assembly language. For more information, see "C Calling Conventions" on page 266 and "Pascal Calling Conventions" on page 269.

Returning from a function

Do not use the RTS instruction unless you're absolutely sure you know what you're doing. If you want to return a value, always use a `return` statement. This way, the C stack frame will be cleaned up properly. For more information, see "C Calling Conventions" on page 266 and "Pascal Calling Conventions" on page 269.

Multiple entry points

You can create multiple entry points into a function by using labels declared as `extern`. First, declare each additional entry point to be a new function. Then, inside an inline assembly block, mark the beginning of each additional entry point with a label preceded by `extern`. The names of the label and

the new function must be the same. For example, this code fragment creates an additional entry point into `foo()` called `foo1()`:

```
void foo1(void);

void foo(void)
{
   asm {
         moveq #0,d0
         bra.s @1
extern foo1:
         moveq #1,d0
   @1    ...
         ...
   }
   ...
}
```

Warning

Use this feature carefully! It is very powerful, but improper use can wreak havoc.

Using the Macintosh Toolbox in assembly language

In your assembly language code, you can use any of the Macintosh Toolbox functions defined in the headers included with THINK C. The inline assembler is case sensitive with respect to trap names. If you like, you can precede trap names with an underscore.

Note

If you include `Traps.h` in your source file, you can't precede trap names with an underscore, because `Traps.h` uses the `#define` preprocessor directive to define all the trap names as constants.

Previous versions of THINK C had a built-in list of trap names and did not require you to include header files for the inline assembler.

To use Toolbox routines, you must either include `MacHeaders`, or the header file `asm.h` as well as the header file for the Toolbox routines you plan to use.

The inline assembler uses inline function definitions to generate instructions for Toolbox traps. For instance, two of the Text Edit routines are defined like this in `TextEdit.h`:

```
pascal void TECopy(TEHandle hTE) = 0xA9D5;
pascal Point TEGetPoint(short offset
   TEHandle hTE) = {0x3F3C,0x0008,0xA83D};
```

This example and the ones that follow are meant to show how the inline assembler generates code. Normally, you would not call these routines like this.

The inline assembler lets you use these two routines like this:

```
void F(void)
{
   asm {
            _TECopy
            _TEGetPoint
   }
}
```

It would generate machine code something like this:

```
DC.W     0xA9D5       ; trap for _TECopy
MOVE.W   #8, -(SP)    ; selector for _TEDispatch
DC.W     0xA83D       ; trap for _TEDispatch
```

This mechanism lets you treat certain Macintosh routines as if they were Toolbox traps even though they're really calls to dispatched traps or packages. The THINK C inline assembler takes care of calling such routines the right way. For instance, the inline assembler lets you treat the `SFGetFile` Toolbox routine as if were a Toolbox trap:

```
asm {
   ...
   _SFGetFile
   ...
}
```

What the inline assembler actually generates is this:

```
asm {
   ...
   move.w   #2,-(SP)     ; 2 is the routine
                         ; selector for
                         ; SFGetFile
   _Pack3                ; The trap for Std File
   ...
}
```

There are a few routines that have inline definitions that the assembler treats differently. For instance, the `MaxMem` Memory Manager routines is defined like this:

```
#pragma parameter __D0 MaxMem(__A1)
pascal Size MaxMem(Size *grow)= {0xA11D,0x2288};
```

But when _MaxMem is used in the inline assembler, only the trap 0xA11D is generated. These special routines, as well as traps for routines that can only be called with glue in C, are defined in asm.h.

The inline assembler lets you use the following 2-bit trap modifier flags to set bits 9–10 of certain traps. These trap modifiers are defined in asm.h:

Trap family	Modifier flag	Usage
Memory	SYS	Applies to system heap
	CLEAR	Zero allocated block
File & Driver	ASYNC	Asynchronous I/O
	HFS	Use HFS version of trap
	IMMED	Bypass driver queue
Strings	MARKS	Ignore diacritical marks
	CASE	Don't ignore case
Get/Set Trap	NEWOS	OS trap, new numbering
	NEWTOOL	Toolbox trap, new numbering
Toolbox	AUTOPOP	Return to caller's caller

This example is for illustrative purposes only. If you really wanted to allocate a cleared handle in the system heap, you'd use NewHandleSysClear *defined in* Memory.h.

Remember that these modifiers apply only to particular traps. Here's an example of a Toolbox routine call with trap modifier:

```
Handle NewClearSysHandle(size)
{
   asm {
      move.l      size,d0
      _NewHandle  CLEAR+SYS
      move.l      a0,d0
   }
}
```

Most of the low-level File Manager routines (the PB routines) may be called synchronously or asynchronously. When you call these routines from C, you provide a Boolean argument to specify which way you want to call the routine, or you can use a special inline definitions to call exactly the one you want:

```
void SamplePB (ParmBlkPtr pb, Boolean how)
{
   PBWrite(pb, how); /* call it one way      */
                     /* or the other         */

   PBWriteSync(pb);  /* Force it synchronous */
   PBWriteAsync(pb); /* Force it asynch.     */
}
```

In the inline assembler, you can call the synchronous or asynchronous traps directly. If you don't specify which version of the trap you want, the inline assembler assumes you meant the synchronous one.

```
void SampleAsmPB (void)
{
   asm {
         _PBWriteSync    ; use the synch version
         _PBWriteAsync   ; use the asynch version
         _PBWrite        ; same as _PBWriteSync
         _PBWrite ASYNC  ; same as _PBWriteAsync
   }
}
```

Register usage

You may modify the following registers in inline assembly code without saving and restoring their values:

```
A0    A1
D0    D1    D2
FP0   FP1   FP2   FP3
```

If you use any other registers, you should save and restore their values.

In general, if you need to use a register for other than scratch storage, you should declare a register variable and refer to it by name.

If intervening C code is executed between two stretches of inline assembly, you can assume that the C code preserves the values of registers `A5`, `A6`, and `A7` — as well as `A4` for drivers and code resources. All other registers may have been modified. It is safe to leave things on the stack while C code is executing, provided you clean up the stack before returning from the function.

For more information, see "C Calling Conventions" on page 266 and "Pascal Calling Conventions" on page 269.

Differences from Other Assemblers

This sections describes the differences between THINK C's inline assembler and most other assemblers.

Note

The inline assembler strictly enforces the syntax for MC68020 addressing modes. The previous version of THINK C allowed you to change the ordering of some operands.

Hexadecimal constants

THINK C does not accept the syntax `$NNNN` to designate a hexadecimal constant. Use the C syntax `0xNNNN` instead.

THINK C treats hexadecimal constants as unsigned numbers. For example, `0xFF` is equivalent to `255`, `0xFFFF` is equivalent to `65535`, and neither is equivalent to `-1`.

Directives

Two or more `DC.B` directives in a row are treated as one `DC.B` directive. This means that a `DC.B` directive can occupy an odd number of bytes when another `DC.B` directive follows. For example:

```
dc.b  'a'
dc.b  'b'
dc.b  'c'
dc.b  'd'
```

is equivalent to:

```
dc.b  'a', 'b', 'c', 'd'
```

However, preceding a `DC.B` directive with a label will word-align it. For example, this:

```
    dc.b  'a'
@1  dc.b  'b'
@2  dc.b  'c'
@3  dc.b  'd'
```

will assemble in the same way as:

```
dc.b  'a', 0, 'b', 0, 'c', 0, 'd', 0
```

The `DC.B` directive accepts strings (both Pascal and C) as operands. The terminating null byte is not assembled. Continuing the above example, this:

```
dc.b  'a', 'b', 'c', 'd'
dc.b  5, 'e', 'f', 'g', 'h', 'i'
```

is equivalent to this:

```
dc.b  "abcd"
dc.b  "\pefghi"
```

If you want a string defined as a C string, you must supply the terminating null. You can do it like this:

```
dc.b  "ABCDE\0"
```

The `DC.W` directive can accept a function name or a label as an operand and generate the PC-relative offset of the function or label as a word. The function or label must be defined within the segment in which the directive appears.

The `DC.S`, `DC.D`, and `DC.X` directives accept floating-point constants. For example:

```
dc.s  3.14            ; single-precision float
dc.d  "inf"           ; double-precision float
dc.x  "10000000000"   ; extended-precision float
```

Addresses

The difference of two addresses is not a constant expression, so instructions like this are not allowed:

```
move.w   #@2-@1,d0
```

Similarly, the inline assembler does not support the following syntax to assemble the PC-relative offset of the label `@1`:

```
dc.w     @1-*
```

Instead, use this:

```
dc.w     @1
```

For example, to code a dispatch table, use:

```
; d0 contains 0,1,2,...
   add.w d0,d0
   add.w @0(d0.w),d0
   jmp   @0(d0.w)
@0 dc.w  @1        ;  case 0
   dc.w  @2        ;  case 1
   ...             ;  ...
```

Tips

Inline assembly can be tricky if you are not familiar with assembly language. It can be especially dangerous if you're used to thinking in terms of high-level languages. The following problems are not specific to THINK C; they are common to assembly language in general.

Using constants

Don't forget the # sign when using an immediate constant. Otherwise, the result will be *very* different than what you intended.

```
extern short MemErr : 0x220;  // declare MemErr
                              // low mem global

  asm {
   move.w   0x220,D0     ;Moves contents of
                         ; location 0x220
                         ; (MemErr) into D0
   move.w   MemErr, D0   ;Same thing, but
                         ; symbolically
   move.w   #0x220, D0   ;Moves the value 0x220
                         ; into D0
   move.w   5, D0        ;WRONG: On a MC68000
                         ; this will cause an
                         ; odd address error.
                         ; On a MC68020 this will
                         ; move the contents of
                         ; location 5.
   move.w   #5, D0       ;RIGHT
}
```

Local storage

Use C variables to declare local storage. If you use the directive DC instead, storage will be allocated in your code segment. **Most of the time, this is not what you want.**

Instruction size

Be sure to use the right-sized instruction to refer to variables. Example:

```
function()
{
   short GetsTrashed;
   short v;

   asm{
      move.l   #3,v      ;WRONG: will overwrite
                         ; variable GetsTrashed
      move.w   #3,v      ;RIGHT: Word size
                         ; matches int.
   }
}
```

C Calling Conventions

Most of the time, the functions you write will be C functions. They will follow C calling conventions for placing arguments on the stack and returning values in register D0.

C calling sequence

*To see how THINK C compiles your code, use the **Disassemble...** command, described in "Disassembling your code" on page 168*

The caller pushes the arguments in right-to-left order, then calls the function. When the function returns, it's the caller's responsibility to remove the arguments from the stack. The caller's code looks something like this:

```
move   ...,-(SP)   ;last argument
...
move   ...,-(SP)   ;first argument
jsr    function
add    #...,SP     ;total size of arguments
```

The function's code looks something like this:

```
link   A6,#...     ;(optional)
...
move   ...,D0      ;result
unlk   A6          ;(optional)
rts
```

For more information on the "Far CODE" option, see "Using a larger jump table with Far CODE" on page 100.

When the "Far CODE" option is on, the code generator generates 32-bit absolute addresses for the jump table entry of a non-static routine. If the called routine ends up in the same segment as the calling routine, the linker replaces the 32-bit address with a 16-bit relative address from A5 and a 16-bit NOP instruction. This replacement makes it more likely that the linker can remove the jump table entry for that function.

C function entry

The arguments to the function appear on the stack in right-to-left order. The first (leftmost) argument appears just above the return address, followed by the remaining arguments.

The stack looks like Figure 13-1 on entry (just after the call).

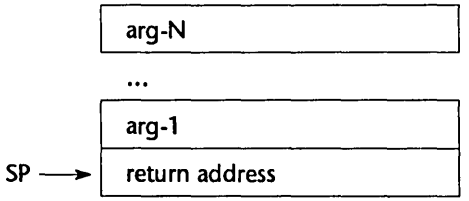


Figure 13-1 The stack during a C function entry

The first argument can be found at `4(SP)`. If the function begins with a `LINK A6,#...` instruction, the first argument can also be referenced by `8(A6)`.

All arguments occupy an even number of bytes on the stack. If there is no prototype for a `char`-size argument, it is placed in the high byte of an `int` which may be 2 or 4 bytes, depending on your option settings.

If there is a prototype for a `char`-sized argument, it is placed in the low byte of a `short`.

C function exit

C functions return their result (if any) in register `D0`. The result may be 1, 2, or 4 bytes long. Unused high-order bits may contain garbage.

The stack looks like Figure 13-2 on exit (just after the return).

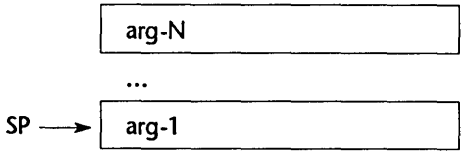


Figure 13-2 The stack during a C function exit

It is the caller's responsibility to remove the arguments from the stack.

Functions that return struct, union, or double

THINK C uses a different method to return a result of type `struct`, `union`, or `double`, since values of these types are in general too large to fit in `D0`. (Some `structs` or `unions` may be small enough to be returned in `D0`, but the alternate method is used anyway.)

If the "Generate 68881 instructions" option is on, and the "Native floating-point" option is on, all floating point values are returned in register `FP0`.

After pushing the arguments, but before issuing the actual call, the caller pushes the address of the location where the return value is to be placed. This address appears at `4(SP)` (or `8(A6)`) and the first argument appears instead at `8(SP)` (or `12(A6)`). The function must get this address and store the result at the location it points to. The address is considered a hidden argument to the function, and it's the caller's responsibility to remove it from the stack.

Because a function returning a `struct`, `union`, or `double` expects its caller to have placed a hidden argument on the stack, it is essential that the caller do so! Therefore, even when you are not interested in the actual return value, always be sure that the function is declared correctly before calling it.

Note

To ensure that THINK C places the return value in the right place, always use a `return` statement.

Functions that accept a variable number of arguments

The C calling conventions make it easy to write functions that take a variable number of arguments. The first argument can always be found in the same place regardless of how many additional arguments are supplied. Because responsibility for removing the arguments from the stack lies with the caller, the function doesn't need to clean up the stack.

The standard ANSI library includes the `stdarg.h` header file, which provides a standard interface for all C programs that handle variable arguments. As an elementary example, here is a function that returns the minimum of an arbitrary number of integers. The first argument is the number of additional arguments passed and must be at least 1.

```
#include <stdarg.h>

int minimum (int count, ...)
{
   va_list xp; /* Declare the arg pointer    */
   int x, min;


   va_start (xp, count);    /* Initialize the */
                            /* pointer.       */
   min = va_arg (xp, int); /* Get the first  */
                            /* arg.           */
```

```
    while (--count) {
        x = va_arg (xp, int);/* Get the next    */
        if (x < min) min = x;/* arg.            */
    }
    va_end (xp);              /* Clean up.      */
    return (min);
}
```

Note

You don't need to add the ANSI library to your project to use the `stdarg.h` header file.

A function using Pascal calling conventions cannot accept a variable number of arguments, unless it is written in assembly language.

Pascal Calling Conventions

The Macintosh Toolbox expects that calls to it follow Pascal calling conventions. When you write functions that expect to be called as Pascal functions, be sure to use the `pascal` keyword when you define them. This section tells you how Pascal functions expect to be called.

Pascal calling sequence

To see how THINK C compiles your code, use the Disassemble... command, described in "Disassembling your code" on page 168

The caller pushes the arguments in left-to-right order, then calls the function. Upon return, the result (if any) may be found on the stack. The caller's code looks something like this:

```
CLR    -(A7)          ;reserve space for result
MOVE   #..., -(A7)    ;first argument
...
MOVE   #..., -(A7)    ;last argument
JSR    ...            ;go to the function
MOVE   (A7)+, ...     ;result
```

The function's code looks something like this:

```
LINK   A6, #...       ;(optional)
...
MOVE   ..., ...(A6)   ;store return result
. . .
UNLK   A6             ;(optional)
MOVEA  (A7)+, A0      ;return address
ADDQ   #..., A7       ;total size of arguments
JMP    (A0)
```

Pascal function entry

The arguments to the function appear on the stack in left to right order. The last (rightmost) argument appears just above the return address, followed by the remaining arguments in reverse order. If the function returns a result, space for it is reserved above the first argument. If the return value is 1 byte long, 2 bytes are reserved.

The stack looks like Figure 13-3 on entry (just after the call).

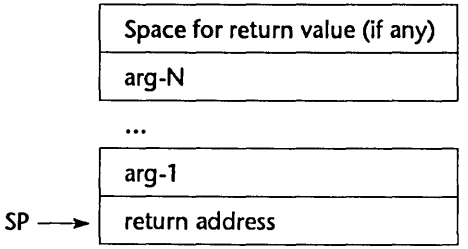


Figure 13-3 The stack during a Pascal function entry

The last argument can be found at `4(SP)`. If the function begins with a `LINK A6,#...` instruction, the last argument can also be referred to as `8(A6)`.

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 `SP` (or `A6`).

Pascal function exit

The function stores its return result (if any) on the stack in the location reserved by the caller. If the result is 1 byte long, it is placed in the high byte of the word reserved.

The stack looks like Figure 13-4 on exit (just after the return).

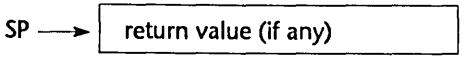


Figure 13-4 The stack during a Pascal function exit

It is the function's responsibility to remove the arguments from the stack

---

Note

To ensure that THINK C places the return value in the right place, always use a `return` statement.

Libraries 14

Libraries are collections of compiled code you can use in many programs. They usually contain utility functions or interface functions to the operating system. The `MacTraps` and `MacTraps2` libraries, for instance, contain the interfaces to the Macintosh Toolbox.

This chapter shows you how to use libraries, how to build a library, and how to convert object files from other development environments into THINK C libraries.

Contents

Using Libraries

To add a library to a project, choose the **Add...** command in the **Source** menu You see the dialog in Figure 14-1.

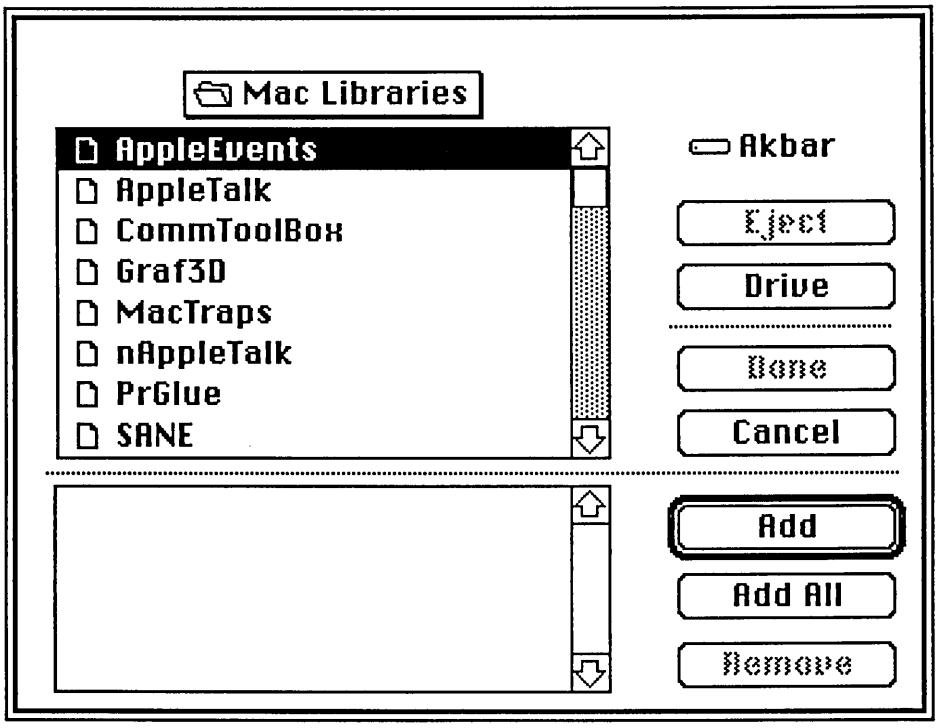


Figure 14-1 The Add... dialog

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 it.
- 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 it. (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.

When you add a library to a project, its object code isn't loaded, so its object code size is zero. You can use the **Load Library** command in the **Project** menu to load a library, or you can let THINK C's auto-make facility to load it. The auto-make facility loads a library if it was never loaded or if it changed since the last time it was loaded.

Creating Libraries

THINK C gives you two ways to create libraries. You can use any THINK C project as a library, or you can create a separate library document called a binary library. When you use a project as a library, the smart linker uses only the code it needs. Binary libraries, on the other hand, are smaller, and take up less space on disk.

Most of the libraries in your THINK C package are actually THINK C projects. The binary library format is included mostly for compatibility with older versions of THINK C.

Projects as libraries

See Chapter 7, "The Project," to learn about the different components of a THINK C project.

You can use any THINK C project document as a library. Libraries can contain functions, globals, or other libraries. When you use a project as a library, THINK C uses smart linking to build your project. The linker links only the CODE components that contain functions that your program uses

If your library contains global variables, and you want to use it in code resources, desk accessories, or drivers, you need to make sure that you compile everything in the library with A4 addressing. To make THINK C compile your globals with A4-relative addressing, choose **Set Project Type...** from the **Project** menu and click on the Code Resource radio button.

Note

When you choose Code Resource in this case, it doesn't mean that you're building a code resource or that the library will only work with code resources. It's just a way of letting the THINK C compiler know that you want A4-relative globals.

If your library doesn't contain any global variables, you can use it with either applications or code resources, drivers, or desk accessories. If your application does contain global variables, you'll need to create two versions: one for applications and one for everything else.

The ANSI libraries that come with your THINK C package are good examples of project libraries. If you want to explore them, make a copy of them

first so you don't damage the original ones accidentally.Then use the **Open Project...** command in the **Project** menu to open them.

Binary libraries

Use the **Build Library** command in the **Project** menu to create a binary library. When you use this command, you'll see a standard file dialog asking you to name the library. By convention, libraries end in `.lib`. When you use a library in your project, the linker adds all of the object code in the library to your project.

Binary libraries are seldom used. The facility to use and create them is provided for compatibility with older versions of THINK C and some third party products.

Converting MPW .o files into THINK C Projects

If you have an MPW .o file that you want to use in a THINK C project, use the oConv utility that comes with your THINK C package. This section gives you step-by-step instructions on how to use oConv. If you want more information about oConv, see Chapter 16, "oConv."

Converting an MPW .o file into a THINK C project is a two pass process. First you use the oConv utility to create a vocabulary file from the .o file. Then you edit the vocabulary file and use oConv again to create a project from the .o file and the edited vocabulary file.

To create a vocabulary file from a .o file, follow these steps:

1. Double-click on oConv to launch it
2. Make sure that both the TrapList and the ".v" file check boxes are checked
3. Choose the .o file you want to convert, and click on the Convert button
4. Click on the Cancel button to quit oConv

The oConv utility produces a THINK C project and a vocabulary file.Ignore the project file for this part of the process.

Note

There are some .o file records that the oConv utility cannot convert. If you get an error during this pass, look at Chapter 16, "oConv," to see if there is a way to work around the error.

The vocabulary file contains the name of every external function in the .o file. In most cases, these names are in all upper case. Use the THINK C editor or any other text editor to edit the vocabulary file so the function names have the correct upper and lower case spellings.

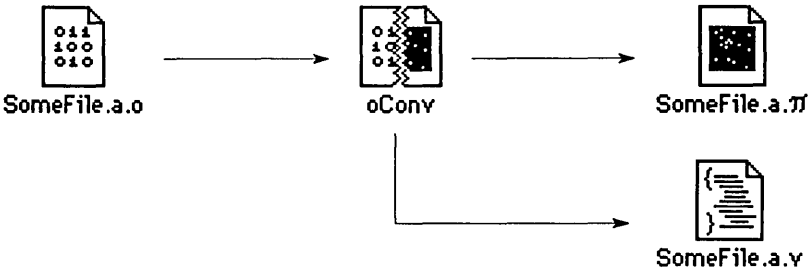


Figure 14-2 Creating a vocabulary file from an MPW .o file

Now that you have a vocabulary file, the second part is a little simpler.

1. Double-click on oConv to launch it
2. Make sure that both the TrapList and ".v" file check boxes are checked
3. Choose the same .o file that you chose before
4. Click on Cancel to quit oConv

In this second pass, oConv uses the vocabulary file that you created and edited in the first pass. The resulting project contains all the functions in the appropriate case.

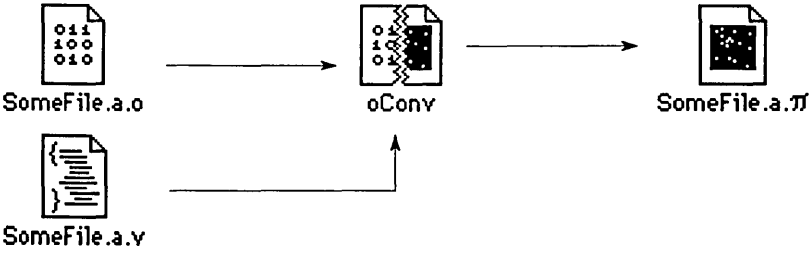


Figure 14-3 Creating a THINK C project from an MPW .o file

THINK C

Utilities

Part Five

The Profiler 15

Profiling your program lets you analyze how it runs. A profiler can log how much time your program spends in each function, and it can trace your program by printing the name of each routine as it's called. This chapter shows you how to use the THINK C profiler.

Contents

Using the Profiler

The profiler helps you analyze how your program runs. It can log how much time your program spends in each function, and it can trace your program by printing the name of each routine as it's called.

To use the profiler, follow these steps:

1. Check the "Generate profiler calls" check box in the Debugging page of the **Options...** dialog. When this option is on, THINK C inserts calls to profile routines at the beginning and end of your functions.
2. Include both the `profile` library and the `ANSI` library in your project. They're in the `C Libraries` folder. Make sure you `#include profile.h` in the file that calls `InitProfile()`.
3. Call `InitProfile()` in your program to start the profiler. This function takes two arguments. The first `nsyms` is the maximum number of functions that the profiler will track. The second `depth` is the maximum call depth. A good default for both arguments is 200 (for example, `InitProfile(200,200)`).
4. Set the variables `_profile` and `_trace` to control what the profiler does. `InitProfile()` initially sets `_profile` to non-zero and `_trace` to zero. If `_profile` is non-zero, the profiler logs time statistics. If `_trace` is non-zero, the profiler traces your routines.
5. If you want to see the timing statistics, make sure the profiler prints them out. You must call `DumpProfile()` or `exit()`, or use the console environment.

For example, this program initializes the profiler, writes a profile, and exits:

```
#include <profile.h>

main()
{
    /* Initialize the profiler with the */
    /* recommended arguments.           */
    InitProfile( 200, 200 );

    /* Have the profiler write a trace  */
    /* of your code                     */
    _trace = 1;

    /* Do some stuff.                   */
    . . .
```

```
    /* Write a profile.                    */
    DumpProfile();

    /* Do some more stuff.                 */
    . . .

    /* Exit. The profiler writes a         */
    /* profile automatically.              */
    exit(0);
}
```

The profiler won't collect statistics for some unusual functions. These conditions control whether the profiler can profile a function:

- Locals Variables. Does the function declare any local variables?
- Function Parameters. Does the function declare any parameters?
- Inline Assembly. Does the function use inline assembly?
- Stack Frames Option. Is the "Always generate stack frames" option in the **Options...** dialog on?

*For more information on the **Options...** command, see "Using the Options... Dialog" on page 179.*

When the conditions for a function match one of the cases in this table, the profiler won't collect statistics for the function:

Locals Variables	Function Parameters	Inline Assembly	Stack frames Option
none	none	present	on or off
none	none	none	off

To find out why the profiler doesn't profile these functions, see "Always generate stack frames" on page 244.

Logging Time Statistics

The profiler can log the amount of time spent in each function.To print these results to `stdout` at any time, call `DumpProfile()`. The profiler automatically prints out these results when your program exits, if your program does one of these:

For more information on the console environment, see the Standard Libraries Reference, *Chapter 3, "Using Console Windows."*

- Calls `exit()` to exit
- Uses the console environment.

For more information on `exit()` *and* `abort()`, *see the* Standard Libraries Reference, *Chapter 5, "The Standard Library Functions."*

Note

If your program exits by calling `ExitToShell()` or `abort()`, the profiler doesn't print its statistics automatically.

The profiler uses the units of the VIA 1 timer. Each unit is 1.2766 µsec (approximately 780,000 in a second). You can change the units to ticks (60ths of a second) by editing the profiler code. You might need to use the tick counter if your program uses the Sound Driver, since the Sound Driver uses the VIA timer. Also the VIA timer is less accurate if there's a lot of interrupt-level activity. See section "Changing which timer to use" on page 284.

This is an example of the profiler report. It's for a program that compares two sorting algorithms.

```
Function        Minimum   Maximum   Average     %    Entries

compare              26       238        27     4      12948
std_swap             64       948        67     2       2353
SlowSort            582   4880405   1643130    68          3
QuickSort            61        61        61     0          3
_QuickSort           57        58        57     0          3
QuickSort1           27    177573      1051     8        564
std_compare          85       436        88    15      12948
TryAlgorithms       364     24996      9294     0          3
```

This describes the report's headings:

Heading	Description
`Function`	Name of the function.
`Minimum`	Minimum time spent in routine.
`Maximum`	Maximum time spent in routine.
`Average`	Average time spent in routine.
`%`	Percentage of profiling period spent in the routine. (The profiling period is the accumulated time spent in routines that were compiled with the "Generate profiler calls" option on.)
`Entries`	Number of times the routine was called.

If you don't want the profiler to keep these statistics, set the variable `_profile` to 0 after you call `InitProfile()`.

Tracing Your Functions

The profiler can also print a trace of the functions your program calls as it runs. Every time you call a function, the profiler prints out the function's

name, indenting it to show how far nested it is. For example, this is an excerpt from the trace of the sorting program:

```
TryAlgorithms
 SlowSort
 QuickSort
  _QuickSort
   QuickSort1
    std_compare
     compare
    std_compare
     compare
    . . .
```

According to this trace, `TryAlgorithms()` called `SlowSort()` and `QuickSort()`, then `QuickSort()` called `_QuickSort()`, and so on.

If you want the profiler to trace your functions, set the variable `_trace` to a non-zero value after you call `InitProfile()`.

Modifying the Profiler

This section describes how to change where the profiler prints its reports and which timer the profiler uses.

Changing where to print

The profiler prints its report and trace to `stdout`. By default, `stdout` is a console window, a simple window that acts like a glass terminal. You can have the profiler print to both a console and a file, to both a console and a printer, or to just a file.

For more information on `cecho2printer()`, `cecho2file()`, *and* `freopen()`, *see* Standard Libraries Reference, *Chapter 5, "The Standard Library Functions."*

To print to...	Use this function...
a console and a file	`cecho2file()`
a console and a printer	`cecho2printer()`
just a file	`freopen()`

For example, these statements initialize the profiler and redirect `stdout` to the file `profiler report`:

```
InitProfile( 200, 200 );
freopen( "profiler report", "w", stdout );
```

Changing which timer to use

You can choose whether the profiler uses the VIA timer or the tick counter. The profile library shipped with THINK C uses the VIA timer.

Usually you'll want to use the more precise VIA timer Each of its units is 1.2766 µsec (approximately 1/780,000th of a second) and each tick unit is 1/60th of a second. However, you may need to use the tick counter in these two cases:

- You use the Sound Driver. The Sound Driver uses the VIA timer and will interfere with the profiler.
- Your program has a lot of interrupt-level activity. The VIA timer is less accurate in this case.

To create a profile library that uses the tick counter, follow the steps in section "Recompiling the profiler" on page 286. When you come to the step "Make your changes," comment out the line that defines `VIATIMER`. It should look like this:

```
/* #define VIATIMER  */
```

Changing the profiler's report

You can change the profiler's report. For example, you might alter how a field is computed or what it looks like. Follow the steps in section "Recompiling the profiler" on page 286. When you come to the step "Make your changes," edit the function `DumpProfile()`.

You'll need to edit two statements in `DumpProfile()`. The first is the `printf()` statement that prints a line of information for each function profiled. Here it is, with comments explaining the arguments:

```
printf("%#-32s%8lu %8lu %8lu   %3u %8lu\n",
   p->fname,                                    // Name
   p->min == LONG_MAX ? 0 : p->min,             // Min
   p->max,                                      // Max
   p->count ? p->total / p->count : 0,          // Ave
   total ? (int) (100. * p->total/total) : 0,// %
   p->count);                                   // Ent.
```

The second is the `printf()` statement that prints the report's heading. You need to edit it only if you change the length of a field:

```
printf("\n%-32s  Minimum  Maximum  Average     "
   "%%   Entries\n\n", "Function");
```

This example shows how to print the percentage as a floating-point number with five digits of accuracy, instead of as an integer. Firstly, edit the format

string argument in the first `printf()` statement. Since the percentage will be a floating point number, change its format specifier from `%3u` to `%5.2f`.

```
printf("%#-32s%8lu %8lu %8lu   %5.2f %8lu\n",
```

Secondly, change the argument that computes the percentage. Simply delete the `(int)` cast, which casts the argument to an integer, and change `0` to `0.0`.

```
total ? (100.* p->total/total) : 0.0, /* % */
```

Lastly, add some spaces to the report's heading, since the percentage field is longer. Add three spaces between `%%` and `Entries`:

```
printf("\n%-32s  Minimum  Maximum  Average     "
   "%%      Entries\n\n", "Function");
```

Recompiling the profiler

To change how the profiler works, you need to edit and recompile the `profile` project. These steps show you how:

1. In the Finder, create a new copy of the `profile` project and give it a new name, like `new profile`.
2. Open the new project with THINK C.
3. Open the file `profile.c`.
4. Make your changes.
5. Choose the **Save As...** command. Go to the `Sources` folder in the `C Libraries` folder. Save the file under a new name, such as `new profile.c`.
6. Choose the **Compile** command to recompile the file.

Warning

When you rebuild the profiler library, make sure that the "Generate profiler calls" option is not selected. You cannot profile the profiler. You would be in an infinite loop at run time.

Now you can use the new project as a library in any of your projects.

Summary

This section describes the functions and variables in the `profile` library. They're declared in `profile.h`.

User functions

Use these functions and variables in your program to control how the profiler works.

```
void InitProfile( unsigned nsyms, unsigned depth );
```

Start profiling. It takes these arguments:

Argument	Description
`nsyms`	The maximum number of functions to track
`depth`	The maximum call depth.

```
void DumpProfile();
```

Write the statistics collected so far to `stdout`. You can call this function any time to print the current statistics. The profiler automatically calls it when your program exits.

```
int _profile;
```

Variable. If non-zero, the profiler collects timing statistics as your functions execute. By default, `_profile` is non-zero. You can change the value after you call `InitProfile()`.

```
int _trace;
```

Variable. If non-zero, the profiler traces your functions Every time you enter a function the profiler prints its name to `stdout`. By default, `_trace` is zero. You can change the value after you call `InitProfile()`.

Internal functions

The profiler calls these functions before and after each function call. You shouldn't call them in your programs.

```
void _profile_( unsigned char *fname );
```

Called at the beginning of each function when the "Generate profiler calls" option is on. When this routine is called, the stack contains the address of a Pascal string that is the name of the function. `_profile_()` changes the return address of the routine to be `_profile_exit()`.

```
static void *_profile_exit();
```

Called at the end of a function. Code to execute this routine is not generated. Instead, `_profile_()` changes the stack so the original routine "returns" to this one.

oConv 16

From time to time, you may need to use an object file that was compiled with Apple's Macintosh Programmers Workshop. The oConv utility converts .o files into a form that THINK C can use.

Contents

Using oConv to Convert MPW .o Files

The oConv application converts object files created by Apple's MPW compilers and assemblers into THINK C projects. You can use these projects as libraries or, if you prefer, you can build a library from the resulting project

Note

"Converting MPW .o files into THINK C Projects" on page 275 gives you step-by-step instructions on how to convert MPW.o files into THINK C projects. If you want to convert a file quickly, look there first, then come back here to learn more about oConv.

When you double click on the oConv icon, you'll see a standard file dialog.

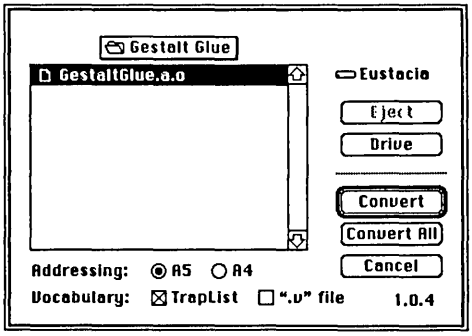


Figure 16-1 The oConv dialog

The converter displays only files of type 'OBJ '. That means that it can convert MPW .o files as well as THINK Pascal libraries. When you double click on a .o file name (or click on the Convert button), the converter generates a file ending in .π. For instance, if you convert a file called xyzzy.o, the resulting project file will be called xyzzy.π.

When the converter is through converting the file, it displays the standard file dialog again to let you convert more files. To quit the converter, click on the Cancel button.

To convert all the .o files in a folder, click on the Convert All button. The converter doesn't convert files that have already been converted.

The converter lets you choose whether to use A5 or A4 addressing. If you'll be using the project in an application, choose A5. If you'll be using it in a

desk accessory, device driver, or code resource, use A4 addressing. To learn more about A4 addressing, read "Global data in drivers" in Chapter 7.

In some .o files, symbol names appear in all upper case. The vocabulary mechanism provides a way to translate such names back to their proper capitalizations. Only names entirely in upper case will be translated.

When TrapList is checked, the Toolbox glue routine names are added to the vocabulary. This includes all the routines in *Inside Macintosh*.

When the ".v" file option is checked, the converter looks for a vocabulary file for each .o file. The vocabulary file is a text file that contains the proper capitalization for each symbol in the .o file, one symbol per line. If the file to be converted is named xyzzy.o, the vocabulary file must be named xyzzy.v.

If the vocabulary file doesn't exist, the converter will create it. The file will contain one line for each symbol that appears entirely in upper case in the .o file. You can edit this file to supply the proper capitalization, and then run oConv again.

Converting Large .o Files

If you want to convert an object file that has more than 32K of code and data, you need to break it up into smaller object files that oConv can handle. The best way to split up large object files is to use Apple's Macintosh Programmers Workshop. The instructions in this section give you one way of splitting up large .o files.

Create a list of symbols

The first thing you need to do is find out the names of the symbols in the object file. Use the MPW dumpobj tool to do this. Suppose that the object file you want to convert is called OFILE.o. Use this MPW command to get a list of symbols:

```
dumpobj OFILE.o -d -h -i -jn > OLIST.txt
```

The list in the output file, OLIST.txt, has size information about each symbol that you don't need to do the conversion. Use this MPW command to clean up the file:

This is how to type the special characters:
- ∞ *Option 5*
- • *Option 8*
- ≈ *Option x*
- ® *Option r*

```
replace -c ∞ /•≈[0-9]+ (≈)®1∞/ ®1 OLIST.txt
```

Split up the symbols

Now you need to split up the symbols into groups for the smaller object file. For example, suppose that `OLIST.txt` contains these symbols:

```
DrawThings
CalcThings
UtilThings
MemThings
```

To split up the large object file into three pieces, you need create three lists. Each list should contain the names of the symbols you *don't* want in the smaller piece. Make these lists by copying `OLIST.txt` and editing it. Call the resulting lists `filter1.txt`, `filter2.txt`, and `filter3.txt`.

Suppose you want to break up the large object file into these three pieces:

piece1.o	**piece2.o**	**piece3.o**
DrawThings	UtilThings	MemThings
CalcThings		

Your filter files would look like this:

filter1.txt	**filter2.txt**	**filter3.txt**
UtilThings	DrawThings	DrawThings
MemThings	CalcThings	CalcThings
	MemThings	UtilThings

Make the pieces

Now you can use the filter lists to create the pieces from the original object file. You need to use the MPW lib tool for each piece:

```
lib OFILE.o -o piece1.o -df filter1.txt -Sym Off
lib OFILE.o -o piece2.o -df filter2.txt -Sym Off
lib OFILE.o -o piece3.o -df filter3.txt -Sym Off
```

Use oConv to convert the pieces

Finally, you can use oConv to convert the smaller pieces.

Conversion Limits

Though oConv can convert many object files into THINK C projects, there are some object file records it can't convert.

Segmentation limits

OConv ignores all segmentation directives in an object file, and places all the converted code into a single segment. The code and data in an object file cannot exceed a total of 32K bytes.

Computed references

OConv only converts computed references (object records of type 10) that are 16-bit modified references to code. It will not convert 8-bit or 32-bit computed references to code or 8-, 16-, or 32-bit references to data.

Reference records

OConv cannot convert reference records (object records of type 9). This kind of code causes MPW to generate reference records:

```
extern short func();
short (*fp)() = func;    // data to code, 32-bit
                         // load-time addition
                         // of A5
short (**pfp) = &fp;     // data to data, 32-bit
                         // load-time addition
                         // of A5
```

SADE records

OConv ignores all object file records of type greater than 10.

MPW runtime routines

Sometimes, an object file will convert with no problems, but when you try to use it, you get an "undefined symbols" error message. If the object file was produced from a high level language, like MPW C or MPW Pascal, these references are usually calls to runtime routines for long multiplication and division. To use object files that make calls to MPW runtime routines, you may want to extract the runtime routines from the MPW runtime libraries.

THINK C

Resource Utilities

Part Six

Resource Description Files 17

Resource description files are the files compiled by SARez and produced by SADeRez. See Chapter 18, "Using SARez," and Chapter 19, "Using SADeRez," for more information on these utilities. Complete background information on Macintosh resource files is given in *Inside Macintosh I*, Chapter 5; *Inside Macintosh IV*, Chapter 3, *Inside Macintosh V*, Chapter 3; and *Inside Macintosh VI*, Chapter 13.

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 a powerful and useful utilities.

Contents

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, SADeRez, decompiles an existing resource, producing a new resource description file that can be understood by SARez. Figure 17-1 illustrates the complementary relationship between SARez and SADeRez.

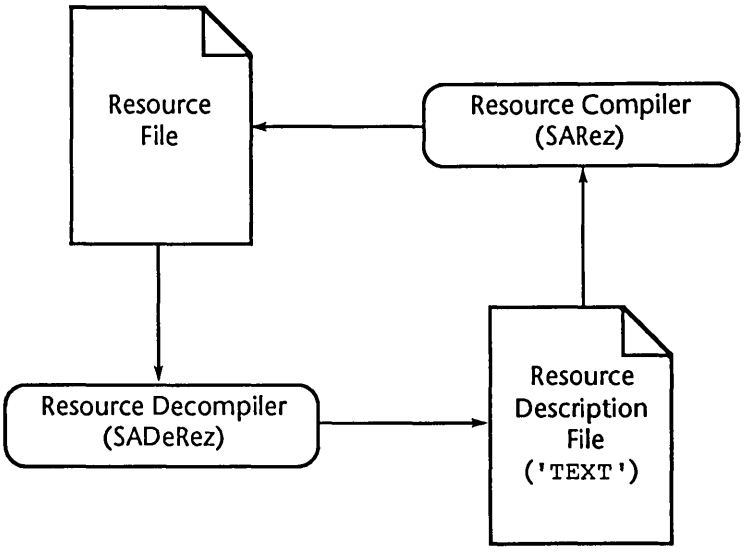


Figure 17-1 SARez and SADeRez

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 in section "Preprocessor Directives" on page 328.)

Resource decompiler

SADeRez 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 SADeRez takes the form of raw data statements.) The output of SADeRez 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.

This file...	Contains...
`Types.r`	Type declarations for the most common Macintosh resource types (`'ALRT'`, `'DITL'`, `'MENU'`, and so on).
`SysTypes.r`	Type declarations for `'DRVR'`, `'FOND'`, `'FONT'`, `'FWID'`, `'INTL'`, `'NFMT'`, and many others
`Pict.r`	Type declaration for `PICT` resources for debugging `PICT`s

Using SARez and SADeRez

SARez and SADeRez are primarily used to create and modify resource files. Figure 17-2 illustrates the process of creating a resource file.

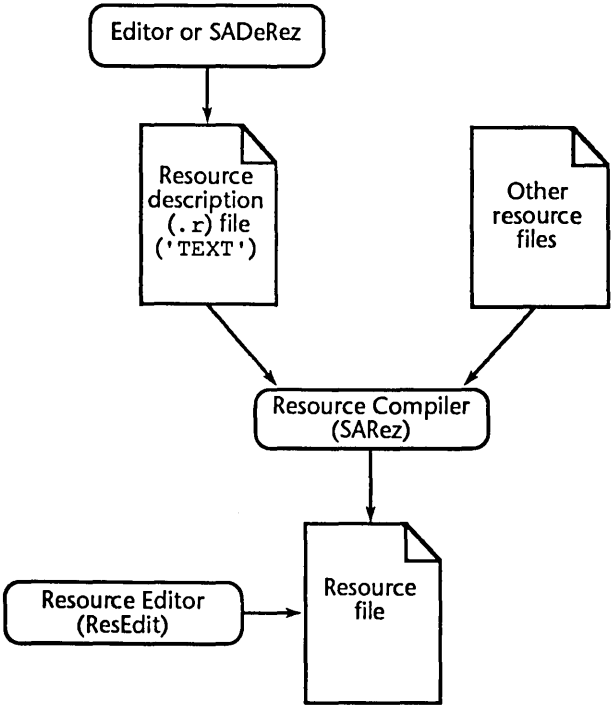


Figure 17-2 Creating a resource file

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' {        // Ignore rest of this line.
```

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 SADeRez 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;                              /* bounds     */
   integer documentProc,              /* procID     */
      dBoxProc, plainDBox,
      dBoxProc, 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      */
};
```

Type and resource statements are explained in detail in the reference section that follows. The SADeRez utility is explained in detail in Chapter 19, "Using SADeRez."

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.
nonterminal	Items in italics can be replaced by anything that matches their definition.
[*optional*]	Square brackets mean that the enclosed elements are optional.
repeated...	An ellipsis (...), when it appears *in the text of this reference only*, indicates that the preceding item can be repeated one or more times.
a | *b*	A vertical bar indicates an either/or choice
(*grouping*)	Parentheses indicate grouping (useful with the | and ... notation).
'[' *x* ']'	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 *not* mean that the *x* 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:

Expression *is defined later in this chapter in section "Expressions" on page 332.*

Term	Definition
resource-type	*long-expression*
resource-name	*string*
resource-ID	*word-expression*
ID-range	*ID* [: *ID*]

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

SADeRez 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:

Variable	Description
`$$Type`	Type of resource from include file
`$$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

To learn more about attributes, see Inside Macintosh I, *Chapter 3, "Resource Manager."*

You can specify **attributes** as a numeric expression, 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.
`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 SADeRez 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. SADeRez 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[`*n*`]`	
`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 in section "Expressions" on page 332.

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 SADeRez 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 SADeRez 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 (–); `0xFFFFFF85` and `-0x7B` 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 '[' length ']'`	Declare a bitstring of *length* 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:

`[hex] string`	**Plain string** has no length indicator or termination character is generated. The optional `hex` prefix tells SADeRez to display it as a hex string. `String[`*n*`]` contains *n* characters and is *n* bytes long. The type `char` is shorthand for `String[1]`.
`pstring`	**Pascal string** has a leading byte containing the length information is generated. `Pstring[`*n*`]` contains *n* characters and is *n*+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[`*n*`]` contains *n* characters and is *n*+2 bytes long.
`cstring`	**C string** generates a trailing null byte. `cstring[`*n*`]` contains *n*–1 characters and is *n* 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` ≤ *length* ≤ `2,147,483,647` for

`string` and `cstring`, and in the range `1` ≤ *length* ≤ `255` for `pstring`, and in the range `1` ≤ *length* ≤ `65,535` 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 SADeRez and C programs. However, the full string, including the explicit null and any text that follows it, will be stored by SARez as input.

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 ≤ `2,147,483,647`.

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. SADeRez 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 SA-DeRez)—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 SADeRez 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:
      booleanenabled, disabled;
      key bitstring[7] = 4;     /* key value */
      pstring;
   case CheckBox:
      booleanenabled, 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,              /* procID    */
      dBoxProc, plainDBox,
      dBoxProc, 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 stan-

dard Macintosh resource type, as shown in *Inside Macintosh VI*, Chapter 13 "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" sys-

tem 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, SADeRez 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, SADeRez 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

SADeRez 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

See Inside Macintosh *I, Chapter 9, "Window Manager," for information about resources in windows.*

This section describes a sample resource description file for a window. Here, again, is the type declaration given above in section "Sample type statement" on page 316:

```
type 'WIND'{
   rect;                              /* bounds    */
   integer documentProc,              /* procID    */
      dBoxProc, plainDBox,
      dBoxProc, 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) {
                    /* Status report window    */
   {40,80,120,300}, /* Bounding rectangle      */
   documentProc,    /* documentProc etc..      */
   Visible,         /* Visible or Invisible    */
   goAway,          /* GoAway or NoGoAway      */
   0,               /* Reference value RefCon */
   "Status Report"  /* Title                   */
   };
```

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 (`'\000'`). 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, 8, 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 the section "Declaring labels within arrays" on page 324 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:

```
( len("Neato") [5] + null byte [1] ) *
8 [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 SADeRez 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 SADeRez 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.

```
/* PixPat record */

type 'ppat' {
   integer oldPattern, newPattern, ditherPattern;
                     /* Pattern type                    */
   unsigned longint = PixMap / 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 */
PixMap:
   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;          /* # bits per field                */
   integer;          /* # components in pixel           */
   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.

```
/* IconPMap (pixMap) record */

type 'cicn' {
   fill long;           /* Base address                 */
   unsigned bitstring[1] = 1;
                        /* New pixMap flag              */
   unsigned bitstring[2] = 0;
                        /* Must be 0                    */
pMapRowBytes: unsigned bitstring[13];
                        /* Offset to next row           */
Bounds: 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;    /* Offset to color table        */
   fill long;           /* Reserved                     */
```

```
   /* IconMask (bitMap) record */
   fill long;          /* Base address                */
maskRowBytes: integer;
                       /* Row bytes                   */
   rect;               /* Bitmap bounds               */

   /* IconBMap (bitMap) record */
   fill long;          /* Base address                */
iconBMapRowBytes: integer;
                       /* Row bytes                   */
   rect;               /* Bitmap bounds               */
   fill long;          /* 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:

1. Each preprocessor statement must be expressed on a single line, beginning on a new line and terminated by a return character.
2. The pound sign (#) must be the first character on the line of the preprocessor statement (except for spaces and tabs).
3. Identifiers (used in macro names) may be letters (A–Z, a–z), digits (0–9), or the underscore character (_).
4. Identifiers may be any length.
5. Identifiers may not start with a digit.
6. 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 SADeRez.

The following predefined macros are provided:

Variable	Value
`true`	`1`
`false`	`0`
`rez`	`1` or `0` (`1` if SARez is running, `0` if SADeRez is running)
`derez`	`1` or `0` (`0` if SARez is running, `1` if SADeRez 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
```

Expression is defined in section "Expressions" on page 332. 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`:

```
#ifdef macro
#ifndef 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. These are the basic constants:

- **Decimal** (*nnn...*): Signed decimal constant between 4,294,967,295 and -2,147,483,648.
- **Hex** (`0X`*hhh...* or `$`*hhh...*): Signed hexadecimal constant between `0X7FFFFFFF` and `0X80000000`.
- **Octal** (`0`*ooo...*): Signed octal constant between `017777777777` and `020000000000`.
- **Binary** (`0B`*bbb...*): Signed binary constant between `0B11111111111111111111111111111111` and `0B10000000000000000000000000000000`.
- **Literal** (`'`*aaaa*`'`): 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 null characters (`'\000'`). 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 in Figure 17-3.

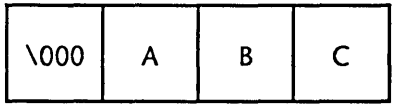


Figure 17-3 Padding of literals

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.	(*expr*)	Parentheses can be used in the normal manner to force precedence in expression calculation
2.	-*expr*	Arithmetic (two's complement) negation of *expr*
	~*expr*	Bitwise (one's complement) negation of *expr*
	! *expr*	Logical negation of *expr*
3.	*expr1* * *expr2*	Multiplication
	expr1 / *expr2*	Division
	expr1 % *expr2*	Remainder from dividing *expr1* by *expr2*
4.	*expr1* + *expr2*	Addition
	expr1 - *expr2*	Subtraction
5.	*expr1* << *expr2*	Shift left—shift *expr1* left by *expr2* bits
	expr1 >> *expr2*	Shift right—shift *expr1* right by *expr2* bits
6.	*expr1* > *expr2*	Greater than
	expr1 >= *expr2*	Greater than or equal to
	expr1 < *expr2*	Less than
	expr1 <= *expr2*	Less than or equal to
7.	*expr1* == *expr2*	Equal
	expr1 != *expr2*	Not equal
8.	*expr1* & *expr2*	Bitwise AND
9.	*expr1* ^ *expr2*	Bitwise XOR
10.	*expr1* | *expr2*	Bitwise OR
11.	*expr1* && *expr2*	Logical AND
12.	*expr1* || *expr2*	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.

To learn more about `#printf`, see section "Print directive" on page 331

`$$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 SADeRez, this returns the null string (`""`). In MPW's Rez and DeRez, 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** (`"`*a*`..."`): The string can contain any printable character except `'"` `'`and `\'`. These and other characters can be created through escape sequences. (See section "Escape characters" on page 337.) The string `""` is a valid string of length 0.

- **Hex string** (`$"`*hh*`..."`): Spaces and tabs inside a hexadecimal string are ignored. There must be an even number of hexadecimal digits. The string `$""` 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.)

Figure 17-4 shows a Pascal string declared as

```
pstring [10];
```

whose data definition is

```
"Hello".
```

\005	H	e	l	l	o	\000	\000	\000	\000	\000

Figure 17-4 Internal representation of a Pascal string

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

```
"Hello \"out\" there."
```

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:

Name	Escape Sequence	Hex Value
Tab	\t	$09
Backspace	\b	$08
Return	\r	$0A
Newline	\n	$0D
Form feed	\f	$0C
Vertical tab	\v	$0B
Rubout	\?	$7F
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	Form (# digits)	Example
2	`\0B`*bbbbbbbb* (8)	`\0B01000001`
8	`\`*ooo* (3)	`\101`
10	`\0D`*ddd* (3)	`\0D065`
16	`\0X`*hh* (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

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 `0x20` and `0xD8` 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 19, "Using SADeRez," for details.

Using SARez 18

SARez is a utility that creates resources from a textual description. It 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.

Contents

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 17, "Resource Description Files."

Running SARez

When you run SARez, you see the dialog in Figure 18-1. 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.

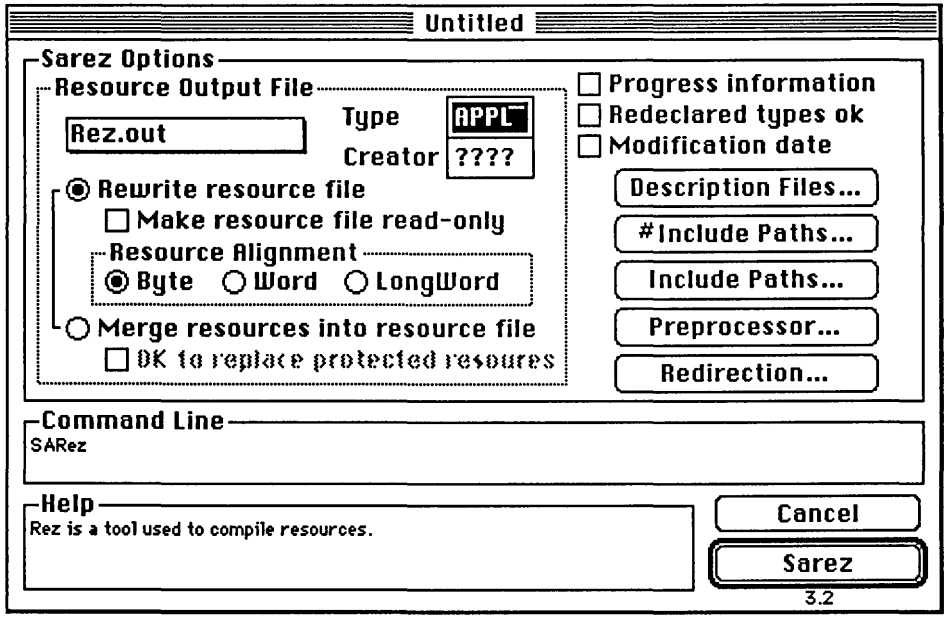


Figure 18-1 The SARez dialog

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 in Figure 18-2 has `Types.r` (in the `{RIncludes}` folder) and `Sample.r` chosen as the description files.

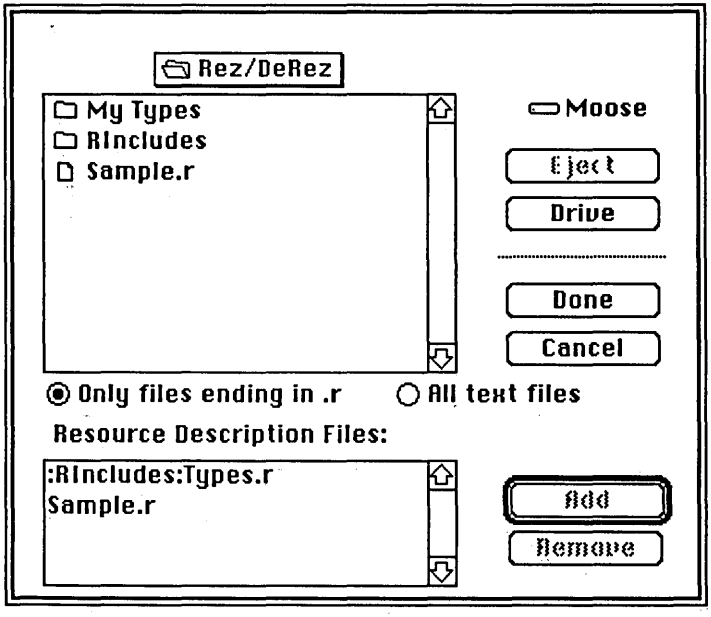


Figure 18-2 The Description Files... dialog

These radio buttons determine which files appear in the top list:

- **Only Files Ending in .r** Such as `Sample.r`.
- **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 low-

er 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 in Figure 18-3. This box is part of the SARez dialog.

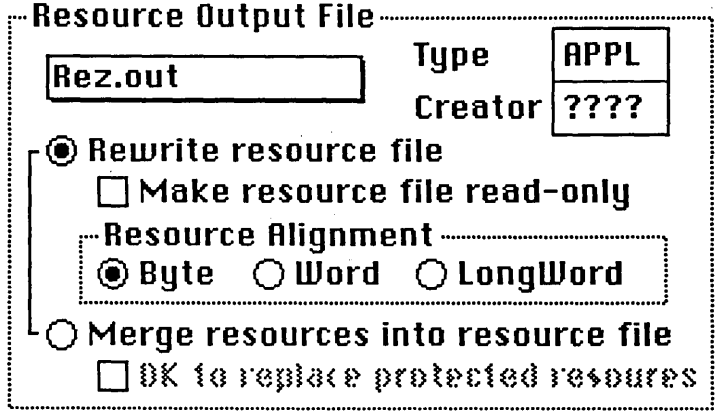


Figure 18-3 The Resource Output File box

To choose the file, click on the button in the upper left hand corner of the box. The pop-up menu in Figure 10-4 appears.

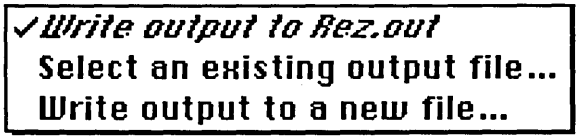


Figure 18-4 Resource Output File menu

This is what the choices mean:

- **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:

- **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:

- **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:

- **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" on page 342 and the Resource Output File box, which is described in the section "Choosing an Output File" on page 343.

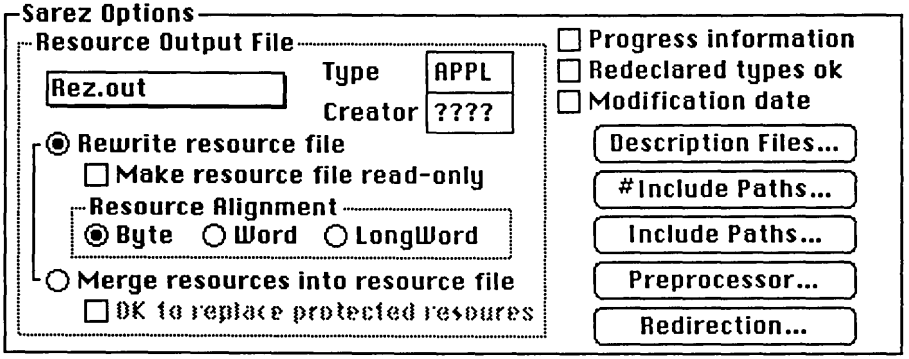


Figure 18-5 The Sarez Options box

The three check boxes in the upper-right hand corner control these options:

- **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 a warning message 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 in Figure 18-6 has the folder `RIncludes` in the list of folders to be searched.The Include Paths... dialog is similar.

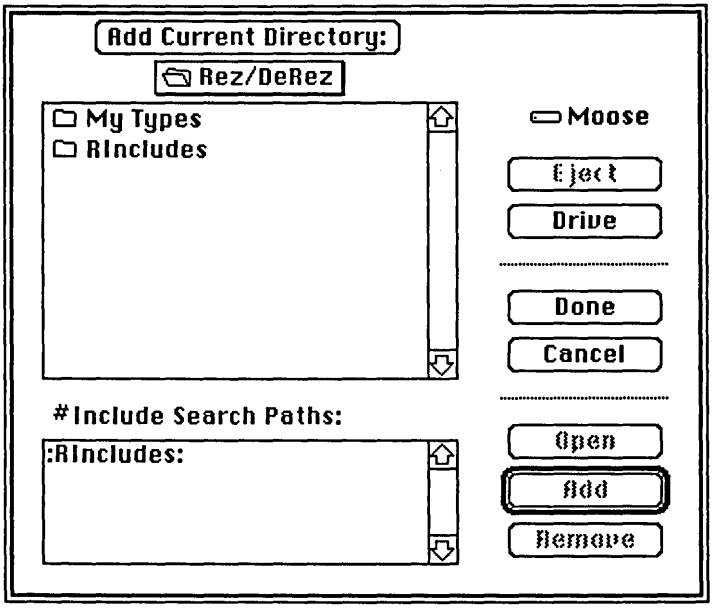


Figure 18-6 The Include Paths... dialog

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 18-6 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 as in Figure 18-7 is like adding these lines to the beginning of each description file:

```
#define LIGHTSPEED 186282
#define THINK_PASCAL
#undef  DESK_ACC
```

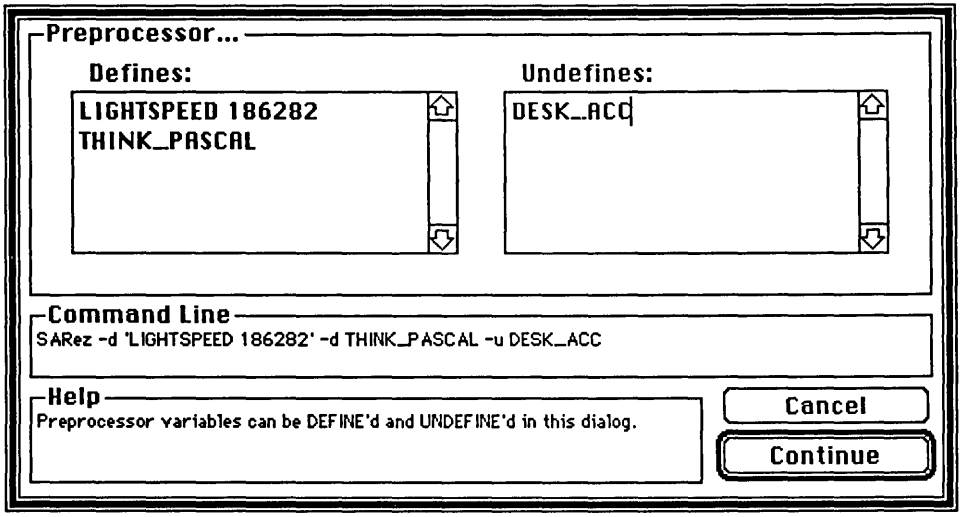


Figure 18-7 The Preprocessor... dialog

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 in Figure 18-8 sets the error file to be `Errors` and sets no alternate input file.

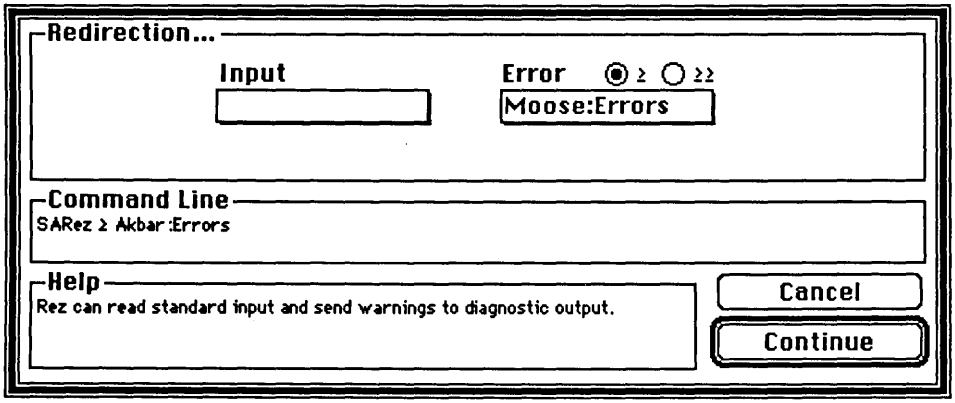


Figure 18-8 The Redirection... dialog

Clicking on the button under Input brings up the pop-up menu in Figure 18-9.

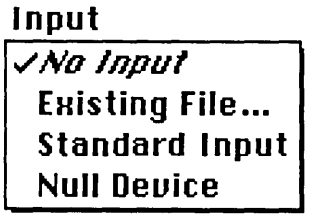


Figure 18-9 The Input menu

This is what the options mean:

- **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, Null Device** These act the same as **No Input**.

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 menu under Error lets you choose an additional place to write error, warning, and status messages. SARez always writes its messages to the messages window, described in section "The Messages Window" on page 350. 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 in Figure 18-10.

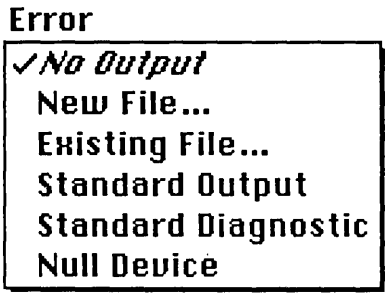


Figure 18-10 The Error menu

This is what the options mean:

- **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 near Error, as in Figure 18-11.

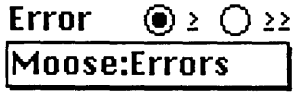


Figure 18-11 The Error menu buttons

You can choose whether or not to overwrite the existing file:

- ≥ 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:

- **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 error, warning, or status messages, it uses the messages window, like the one in Figure 18-12.

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:

- **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.

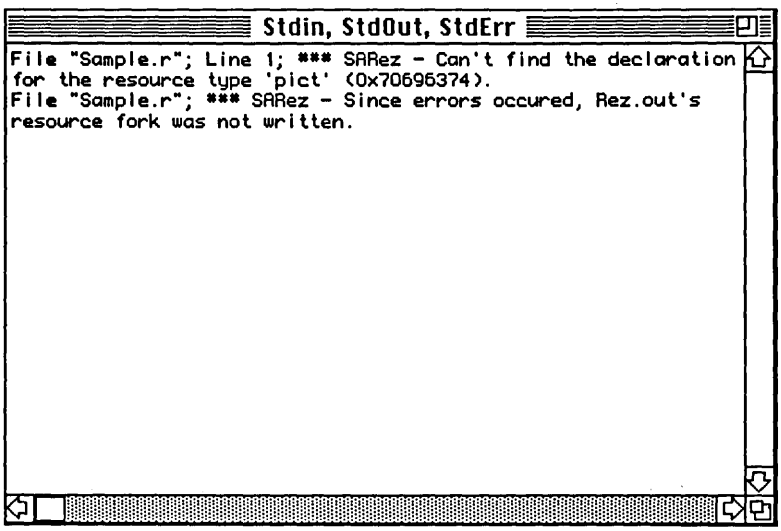


Figure 18-12 The messages window

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 SADeRez 19

SADeRez creates a text representation of the resource fork of a file. It 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 SADeRez works a bit oddly or has options you can't use. This is because of its heritage. But, despite its odd upbringing, SADeRez is still a powerful and useful tool.

Contents

What Is SADeRez?

SADeRez (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.

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 17, "Resource Description Files."

Note

SADeRez 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 SADeRez is used as input to SARez, with the same resource description files, it produces the same resource fork that was originally input to SADeRez. SADeRez 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 grayed 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 in Figure 19-1. To continue, you can change your options and click the button titled "Saderez button." 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 ti-

tled "Cancel" or choose **Quit** from the **File** menu. More information on choosing and saving your options follows.

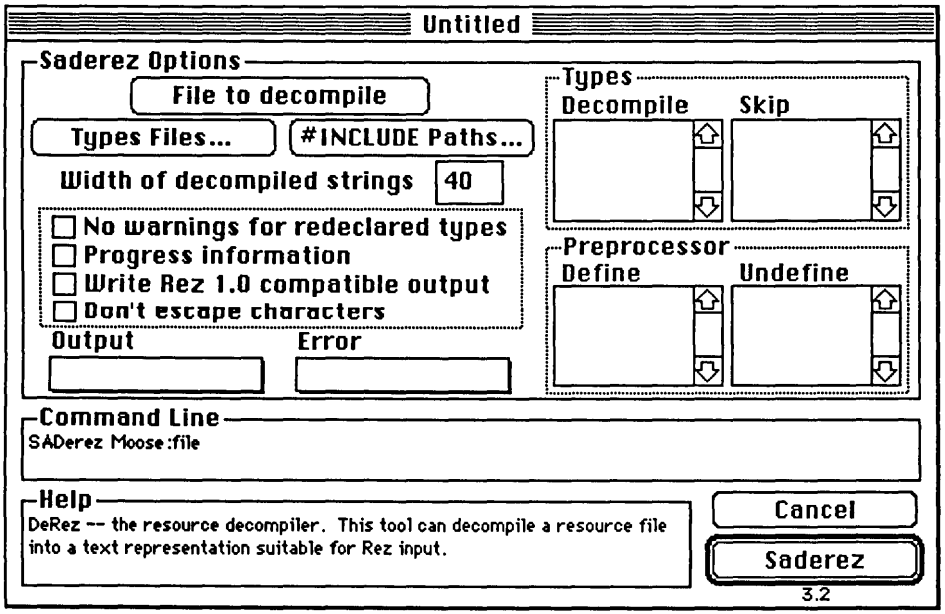


Figure 19-1 The SADeRez dialog

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 SADeRez dialog, as shown in Figure 19-2.

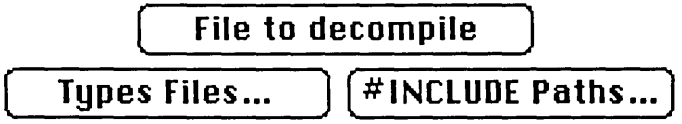


Figure 19-2 The buttons for specifying input files

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 in Figure 19-3 has `Types.r` (in the `{RIncludes}` folder) selected as the description file.

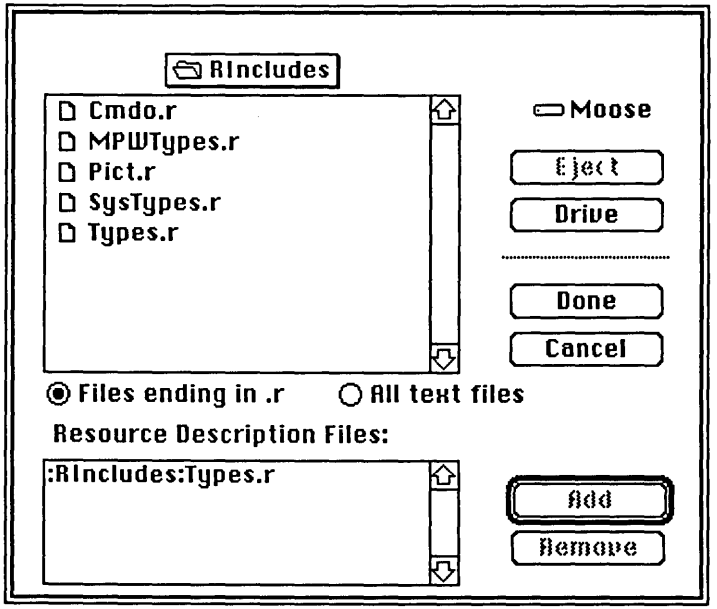


Figure 19-3 The Types Files... dialog

These buttons choose which files appear in the top list:

- **Only Files Ending in .r** Such as `Types.r`
- **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 in Figure 19-4 has the folder `{RIncludes}` in the list of folders to be searched.

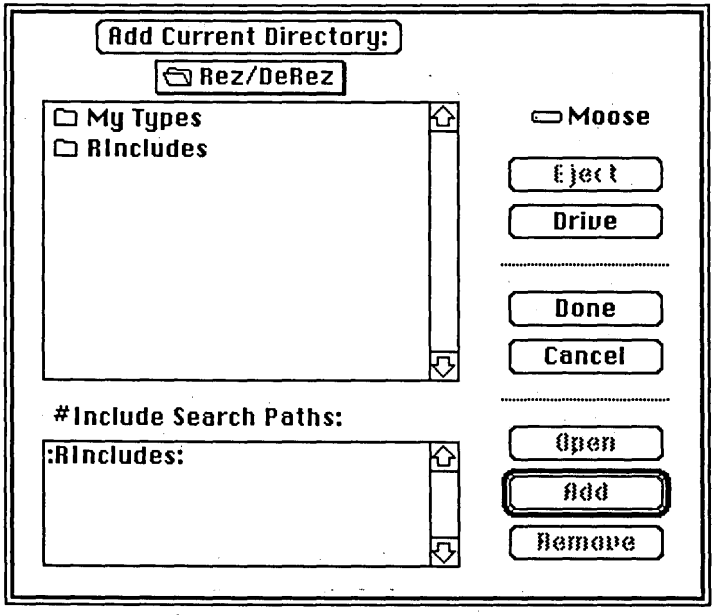


Figure 19-4 The #INCLUDE Paths... dialog

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. It's name is added to the list in the bottom pane.
- To add the folder you're currently in, click Add Current Directory button. For example, clicking Add Current Directory in Figure 19-4 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. SADeRez 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 SADeRez Options box. The buttons in Figure 19-5 set the description file to be the new file `Test.r` and the error file to be the existing file `Errors`.

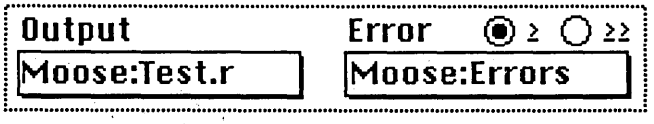


Figure 19-5 The buttons for specifying output files

Clicking on the button under either Output or Error brings up the same pop-up menu, shown in Figure 19-6.

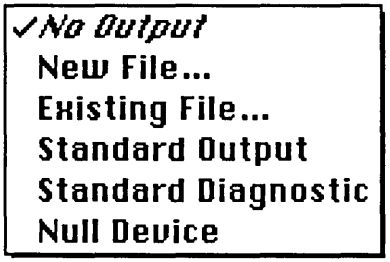


Figure 19-6 The output file menu

This is what the options mean:

- **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 two radio buttons above the menu, as in Figure 19-7 (In the menu under Output, the buttons are labeled ">" and ">>").

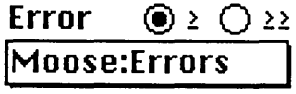


Figure 19-7 The output file menu buttons

These let you choose whether or not to overwrite the existing file:

- ≥ 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 in Figure 19-8 is between the input and output buttons.

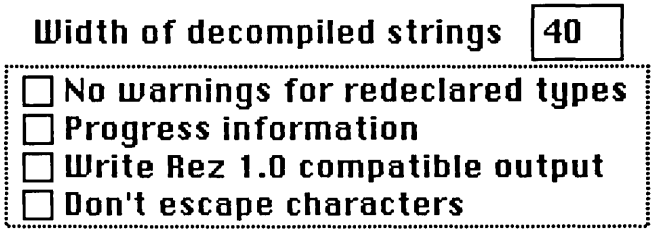


Figure 19-8 Some options

This is what the options mean:

- **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, SADeRez will not print a warning message to the error file when a resource type is redeclared in the description files.
- **Progress information option** If selected, SADeRez writes its version number and information on the decompiled resources to the error file.
- **Write Rez 1.0 compatible output** If selected, SADeRez 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 `0x20` and `0xD8` 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 SADeRez decompiles. This box contains two lists:

- **Decompile box** If you want to decompile only a few resource types, enter their names here. For example, with the box in Fig-

ure 19-9, SADeRez will decompile resources of type `'WIND'` only.

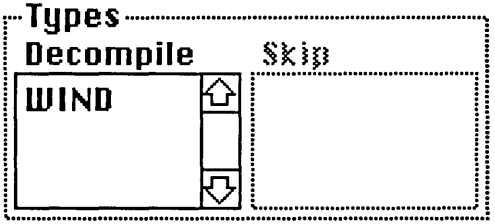


Figure 19-9 Decompile only WIND

- **Skip box** If you want to decompile every resource type with a few exceptions, enter the exceptions here. For example, with the box Figure 19-10, SADeRez will decompile all resources, except those of type `'PICT'`.

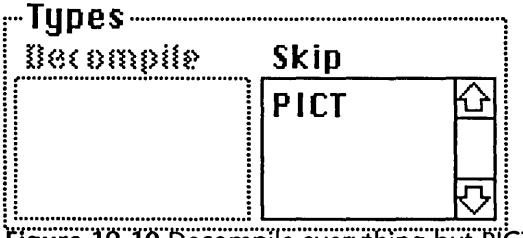


Figure 19-10 Decompile everything but PICTs

You can use only one list at a time. Whenever one has anything in it, the other is grayed out. If you don't use either list, SADeRez 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 in Fig-

ure 19-11 is like adding these lines to the beginning of each resource description file:

```
#define c 186282
#define PASCAL
#undef  DESK_ACC
```

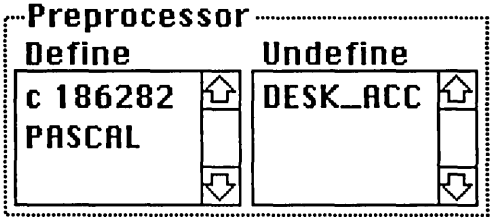


Figure 19-11 The Preprocessor box

Saving and Restoring Options

SADeRez lets you save your option settings in a SADeRez options file. This options file contains the names of your input and output files, in addition to all the other option settings.

When the SADeRez dialog box is in the front, the **File** menu contains these commands to let you save and restore your settings:

- **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 SADeRez options file in the Finder, SADeRez doesn't display its dialog. It just compiles your files, displays your errors (if any), and exits.

The Messages Window

If SADeRez encounters no problems while decompiling your file, 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, like the one in Figure 19-12:

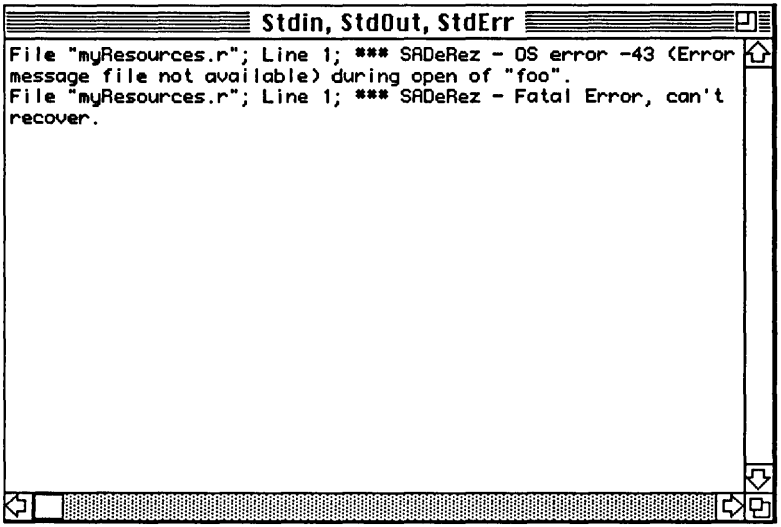


Figure 19-12 The messages window

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:

- **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 SADeRez 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. SA-DeRez will write your messages to both the messages window and the error file.

THINK C
Reference

Part Five

THINK C Menus 20

This chapter describes each of the THINK C 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 in the menu.

Contents

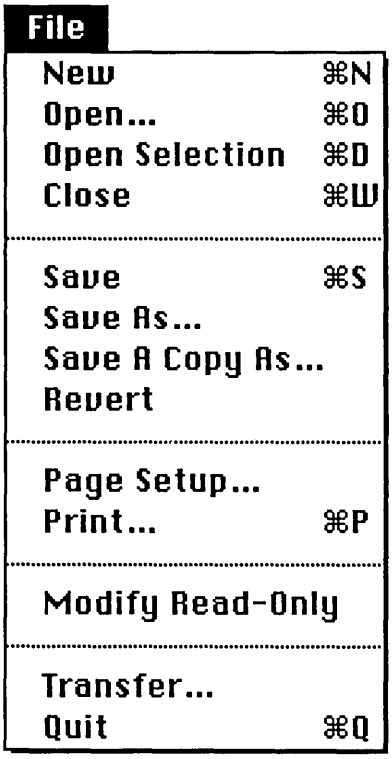
The Apple Menu

About THINK C...

This command tells you what version of THINK C you're using. When you choose this command, you'll see a star field. Click the mouse to see the version of THINK C. Click the mouse two more times to end the display.

The File Menu

Use the File menu commands to work with files that you open and edit with the THINK C text editor. This menu also has the commands that let you launch other applications and that let you quit THINK C.



New

This command opens a new Untitled window. You must save this file with a .c extension if you plan to compile it or add it to the project. However, you can use the **Check Syntax** command on the **Source** menu to compile it without adding it to the project.

Open...

This command displays a dialog box that lets you select from existing files on the disk and open one for editing. When you first begin a project, one way to add a file is to open it with this command, then compile it or add it with the **Add** command. (The **Add...** command lets you add multiple files without compiling or opening the files. Once files have been added to the

project, they can be opened by double-clicking on the file name in the project window.)

You can open multiple files and the edit windows will stack up with the titles showing, so you can easily click on any window to bring it to the front.

If you open too many files at once, you may have to close some windows to free up memory when you compile.

Open Selection

This command lets you open an #included header file simply by selecting its name in the current program text. You don't need to select the .h extension (or full path name for HFS files). The selection will automatically be extended to the right to include it as long as the file name itself is selected.

Close

This command is the same as clicking on the active window's close box. If you try to close an edit window, and the file has been modified since it was last saved, a dialog box will ask you if you want to save the changes, discard them, or cancel the **Close** command. To close the project window, use the **Close Project** command in the **Project** menu.

If the "Confirm Saves" option in the **Options...** dialog box is off, THINK C will save changed files without asking.

Save

This command saves the file in the active edit window to disk. If the file is currently untitled, a dialog box will ask you to name the file.

Note that an updated file can be compiled and added to the project without being saved, as long as it has been saved at least once and given a name with a .c extension.

Save As...

This command lets you save the current file under another name. If you have made edits in the current session, they will be saved under the new name. The original file will remain unchanged, and as you continue editing, you will be editing the new file. This feature is useful for switching to a new version of a file, leaving the old file as a backup.

Save As... tries to preserve the tie between the file you are editing and its entry in the project window. If the file appears in the project window, and the name you want to save it as has a .c extension, and if the new name doesn't already appear in the project window, then the entry for the file in the project window is changed to match the new file name. Use **Save a Copy As...** if you don't want this to happen.

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!

Neither **Save As...** nor **Save a Copy As...** will let you save into a file already open in an edit window.

Revert

This command restores the last-saved version of the current file, and discards any edits made in the current session.

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...

This command lets you print the current file or the Link Errors window. The **Print...** dialog box lets you set the page range among other options. You can also choose the "Print Pages in Reverse Order" option. When you press the OK button, your file will begin to print. Each page of the file has a header showing the name of the file and the last modification date. The thumb of the vertical scroll bar moves from top to bottom (when printing in forward order), and from bottom to top (when printing in reverse order), to show you the printing progress. To cancel printing, press Command-Period.

Modify Read-Only

If you turn on the "Projector-aware" option, this command lets you modify a file that MPW Projector has marked read-only. For more information on using MPW Projector with THINK C, see "Using MPW Projector with THINK C" on page 136.

Transfer...

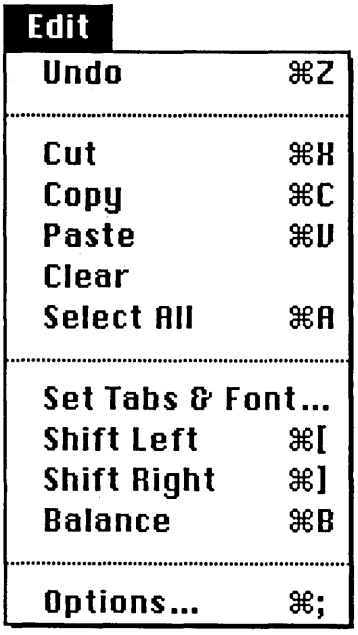
This command lets you launch another application without first returning to the Finder. When you're running under MultiFinder, this command launches applications without quitting THINK C.

Quit

This command exits THINK C and returns to the Finder.

The Edit Menu

*The Edit menu contains the standard Macintosh editing commands (**Cut, Copy, Paste**) as well as other commands that let you format your THINK C source files. The Edit menu also contains the **Options...** command that lets you set several THINK C options so the environment suits your personal needs.*



Undo

The **Undo** command reverses the last edit operation. The actual name of this command changes to let you know exactly what 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 will be dimmed. If the operation to undo doesn't belong to the frontmost window, the name of the command will indicate that there is something to do, but the command will be dimmed.

You can't undo a **Replace All**, **Revert**, or a **Set Tabs & Font...** command.

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 now 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.

Set Tabs & Font... This command lets you change the tab stops and the font used by the THINK C editor. You can select different tab stops and fonts for each edit window. When you choose this command you'll see a dialog box like the one in Figure 20-1.

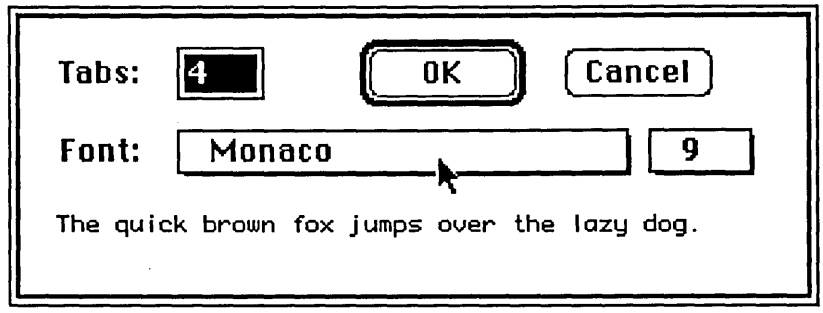


Figure 20-1 The **Set Tabs & Font...** dialog

Type a number to set the number of spaces per tab. To change the font, click on the font name. You'll see a pop up menu with the names of the fonts in your System file. Click on the font size to see a pop up menu of the sizes for the font.

If you are using a proportionally spaced font like New York or Geneva, the THINK C editor uses the width of the letter m to figure out the width of a tab.

When you change the font or tab settings, the editor adds `EFNT` and `ETAB` resources to your text files to record the new settings. Other text editors use these resources as well.

Shift Left This command shifts a selected range of lines to the left. It deletes the first character of each line in the selected range, as long as the line begins with a tab.

Shift Right This command shifts a selected range of lines to the right. It inserts a tab at the start of each line in the selected range.

Balance

This command extends the current selection in both directions until it encloses the smallest surrounding balanced text enclosed in parentheses `()`, brackets `[]`, or braces `{}`. Successive invocations select larger sequences of text.

This is a quick way to check whether all your function definitions are properly balanced. Start at the beginning of a file and search for the first left brace ("`{`"). Then use **Balance** and **Find Again** commands repeatedly until you get to the end of the file.

Balance is a textual operation. It doesn't know about comments or strings, so if you have a lone brace, bracket, or parentheses in a comment, it will try to find a match for it.

Options...

This command opens a dialog box that lets you set six groups of options:

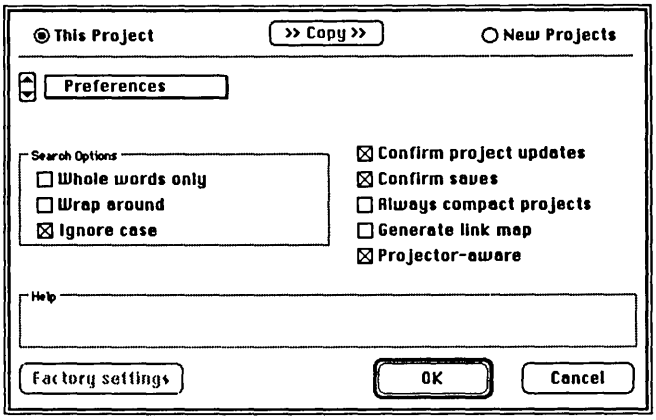
- Preferences, on page 377, lets you specify how THINK C behaves when it rebuilds projects, and opens and closes files.
- Language Settings, on page 379, lets you choose which extensions to the C language you can use.
- Compiler Settings, on page 382, lets you control how THINK C generates code.
- Code Optimization, on page 387, lets you control how THINK C optimizes your code.
- Debugging, on page 389, lets you specify how the THINK C debugger works.
- Prefix, on page 391, lets you write preprocessor code that THINK C will include in all your files.

You can set the options for the current project, or you can set the defaults that THINK C will use when you create a new project. Use the Copy button at the top of the dialog box to copy the THINK C defaults to the current project, or, if you want the options you've set for a particular project to be the THINK C options. Note that even though only one page of the options shows up in the dialog at one time, the Copy button copies *all* of the options, even the ones in the pages you can't see.

When you click on the Factory Settings button, THINK C sets all the options in all the pages back to their factory settings, the settings they had when you first took THINK C out of the box. The factory settings are included in the options' descriptions.

When you click on the OK button, the changes for all pages of the dialog are saved.

PREFERENCES



The Preferences page of the ***Options...*** *dialog lets you set the defaults used in the* ***Find...*** *dialog, specify how THINK C behaves when it rebuilds projects, closes files, closes projects, and how it deals with memory. (When you set this options in the* ***Find...*** *dialog, they apply only for the current session.) The* ***Find...*** *command, is described on page 392.*

Whole words only

When this option is set, the editor's search and replace functions will work on whole words only. For example, a search for "string" will not match the word "strings." The factory setting is off.

When you set this option in the **Find...** dialog, it applies only for the current session. The **Find...** command is described on page 392.

Wrap around

When this option is set, the editor's search and replace functions will search the entire file, rather than from the current position to the end of the file. When the end of the file is reached, the search "wraps around" to the beginning of the file and continues. The factory setting is off.

Ignore case

When this option is set, the editor's search and replace functions will disregard case when performing a search. A search string will match either upper or lower case. The factory setting is on.

Confirm project updates

When this option is off, THINK C does not display the "Bring project up to date?" dialog when you choose the **Run** or any of the **Build...** commands. THINK C assumes you would have answered Yes. The factory setting is on.

Confirm saves

When this option is off, THINK C automatically saves changes to a file that you have modified without asking if you are sure you want to do so. The factory setting is on.

This is a dangerous option to turn off, since it protects you from inadvertently replacing previous versions of your files with newly modified versions. However, it is very convenient when you want to do program development in quick, incremental steps.

Always compact projects

If you turn this option on, the project document will be compacted when you **Close** the project, **Transfer...**, or **Quit** (but not when you **Run**). The factory setting is off.

To achieve its remarkable speed, THINK C pre-allocates space in the project for anticipated requirements, and does not necessarily free up the space when an item is deleted. As much as 20% of an uncompacted project document can contain unused space.

Compaction may be time-consuming, so you will normally want this option on only when disk space is at a premium. The amount of space freed will vary. To compact a project without setting this option, use the **Close & Compact Project** command in the **Project** menu.

Generate link map

The format of a link map is described in "Generating a link map" on page 168

If this option is on, THINK C writes a link map for your application when you build it. The factory setting is off.

The link map lists all your project's segments, including one for global data. For each function (or global variable) in a segment, the map lists its name, its position in the segment, and the file it's defined in.

THINK C creates the link map only when you use the **Build Application...** command. The name of the map is the name of the project with `.map` appended. For example, the link map for the `Bullseye.π` project is `Bullseye.π.map`. THINK C places the link map in the project folder and erases any link map that was there.

Projector-aware

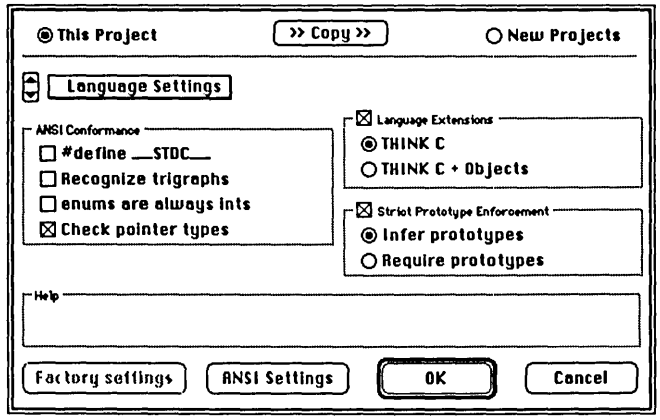
For more information, see "Using MPW Projector with THINK C" on page 136.

When this option is on, THINK C checks for a Projector resource (`CKID` resource) every time it opens a source file. THINK C displays an icon in the lower left corner of the editing window that tells you whether Projector marked the file as read-only. If the file is a read-only file, its window will

have a crossed-out pencil icon. If it's not a read-only file, its window will have a regular icon. If it's a modified read-only file, its window will have a pencil icon with a dotted cross-out.

LANGUAGE SETTINGS

The Languages Settings page lets you choose whether THINK C compiler uses some extensions to the C language.



All the options in this page control whether THINK C is ANSI-conformant. This is the only page that contains the ANSI Settings button. When you click on that button, the options in this page are set up so that THINK C is ANSI-conformant. The button is grayed out if THINK C is already ANSI-conformant.

#define __STDC__

If this option is on, THINK C defines the macro `__STDC__`. If it's off, THINK C doesn't define `__STDC__` at all. The factory setting is off. The ANSI-conformant setting is on.

If THINK C is ANSI-conformant, `__STDC__` is defined to be 1. If THINK C isn't ANSI-conformant, `__STDC__` is defined to be 0.

Recognize trigraphs

If this option is on, you can use trigraph sequences in your code The factory setting is off. The ANSI-conformant setting is on.

Turn this option on only if you're writing strictly conforming ANSI C programs. Many Macintosh programs use character strings that resemble trigraphs. For example, this file type:

```
'????'
```

is interpreted as

```
'?^
```

THINK C replaces the trigraph `??'` with `^` and complains that the character literal lacks a closing apostrophe (`'`). To write that file type with the "Recognize Trigraphs" option on, use this:

```
'???\?'
```

enums are always ints

When this option is on, all enumeration constants are the same size as an `int`. When this option is off, enumeration constants can be the same size as a `char`, `short int`, `int`, or `long int`. The factory setting is off. The ANSI-conformant setting is on.

If this option is off, THINK C makes enumeration constants as small as possible. And, if necessary, it will make a constant as large as a `long int`. For example, these constants will only be as large as a `char`:

```
enum { red=1, yellow, green };
```

And these constants will be as large as a `long int`:

```
enum { million=1000000, billion=1000000000 };
```

If this option is on, THINK C makes all enumeration constants the size of an `int`. You cannot use constants as large as a `long int`.

If you're writing ANSI-conformant code, turn this option on. Otherwise, you'll want to leave it off.

Check pointer types

For more information, see "Checking pointer types" on page 188.

When this option is on, THINK C makes sure that pointer types match when you assign one pointer to another or when you do pointer arithmetic. If this option is off, THINK C treats all pointers as equivalent types, and won't display the "pointer types do not match" error message. When subtracting two pointers, however, the two types must be pointers to objects of the same size.

Language Extensions

This option lets you use THINK C's extensions to the C language. You can choose between two sets of extensions: "THINK C" or "THINK C + Objects." If you don't select this option, you can use only ANSI C with no extensions. To be ANSI-conformant, turn this option off.

This table explains the two sets of extensions:

If you choose...	You can use...
THINK C	THINK C extensions, but no object extensions.
THINK C + Objects	THINK C and object extensions.

For a list of the THINK C extensions, see "THINK C Extensions" on page 174.

The THINK C extensions to the C language make programming on the Macintosh easier. They include inline assembly, the `pascal` keyword, and // comments. The object extensions let you write object-oriented code in C. They're described in *Object-Oriented Programming*, Chapter 4, "Using Objects in THINK C."

If you use the object extensions, these words become keywords, and you can't use them as identifiers:

```
class           delete
new             virtual
```

This list explains which option is best for your program:

- If your program must be 100% ANSI-compatible, choose turn off "Language Extensions." Make sure the rest of the options in this page are set to their ANSI-compatible settings.
- If your program doesn't use objects, choose "THINK C."
- If your program uses objects or the THINK Class Library, choose "THINK C + Objects."

Strict Prototype Enforcement

To relax prototype checking when you precompile a header file, turn off the "Check pointer types" option, described on page 380.

This option lets you choose how strictly THINK C enforces the use of prototypes. If this option is on, you can choose between two enforcement levels: "Infer prototypes" and "Require prototypes." If this option is off, THINK C does nothing when you use a function without a prototype. The factory setting is "Infer prototypes." The ANSI-conformant setting is "Infer prototypes."

This table explains the two enforcement levels:

If you choose...	THINK C does this when you use a function without a prototype...
Infer prototypes	Infers a prototype from the first appearance of a function. That appearance can be a function call or an old-style declaration. If a subsequent call, declaration, or

For details, see "Enforcing prototype use" on page 189.

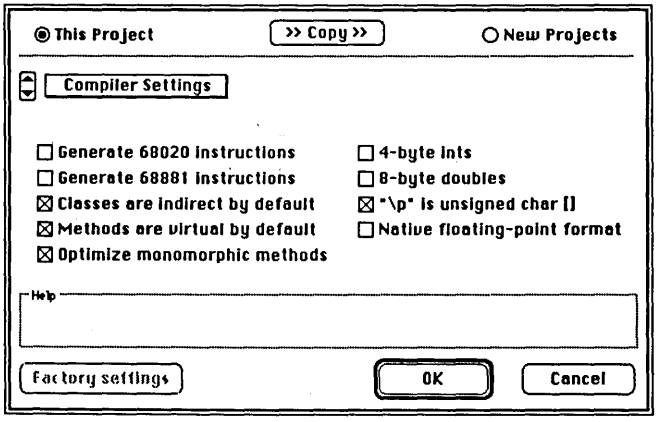
	prototype doesn't match the inferred prototype, THINK C raises an error.
Require prototypes	Raises an error. You can't use or define a function unless it has a prototype. New-style function definitions do *not* count as prototypes.

Note

There is one exception to the "Require prototypes" requirement. A static function does not need a prototype. Just define it with a new-style function definition before using it.

COMPILER SETTINGS

The Compiler Settings page lets you control how THINK C compiles your code.



Generate 68020 instructions

If this option is on, THINK C generates code that is optimized for Macintoshes with the MC68020, MC68030, or MC68040 (like the Macintosh LC, II, or IIfx), and that will not run on Macintoshes with the MC68000 (like the Macintosh Classic or Plus). If it's off, THINK C generates code that will run on all Macintoshes. The factory setting is off.

If the option is on, THINK C generates MC68020 instructions for bitfield operations, addressing, and long word multiplication, division, and modulo operations. Also, the inline assembler accepts all MC68020 instructions and addressing modes for assembly in an `asm { ... }` construct.

Note

Before your program uses MC68020 instructions, make there is a MC68020, MC68030, or MC68040 or your program may crash. Use the Gestalt Manager, described in *Inside Macintosh VI*, Chapter 3, "Compatibility Guidelines."

Generate 68881 instructions

If this option is on, THINK C generates code that is optimized for Macintoshes with the MC68881 or MC68882 floating-point unit (like the Macintosh II or IIfx) or the MC68040, and that will not run on Macintoshes without it (like the Macintosh Classic, LC, or Plus). If it's off, THINK C generates code that will run on all Macintoshes. The factory setting is off.

If this option is on, THINK C generates code for the floating point coprocessor. You may declare up to five local floating-point variables as `register` variables, and THINK C will placed them into MC68881 registers.

For more information on the "Native floating-point format" option, see page 385.

If both this option and the "Native floating-point format" option are on, long doubles use the MC68881's 96-bit format. Since SANE expects 80 bit doubles, you'll need to convert from one format to the other. The `float` (32 bits) and `short double` (64 bits) types are identical for SANE and for the MC68881.

Note

Before your program uses MC68881 instructions, make sure there is a MC68881, MC68882, or MC68040, or your program may crash. Use the Gestalt Manager, described in *Inside Macintosh VI*, Chapter 3, "Compatibility Guidelines."

Classes are indirect by default

Indirect classes are implemented as handles. Direct classes are implemented as pointers.

If this option is on, classes declared with the `class` keyword are indirect by default. If this option is off, they are direct by default. This option is available only in the "THINK C + Objects" option is on. The factory setting is on.

This option doesn't affect classes declared with `struct`. A root class declared with `struct` must use `direct` or `indirect`.

No matter how this option is set, you can still use the words `direct` and `indirect` to declare explicitly how to allocate a class.

For more information on whether to declare classes direct or indirect, see *Object-Oriented Programming*, Chapter 4, "Using Objects in THINK C."

Methods are virtual by default

If this option is on, all methods are virtual, even those that aren't declared `virtual`. If this option is off, only methods declared `virtual` are virtual. This option is available only in the "THINK C + Objects" option is on. The factory setting is on.

This option doesn't affect all methods. Some are always virtual; others are never virtual:

- Static methods, including allocators and deallocators, are *never* virtual.
- Constructors are *never* virtual.
- Destructors are *always* virtual

If you're using code you wrote with the previous version of THINK C, either turn this option on or declare your methods `virtual`. The previous version didn't use the `virtual` keyword and made all methods virtual.

If you're writing new code, turn this option off and declare your methods `virtual` as necessary. Your code will be more compatible with C++.

Optimize monomorphic methods

A monomorphic method is a method that does not override another method and is never overridden itself.

If this option is on, THINK C optimizes monomorphic methods by generating a call directly to the method. If this option is off, THINK C generates a call to a run-time dispatcher.

You'll rarely need to turn this option off. You'll keep it on unless you're working with a program that has replaceable code modules.

4-byte ints

When this option is on, integers are the same size as long integers, four-bytes long. When it's off, they're the same size as short integers, two-bytes long. The factory setting is off.

Use four-byte integers if you're porting code from a compiler that uses four-byte integers, like MPW C, or you're porting code that assumes integers and pointers are the same size. Otherwise, use two-byte integers, since the MC68000 family handles them more efficiently.

Note

No matter how this option is set, long integers are always four-bytes long and short integers are always two-bytes long.

8-byte doubles

When this option is on, doubles are the same size as short doubles, eight-bytes long. When it's off, they're the same size as long doubles. The factory setting is off.

Use eight-byte doubles if you're porting code from a compiler that uses eight-byte doubles, like MPW C. Otherwise, leave the option off, and THINK C uses the most efficient size possible.

When this option is off, the size of doubles is controlled by the "Generate 68881 instructions" option, and the "Native floating-point format" option. For more information, see page 385.

Note

If both the "8-byte doubles" option is on and the "Language Extensions" option is set to "THINK C," no type in THINK C gives you eight-byte floating-point numbers.

"\p" is unsigned char []

When this option is on, Pascal string literals are of type `unsigned char[]`. Otherwise, they're of type `char[]`. The factory setting is on.

Pascal string literals are string literals that start with `\p`, like this:

```
"\pUntitled"
```

If this option is on, Pascal string literals are compatible with the Macintosh Toolbox types `Str255` and `StringPtr`. When you assign a Pascal string literal to a variable of type `Str254` or `StringPtr`, you don't need to cast the string.

Previous versions of THINK C assumed Pascal string literals to be of type `char[]`. Turn this option off only if you have code that took advantage of this assumption.

Native floating-point format

When this option is off, THINK C uses the universal floating-point format for long doubles. When it's on, THINK C uses either the native SANE or MC68881 floating-point format. The factory setting of this option is off.

Note

Usually, doubles use the same format as long doubles. However, doubles use the same format as short doubles if the "8-byte doubles" option, on page 385, is on.

The universal format is a floating-point format you can use no matter what the setting of the "Generate 68881 instructions option" is. THINK C automatically convert numbers to and from the universal format. The conversions are too quick to notice unless your applications are computation intensive. For more information on these different formats, see "Floating point representation" on page 172.

If this option is on, THINK C uses a native floating-point format for long doubles. Which native format it uses depends on the setting of the "Generate 68881 instructions" option. This chart shows which format it uses:

If Native Floating-point is	**and Generate 68881 is**	**Then long doubles use this format...**
Off	Off	Universal
Off	On	Universal
On	Off	Native SANE
On	On	Native MC68881

If your project contains any libraries or projects that were compiled in the universal format, turn this option off. These THINK C libraries use the universal format:

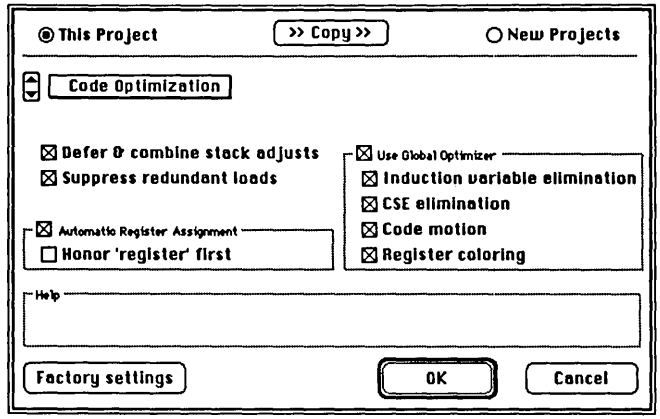
- `ANSI`
- `ANSI-small`
- `ANSI-A4`
- `profile.`

If you're writing a library for general use, turn off both the "Native floating-point format option" and the "Generate 68881 instructions" option. Projects can use your library no matter how they set the "Generate 68881 instructions" option, as long as they turn off the "Native floating-point format" option.

If you want the fastest code possible and use no libraries compiled in the universal format, turn the "Native floating-point format option" on. The conversions to and from the universal format are eliminated.

CODE OPTIMIZATION

The Code Optimization page lets you control how THINK C optimizes your code



The options in this page fall into two categories. The options in the left column control optimizations that the compiler performs. Using them doesn't increase your compile time much.

The options in the right column control the global optimizer. The global optimizer adds an additional pass over your compiled code to improve it. It may double the amount of time it takes to compile your code.

Defer & combine stack adjusts

For more information on this option, see "Defer & combine stack adjusts" on page 186.

If this option is on, THINK C saves time and space in code that calls many functions in a row. The factory setting is on.

If this option is off, THINK C follows this function-calling rule: When a statement calls a C function, that statement pushes the arguments on the stack before calling the function and pops them off after it returns.

If this option is on, THINK C changes that rule. If your code contains a run of C functions, it doesn't pop a function's arguments off the stack immediately after one of the function returns. It continues to call the other functions and push their arguments on the stack. It lets the arguments accumulate and keeps track of how much is on the stack. When the run of functions is done, it pops the arguments off all at once.

Suppress redundant loads

For more information on this option, see "Suppress redundant loads" on page 187.

If this option is on, THINK C doesn't load data into a register when that data is already in a register. This optimization makes your code smaller and faster.

Automatic Register Assignment

For more information on this option, see "Register coloring" on page 186.

If this option is on, THINK C automatically assigns variables to registers to make your code as small as possible. With the option below, you can choose whether THINK C also puts the variables declared `register` into registers. If this option is off, THINK C puts only those variables declared `register` into registers.

If you select this option, the "Honor 'register' first" option below appears. This table explains what that option does:

If "Honor 'register' first is"...	THINK C...
On	Puts the variables declared `register` in registers and puts the variables it wants in the remaining registers.
Off	Ignores your `register` declarations and puts the variables it wants in registers

Use Global Optimizer

This option controls the global optimizer. It is a "master switch." If it's off, the options below it are disabled. If it's on, the options below it are enabled, set to the values they had the last time the master switch was on. The next four sections describe those options.

You probably won't use the global optimizer while you're debugging. It adds an additional pass over your compiled code and may double your compilation time. Also, it generates machine code that is significantly different from your source code. The debugger may not be able to pick out the machine code instructions that correspond to a statement.

Induction variable elimination

For more information on this option, see "Induction variable elimination" on page 185.

This optimization makes loops faster, especially those that cycle through an array. It will make your code slightly larger. Use this optimization if your code contains a lot of loops. It's especially useful on a Macintosh with a MC68000, like a Macintosh Classic or Plus, which performs 32-bit multiplication in software.

CSE elimination

For more information on this option, see "CSE elimination" on page 185.

This optimization makes your code smaller and faster. It replaces subexpressions that are used more than once with a temporary variable set to the subexpression's value. Use this optimization on all your code. It's especially useful if you also use the "Register coloring" option on page 389.

Code motion

For more information on this option, see "Code motion" on page 185.

This optimization makes your loops faster but may make your code slightly larger. It moves expressions out of a loop that remain constant in each iteration. Use this optimization if your code has a lot of loops.

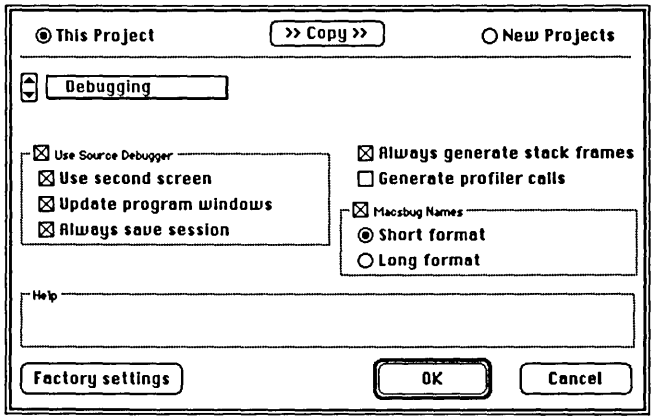
Register coloring

For more information on this option, see "Register coloring" on page 186.

This optimization makes your code smaller and faster. It searches your code for variables of the same type that are never used at the same time and treats them as if they were the same variable. If there's a register free, it lets them share that register.

DEBUGGING

The Debugging page lets you specify how the THINK C debugger works.



Use Source Debugger

When this option is checked, THINK C launches the source level debugger when you run your project. It is a "master switch." If it's off, the options below it are disabled. If it's on, the options below it are enabled, set to the values they had the last time the master switch was on. The next three sections describe those options.

Checking this option is the same choosing the **Use Debugger** command in the **Project** menu.

Use second screen

When this options is on, and you're running on a Macintosh with more than one screen, THINK C will display the source debugger windows in the second screen.

Update program windows

When this option is on, the debugger tries to update your windows for you when your project is stopped. The factory setting is off.

Without this option checked, your program must wait until control comes back so it can handle updates itself. Since it cannot do so until it gets back to its event loop, an update may remain pending for some time.

This option is especially useful when you're trying to step through code that draws in a window. This option uses memory to save your window's image. If there isn't enough memory, the debugger won't be able to perform automatic updates. If you have a large or color screen, you might want to increase the debugger's partition size. For details, see "Update program windows" on page 243.

Always save session

When this option is on, the debugger always saves your breakpoints and the contents of your data windows when you exit the debugger. When this option is off, you must choose **Save** from the debugger's **File** menu to save your session.

Always generate stack frames

For more information on this option, see "Always generate stack frames" on page 244.

When this option is checked, THINK C generates a stack frame for almost every function called. When this option is off, THINK C doesn't generate a stack frame for functions that don't have local variables or parameters. The factory setting is off.

If you're using the debugger's call chain menu or the profiler, turn this option. Otherwise, leave it off. You're program will be smaller and faster without the unnecessary stack frames.

Generate profiler calls

When this option is on, THINK C generates calls to code profiler routines. The code profiler collects timing statistics about your functions. The factory setting is off.

THINK C can profile only for functions that have stack frames. To create stack frames for almost every function, turn on the "Always generate stack frames" option on page 390.

See Chapter 15, "The Profiler," to learn more about the code profiler.

Macsbug Names

This option lets you choose whether THINK C includes function names in your compiled code for assembly language level debuggers such as Macsbug or TMON. If this option is on, you can choose between two types of names "Short format" and "Long Format." If this option is off, THINK C doesn't include function names in your compiled code.

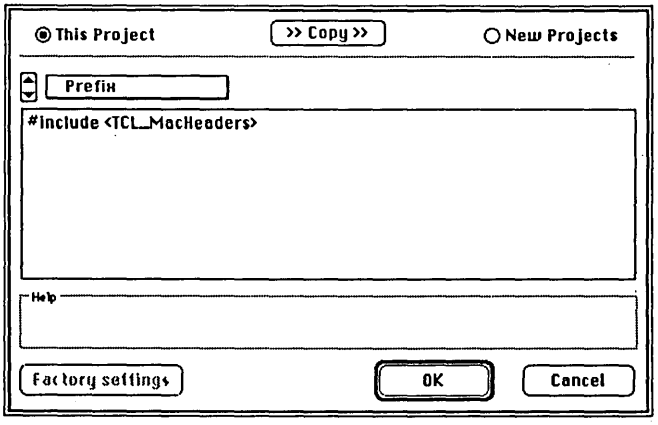
THINK C generates symbols only for functions that have stack frames. To create stack frames for almost every function, turn on the "Always generate stack frames" option on page 390. Be aware that while Macsbug symbols are useful for debugging, they add at least eight bytes to every procedure.

If you use "Short format" names, function names are limited to eight characters. Underscores are stripped off. Names shorter than eight characters are padded with spaces. For methods (like `TDrawWindow::FileWrite`), only the first eight characters of the class and method are stored (like, `TDrawWin::FileWrit`). The THINK C debugger, TMON, and all versions of Macsbug can use these names.

If you use "Long format" names, function names can be any length. For methods (like `TDrawWindow::FileWrite`), the full class and method names are stored (like `TDrawWindow::FileWrite`). The THINK C debugger and Macsbug 6.0 and later can use these names.

PREFIX

The Prefix page lets you write preprocessor code that THINK C will include in all your files.



This page lets you include some text in all the source files in your project. It's as if you put the code from this page at the beginning of all your source files.

If you use a precompiled header file, like `MacHeaders`, #include it here. By default, this page contains the line `#include <MacHeaders>`.

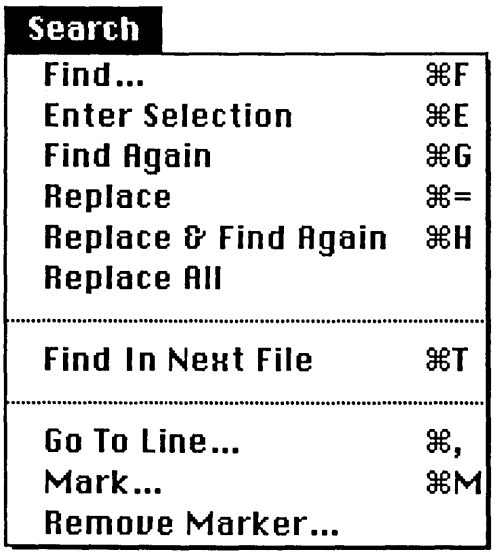
If you need to define a macro in all your files, define it here. For example, you may have some debugging code in your files that's compiled only if the macro DEBUG is defined. To include that code, include this line here:

```
#define DEBUG
```

When you don't need to include the debugging code anymore, just delete that line from this page. You don't need to edit every file in your project.

The Search Menu

The commands in the Search menu let you find and replace strings in your source files. THINK C has extensive search and replace functions including multi-file searching and pattern matching searching. To learn how to search for patterns, see "Searching for a Pattern (Grep)" on page 144.



Find...

This command lets you specify a string to search for. If the string is found, it is highlighted. If it is not found, the editor simply beeps.

At the start of an editing session, only the **Find...** command is active. The dialog box that appears in response to this command lets you specify a string to search for, as well as an optional replacement string.

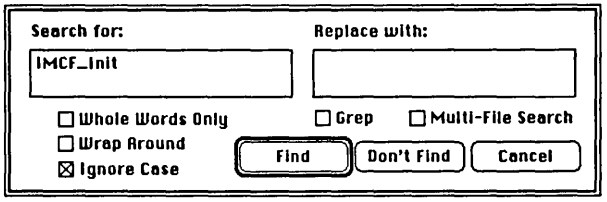


Figure 20-2 The Find... dialog

You can also set search options in this dialog box. Note that you can also set these options in the Search Options page of the **Options...** dialog. When you set the options in the **Find...** dialog, though, they apply only to the current session. If you want to set the defaults permanently for the project you're working on or for new projects you create, use the **Options...** dialog.

Whole words only — When this option is set, the editor's search and replace functions will work on whole words only. For example, a search for "string" will not match the word "strings". This option is off by default.

Wrap Around — When this option is set, the editor's search and replace functions will search the entire file, rather than from the current position to the end of the file. When the end of the file is reached, the search "wraps around" to the beginning of the file and continues. This option is off by default.

Ignore Case — When this option is set, the editor's search and replace functions will disregard case when performing a search. A search string will match either upper or lower case. This option is set by default.

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 in the dialog box. Pressing this button causes the editor to accept the new string and option settings but doesn't initiate a search. This is useful for setting values for a replace operation without executing the first Find.

Enter Selection

This command sets the search string to the current selection, clearing Grep and Multi-File Search. You can then use **Find Again** to begin searching, or **Find...** to set search options.

Find Again

This command searches for the next occurrence of a previously specified string.

Replace

This command replaces the current selection with a replacement string. If you haven't provided a replacement string, this command will clear the string that has been found (that is, it will replace it with nothing). Usually, you use this command after finding a string.

Replace & 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 will 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 & 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 clears the string that has been found (that is, it will replace it with nothing).

Replace All

This command replaces every instance of the search string. If the Wrap Around option is on, it replaces every instance in the file. If the Wrap Around option is off, it replaces every instance 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. If you haven't provided a replacement string, this command will clear the string that has been found (that is, it will replace it with nothing).

Find in Next File

This command lets you search for a string through more than one file.

To use this command, you must check the Multi-File Search check box in the **Find...** dialog box. When you check this box, another dialog box displays all of the text files known to THINK C. You can scroll through the list and select individual files by clicking on them to place a check mark by the name, or you can use the buttons in the dialog box to Check All, Check None, Check All .c or Check All .h. (If a file is already selected, clicking on its name will remove the check mark.)

When you've checked the files you want to include in the multi-file search, click OK to return to the **Find...** dialog box.

THINK C will search for the string specified in the **Find...** dialog box through each of the files that have been checked, starting with the first one. If the search string is found in a given file, THINK C opens an edit window containing the file, and the search string is selected. 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...**, **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.

Multi-file search is useful when you are writing a program, and decide to modify a function that is used in multiple 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.)

Go To Line...

This command lets you move to a specific line in your file. You need to know the line's number. Lines are numbered consecutively, with the first line in the file being number 1.

When you choose **Go To Line...**, you'll see the dialog in Figure 20-3.

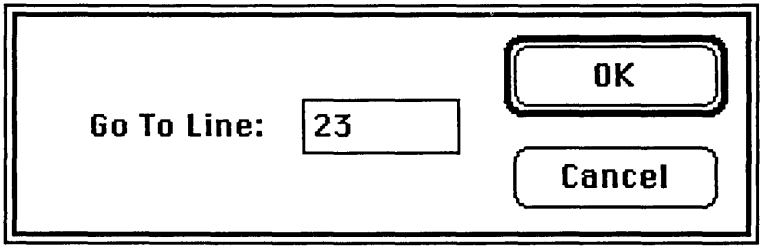


Figure 20-3 The Go To Line... dialog

The default line is the line that the insertion point is in. Edit it and click OK. If you change your mind, click Cancel. The editor moves the insertion point to the beginning of the line you entered.

Mark...

This command lets you place a marker in a file. Place the insertion point in the line you want to mark or select some text in that line. Choose the **Mark...** command in the **Search** menu. You'll see the dialog in Figure 20-4.

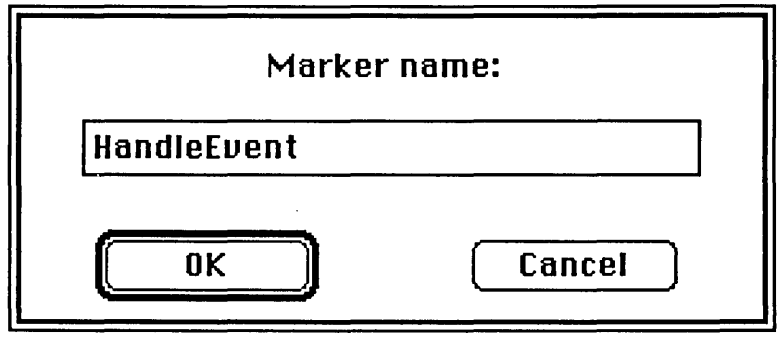


Figure 20-4 The Mark... dialog

Edit the default name if you want, and click OK. If you selected text, the editor uses that text as the default name. If you don't select text, the editor uses the word nearest the insertion point as the default name. If you don't want to insert a marker after all, click Cancel.

To go to a marker in a file, hold down the Command key and click in the title bar of the file's editing window. A pop-up menu of the file's markers appears, sorted alphabetically.

Remove Marker...

This command lets you delete a marker from a file. It displays the dialog in Figure 20-5.

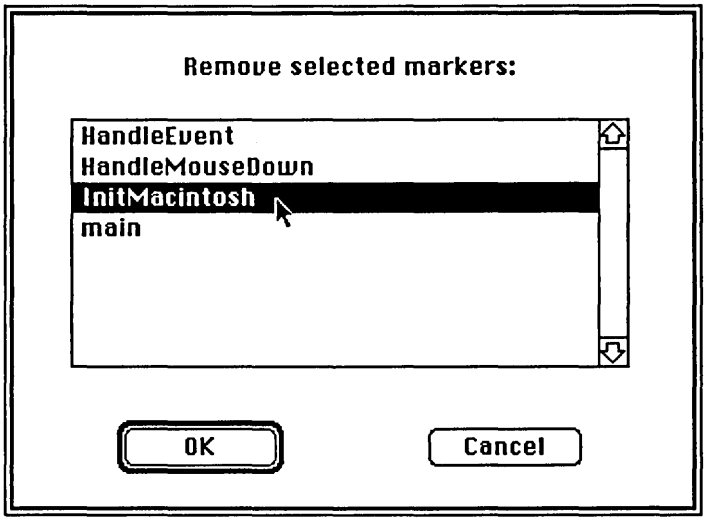
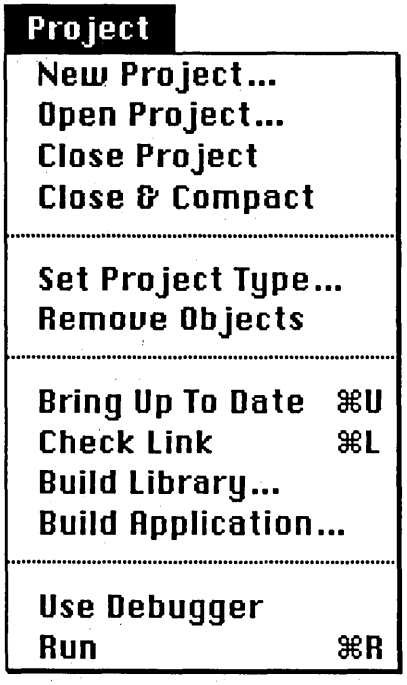


Figure 20-5 The **Remove Marker...** dialog

Select the markers you want to delete and click OK. If you change your mind, click Cancel. The next time you see the marker pop-up menu, you'll notice that the deleted markers are gone.

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'll use when you're ready to build a file containing your application, desk accessory, device driver, or code resource.



New Project... This command creates a new project and opens an empty project window. 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 and then lets you open an existing project or create a new project. If you try to close a project with open files which have been changed but not saved, a dialog box will ask you if you want to save them. To disable this dialog, turn the "Confirm Saves" option off in the Preferences page of the **Options...** menu. When this option is off, changed files will automatically be saved when you close.

Close & Compact This command is the same as **Close Project**, but it makes the project document as small as possible without removing any object code. Use this command before you back up your project, or when you plan to use a project as a library (Chapter 14, "Libraries") or when you plan to transmit the project through a modem (See also the **Remove Objects** command on page 402).

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, Device Driver, or Code

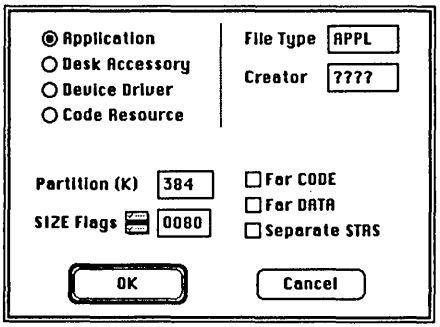
Resource. All project types let you specify the File Type and Creator of the file created by one of the **Build...** commands. To learn the details of each project type, Chapter 7, "The Project."

Set the project type before compiling any of your sources. THINK C will need to throw away any existing object code if you switch the project type once there is compiled code in the project. If the project already contains files, THINK C will ask if you're sure you want to change the project type.

The **Set Project Type...** dialog box lets you specify different attributes for each type.

APPLICATION

The Application dialog lets you specify how string literals and floating point constants are stored and lets you set some of the fields of the `SIZE` *resource MultiFinder uses.*



Partition

The value in this field is the amount of memory MultiFinder allocates for your application. The default is 384K, which is more than enough for most moderate size applications.

SIZE Flags

The pop up menu lets you set the bits that tell MultiFinder and System Software 7 how compatible your application is. You can use the pop up menu or type a hex value into the field. For more information on the menu's choices, see "Setting the partition size and SIZE resource flags" on page 101.

Separate STRS

When this options is set, string literals and floating point constants are placed in their own STRS component. Otherwise, strings and floating point constants are part of the DATA component.

Far CODE

If you need a large jump table, check the "Far CODE" option. It allows a jump table as large as 256K, since it uses address that are 32-bits long instead

of 16-bits long. 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.

Note

If the "Far CODE" option is on, each of your project's files can contribute only 4095 jump table entries.

For more information, see "How Far CODE and Far DATA work" on page 100 and "Mixing Near CODE and DATA with Far CODE and DATA" on page 101.

Far DATA

If you use a lot of global data, check the "Far DATA" option. It allows you to have an unlimited amount of global variables, since it uses address that are 32-bits long instead of 16-bits long. Your application will be about 8% larger because of the longer addresses. If the "Far DATA" option is off, your project can contain only 32K of global variables.

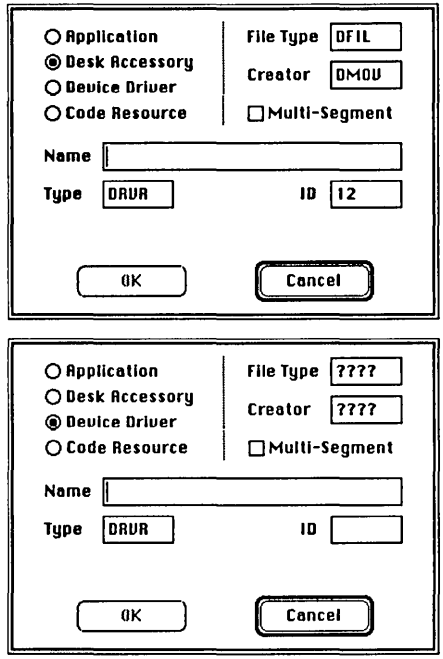
Note

If the "Far DATA" option is on, each of your project's file can contribute only 32K of global data. This limit means, for example, that you can't use an array that's over 32K.

For more information, see "How Far CODE and Far DATA work" on page 100 and "Mixing Near CODE and DATA with Far CODE and DATA" on page 101.

DESK ACCESSORY DEVICE DRIVER

The Desk Accessory and Device Driver dialogs are similar. For desk accessories the File Type and Creator fields are filled in for you so the resulting file will be a Font/DA Mover file. There are other differences between Desk Accessories and Device Drivers which have to do with the header fields. For details, see "Building Desk Accessories and Device Drivers" on page 104.



Multi-Segment

When this option is on, your desk accessory or device driver can have up to 30 segments.

Name

This field is the name of your desk accessory or device driver. By convention, desk accessory names begin with a null byte. THINK C takes care of this for you. Device drivers begin with a period. If you don't provide one, THINK C will add one for you.

Type

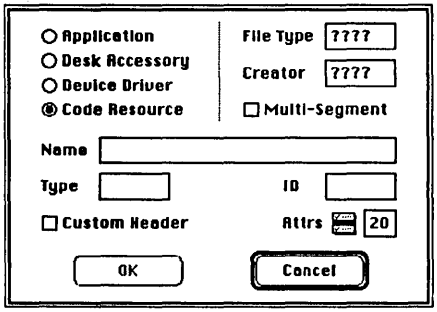
Desk accessories and device drivers are resources of type DRVR. You can change the type if you have some reason for doing so.

ID

The ID number of the DRVR resource. Desk accessories default to 12. The Font/DA Mover will renumber your desk accessory (and its owned resources) if there is a conflict.

CODE RESOURCE

The Code Resource dialog lets you specify whether your code resource will begin with a standard header, its name, type, and ID, and its resource attributes.



Multi-Segment

When this option is on, your code resource can have up to 30 segments.

Name

The name of your code resource. For most code resources, the name is optional.

Type

The resource type of your code resource.

ID

The ID number of your code resource.

Custom Header

When this option is off, THINK C uses a standard header for your code resource. The header places the address of your resource in register `A0` and branches to your `main()` function. If you check this option, your code resource will begin with the first function in the file in which `main()` is defined.

Attrs

The resource attributes (locked, purgeable, system, etc.) of your code resource. You can select the attributes from the pop up menu, or you can set them by typing a hex number into the field.

Remove Objects

This command removes all the object code from a project. It reverses the effects of all previous compilations and loading of libraries. The project document is "dehydrated"; it can be "reconstituted" by recompiling all source files and reloading all libraries.

Use **Remove Objects** when you need to make the project document as small as possible, e.g., for archiving or for transmitting to someone else.

Bring Up To Date

This command compiles and source files that need to be compiled and loads any libraries that haven't been loaded.

Check Link

This command checks for all the same error conditions as **Run** or **Build Application...** would, but without running the project or building an application. If any files need to be made, you will be asked whether you want to bring the project up to date, even if you have set **Confirm Auto-Make** off.

The **Check Link** command displays errors in the **Link Errors** window. If there are multiply defined or undefined symbols, the names of the files containing the symbols appear in parentheses.

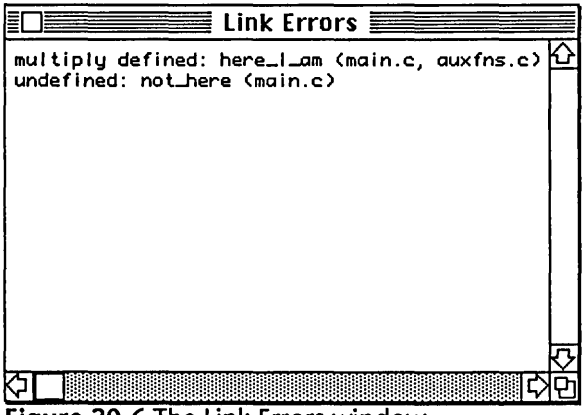


Figure 20-6 The Link Errors window

When the **Link Errors** window comes up as the result of an attempt to **Run** or **Build...**, this additional information is not displayed. Since some disk activity is required to compute the information, it is only displayed when you specifically request **Check Link**.

Build Library...

This command saves the current project as a single binary file that can be added as a library to other project documents. A dialog box prompts you for the name of the library file. The convention is `name.Lib`; however, a library may have any valid file name. Note that you can include a project in another project without first saving it explicitly as a library. See Chapter 14, "Libraries," for details.

Build Application...
Build Desk Accessory...
Build Device Driver...
Build 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 (the dialog on the left is for applications, desk accessories, and drivers; the one on the right is for code resources):

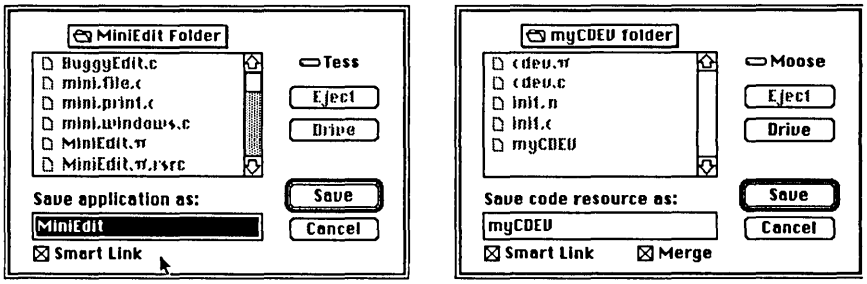


Figure 20-7 The **Build...** dialog, with and without the Merge button.

If the Smart Link option is checked, THINK C uses the smallest number of code components of the source files or libraries to create the resulting file. It takes a little longer to put the application or resource together, but the resulting file will be as small as possible. Uncheck this option if you're building frequently for testing.

If the Merge option is checked in the **Build Code Resource...** dialog, you can place your code resource into an already-existing file. Just type in the name of the file.

If there is a file with the same name of the project that ends in `.rsrc` in the same folder as the project, the resources in the `.rsrc` file will be merged into the resulting file. However, code resources built with the Merge option will not use the `.rsrc` file.

Use Debugger

This command turns the source debugger on and off. When the source debugger is on, you'll see a "bug" column in the project window, and when you run the project the debuggers windows appear on the screen. Selecting this command is the same as clicking on the "Use Debugger" check box of the Debugging page of the **Options...** dialog box. See Chapter 12, "The Debugger," to learn how to use the source level debugger.

Run

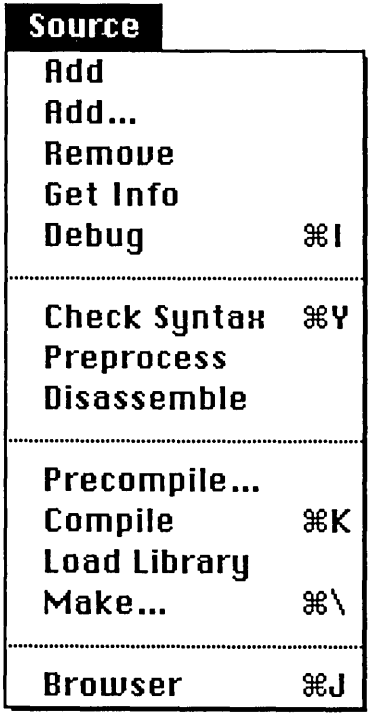
This command will run the program contained in the project. If the project is not up to date, a dialog box will ask if you want to bring it up to date. If the "Confirm Auto-Make" option in the **Options...** dialog box is not checked, then the project will automatically be brought up to date. This lets you edit a

source file and run the changed program, without ever explicitly recompiling or relinking the program.

If the project type is Desk Accessory, the **Run** command uses an auxiliary program, DAShell, to run your desk accessory. THINK C needs to build you desk accessory and save it in a file first, then THINK C launches DAShell. Your desk accessory will be in the **Apple** menu.

The Source Menu

*The commands in the **Source** menu let you add and remove source files to your project. This menu also contains commands to create precompiled headers, to compile source files, to load libraries, and to control the auto-make facility yourself.*



Add

This command adds the file in the frontmost edit window to the project to the project window. The file must end in `.c`.

Add...

This command lets you add existing source files and libraries to a project. It brings up the dialog in Figure 20-8.

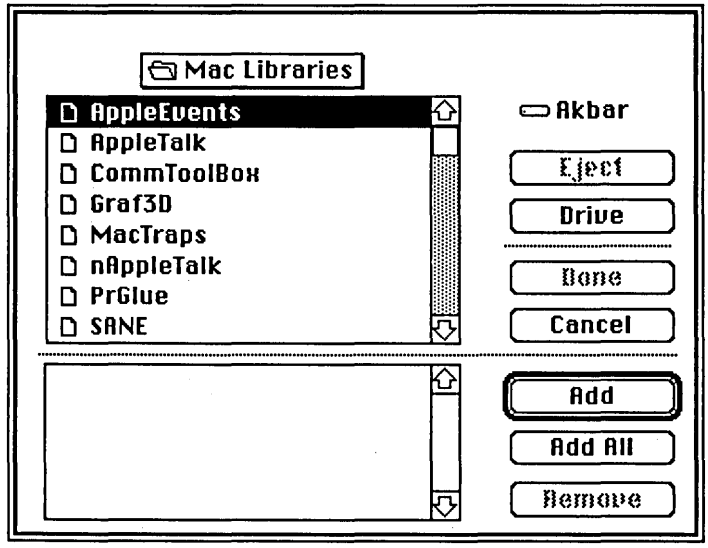


Figure 20-8 The Add... dialog

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 it.
- 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 it. (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.

Remove

This command removes a selected source file or library from the project.

Get Info...

The **Get Info...** command display a dialog box that contains the sizes of each project component. The displays initially contains information for the currently selected file.

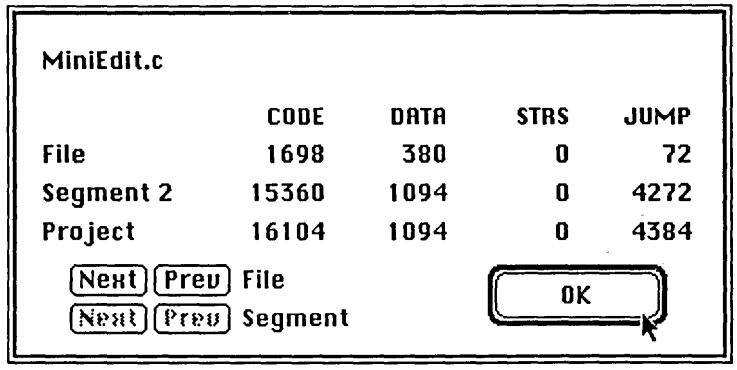


Figure 20-9 The **Get Info...** dialog

The dialog shows the size of the CODE, DATA, STRS, and JUMP components for the selected file. If the file hasn't been compiled, these values will be zero. The dialog also shows the same information for the segment and the entire project. The Next and Prev buttons let you examine the other files and segments in the project.

Check Syntax

This command lets you compile a file in order to check its syntax. This command compiles the front window but does not add the file to the project window or post the results of the compilation to the project document. You can check the syntax of the contents of the edit window, even an untitled window. **Compile**, by contrast, works only on files that end in `.c`.

Preprocess

This command runs the code in the frontmost window through the THINK C preprocessor and displays the result in a new window. It's especially useful if you think you have a bug in one of your macros. The preprocessor expands your macro, includes the contents of your #include files, and evaluates your `#ifdef` statements. You can save and print the contents of this window like you would any other file.

Disassemble

This command dissasssembles the code in the frontmost window and displays the result in a new window. This command helps you figure out how efficient your code is and how to write inline assembly code. You can save and print the contents of this window like you would any other file

Precompile...

This command creates a precompiled header from the contents of the frontmost edit window. Precompiled headers may not have any code or data def-

initions. You can include #include files (even other precompiled headers) in precompiled headers.

Debug

When you're using the source level debugger, this command sends the frontmost edit window (or the selected file in the project window) to the Source window of the debugger.

Compile

This command will compile either the contents of the active edit window, or the currently selected file in the project window. The results of the compilation will be posted to the project document.

Only files that end in .c can be compiled. To check the syntax of a source file without adding it to the project document, use the **Check Syntax** command instead. The **Compile** command is dimmed when a library file is selected in the project window.

Load Library
Load Project

These commands are active when the selected file in a project window is a library or a project. (You can use projects as libraries. See Chapter 14, "Libraries," for details.) When you select this command, THINK C loads the code for the library into the project. To add a library to a project for the first time, use the **Add...** command.

Make...

When you select this command, a dialog box appears showing all the files in the project inside a scroll box. The files that THINK C thinks need to be recompiled (or in the case of libraries, reloaded) are checked. You can alter the list if you like by using the cursor to check or uncheck files, or by clicking the Check All, Check All .c or Check None buttons. Your changes will not be remembered if you click the Cancel button.

When you press the Make button, THINK C will bring the project up to date. It will recompile files that need to be recompiled and load libraries that need to be reloaded. If you press the Don't Make button, the project will be up-

dated next time you use any of the commands that the update the project: **Bring Up To Date**, **Run**, **Check Link**, or **Make...**.

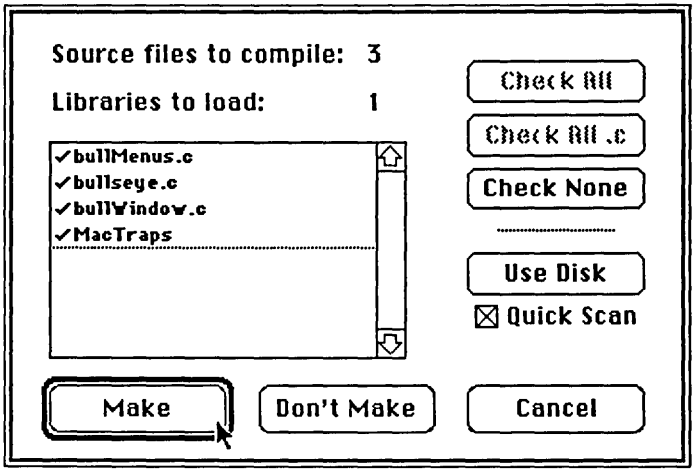


Figure 20-10 The Make... dialog

If you click the Use Disk button and the Quick Scan check box is checked, THINK C checks the date/time-modified of all the files in the project. Normally this is not necessary because THINK C automatically tracks the changes you make as you edit. This knowledge is project-specific, though, so if you have a source file that belongs to two different projects and you change it in one, the other project won't know it's been changed unless you say Use Disk.

If the Quick Scan check box is not checked, THINK C does a more extensive check. If a file can't be found, THINK C searches the tree the file was originally in to find the file. (The Use Disk feature can't help you detect when you've moved files from one tree to another. See "Moving Files Within a Project" on page 157 for details.)

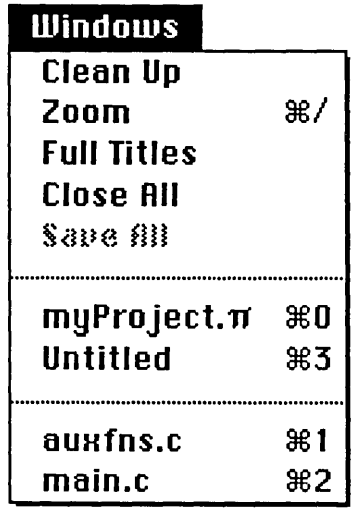
Use Disk displays its progress if the Quick Scan option is off. You can abort it by typing Command-. (command-period), although the next **Use Disk** will start again at the beginning.

If a source file or library is not found by Use Disk, its status (i.e. whether it is checked in the **Make...** box) is unchanged. However, if a source file is found but one of the files it includes is not found, the source file is marked as needing to be compiled (i.e., it is checked).

Clicking Cancel does not undo the effect of Use Disk. Unlike the other buttons, which simply add (or remove) a check mark to those specified by Auto-Make, Use Disk actually updates the date/time record associated with each file that is in the project. You can, of course, manually check or uncheck files after telling THINK C to Use Disk.

Windows Menu

The Windows menu has three sections, separated by dotted lines. The first section has five commands: Clean Up, Zoom, Full Titles, Close All, and Save All. The second section has an entry for the project window and one for each Untitled window. The third section has an entry for each file open in an edit window, in alphabetical order.



Clean Up

Restacks the windows as though they were freshly opened. The rearmost window is assigned the first slot, as though it was the first window opened, the next-rearmost window is assigned the second slot, etc. The front-to-back order of the windows is not changed.

Zoom

Resizes the frontmost window to occupy the full screen. If the window already at full screen, it is restored to its previous position and size. This is the same as clicking the window's zoom box in the upper right corner of the window.

Full Titles

This is a checkable item, initially unchecked. When checked, the title of each edit window indicates the volume name and directory name as well as the file name.

Close All

This command closes all the edit windows. If the "Confirm Saves" option is checked, THINK C asks whether you want to save each modified window, otherwise windows are automatically saved. Holding down the Command or Option key as you click in the close box of an edit window is the same as **Close All**.

Save All

Saves all the modified windows. No confirmation is requested.

Project window

The project window is brought to the front. (The menu item will be the name of the project, not the literal words "Project window".)

Edit Windows

Brings the selected window to the front. The number reflects the "slot number", i.e., the initial position of the window. (The first created window occupies slot #1, and slots #6-10 occupy the same screen positions as slots #1-5, etc. Slots vacated by closed windows are reused at the next opportunity.) The number of windows is limited only by available memory, but only windows in the first nine slots have a Command-key equivalent. A diamond (◊) appears next to windows which have been modified.

Debugger Menus 21

This chapter describes each of the source level debugger's 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 in the menu.

Some of the debugger's commands operate on the selected statement. To select a statement, just click on its line in the Source window. If there isn't a selected statement, the commands operate on the current statement.

Contents

The Apple Menu

Shortcuts...

This command displays a series of dialog boxes that describe some shortcuts that make working with the debugger faster.

The File Menu

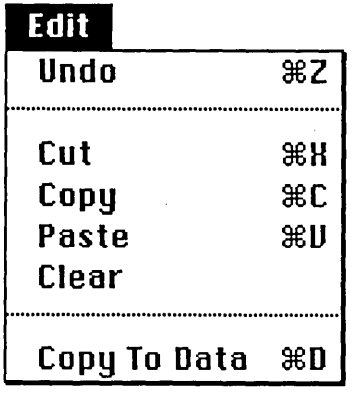
The File menu lets you use save your current debugging session

Save

This command saves your current debugging session, including your breakpoints and the contents of your data windows, so the debugger can restore your session the next time you use the debugger. If the "Always save session" option is on in the Debugging section of the **Options...** dialog, the debugger also saves your session whenever you quit the debugger. The **Options...** dialog is described in "Using the Options... Dialog" on page 179.

The Edit Menu

The Edit menu lets you use the standard editing commands on expressions in the Data window.



Undo

This command undoes any edits you make to an expression in the Data window. Choosing this command is the same as clicking on the deselect button and then selecting the same expression again.

Cut

This command deletes the selected text and copies it to the Clipboard. You can't cut text out of the Source window.

Copy

This command copies the selected text to the Clipboard.

Paste

This command pastes the text in the Clipboard into the current window. You can't paste into the Source window.

Clear

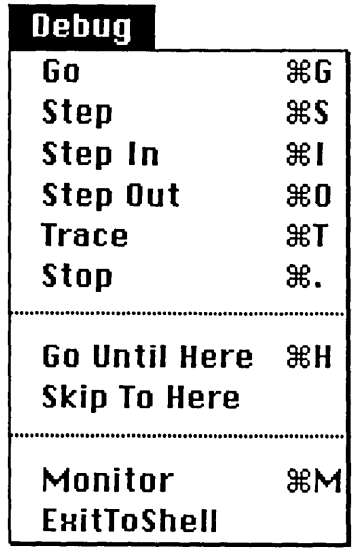
This command removes the selected expression from the Data window.

Copy To Data

This command is active only when the Source window is the frontmost window. It copies a selected expression from the Source window and pastes it into the Data window, where the expression is compiled. It's up to you to make sure that the text selected in the Source window is a valid expression. **Copy To Data** leaves the expression selected, so you can use some of the formatting commands right away.

The Debug Menu

Use the Debug menu to control the execution of your program. The first six commands in this menu have equivalent buttons in the Source window status panel.



Go

The **Go** command starts your program if it was stopped. Your program will run until you stop it (with the Stop button, for example) or until it's about to execute a line with a breakpoint or until it hits an exception (dividing by zero, for instance). If your application is already running, the **Go** command brings it to the foreground.

Step

The **Step** command goes on to the next statement marker in the current function. If you're at the end of a function, **Step** returns to the calling function. Use **Step** when you want to follow the execution within a function without falling into other functions. (Technically speaking, **Step** skips over JSRs.)

Step In

The **Step In** command executes your program until the current statement arrow falls into a function. **Step In** is useful when you want to skip over a set of assignments or Toolbox traps, to fall into the next function call. If **Step In**

reaches the last statement of the current function without falling into another function, it will stop immediately after the current function returns.

Step Out

The **Step Out** command executes your program until the current statement arrow falls out of the current function. This operation can be slow if there's a lot left to do, but it's a sure way of leaving the current routine. A faster way of leaving the current routine is to use the **Go Until Here** command or to set a temporary breakpoint at the last diamond in the function.

Trace

The **Trace** command executes the current statement. In most cases, the statement indicator will go on to the next statement marker, even if the next statement marker is in another function. The only time it won't is when the program counter steps into some code that the debugger doesn't have the source text for. This usually happens when you step into a trap that's not generated in line. So, for a brief period, the current statement arrow isn't really anywhere in your program, but somewhere in `MacTraps` instead. Though you can't see the current statement arrow, the current function indicator at the lower left tells you which file it's in.

Stop

The **Stop** command stops execution of your program. The **Stop** command works when any debugger window is active, and the Stop button only works when the source window is the active window. When you press the Stop button you'll usually be coming out of your call to `GetNextEvent()` or `WaitNextEvent()`.

If your program is not in its event loop, you might not be able to make the debugger the frontmost application. In this case, Command-Shift-Period is the **panic button**. Use Command-Shift-Period to stop execution when one of your application's windows is frontmost or when you think its stuck in a loop. (Command-Shift-Period won't work if you're stuck in an infinite loop in ROM, though.)

Go Until Here

The **Go Until Here** starts execution and stops at the selected line. This command is exactly the same as setting a temporary breakpoint (see "Setting temporary breakpoints" on page 229) at the selected line. Use this command when you want to move through a block of code quickly.

Skip To Here

The **Skip To Here** command changes the program counter to the selected line without executing any intervening code.Use it when you want to skip over code you know to be buggy but not crucial to the rest of the program's operation.

Warning

This command is potentially dangerous. Make sure the code you're skipping to doesn't depend on anything the skipped code does. For instance, it is a very bad idea to skip over initialization routines.

Monitor

The **Monitor** command drops you into a low level debugger. All your registers (including status registers) and low memory globals will be correct. The PC (program counter) will be somewhere in the source debugger, not in your program. You can still get the value of your PC.

If you're using TMON, the PC is in TMON's V register. If you selected an expression in the Data window, its value is in TMON's N register.

If you're using Macsbug, your PC is one long word before the current PC. To look at the instructions in your program, type:

```
DM PC-4
IL @.
```

Note

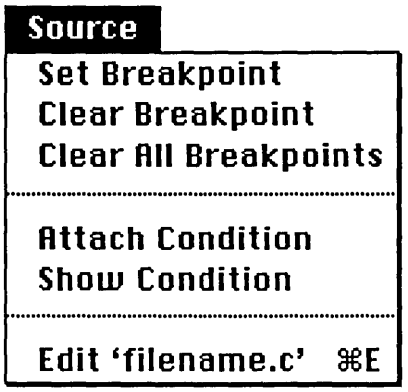
If you don't have a low level debugger installed, don't use the **Monitor** command.

ExitToShell

This command aborts the source level debugger. You should use your application's **Quit** command to quit the debugger. Use **ExitToShell** only if you can't use your application's **Quit** command.

The Source Menu

The Source menu contains commands for working with the Source window.



Set Breakpoint Sets a breakpoint at the selected statement.

Clear Breakpoint Clears the breakpoint at the selected statement

Clear All Breakpoints Clears all the breakpoints in the project.

Attach Condition Use the **Attach Condition** command to attach an expression in the Data window to a breakpoint to create a conditional breakpoint. To set a conditional breakpoint:

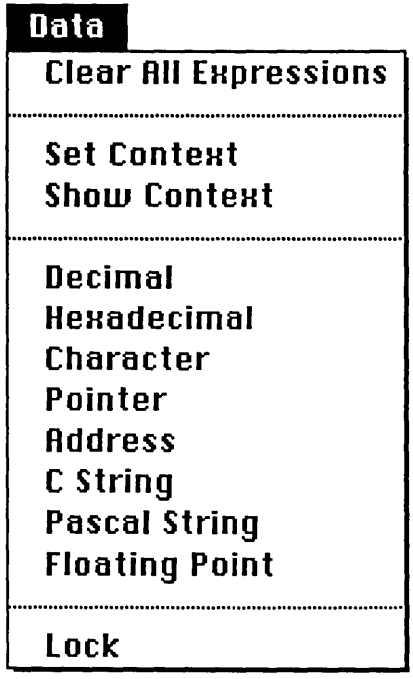
1. Set a breakpoint by clicking on the statement marker diamond
2. Click on the line to select it
3. Click on an expression in the Data window
4. Choose **Attach Condition** from the **Source** menu.

Show Condition Use the **Show Condition** command to display the condition attached to a conditional breakpoint. If the selected statement has a conditional breakpoint (a gray diamond), the attached expression in the Data window will be highlighted.

Edit 'filename.c' Brings THINK C to the foreground and opens an edit window for the file in the Source widow. This is the inverse of the **Debug** command in THINK C's **Source** menu.

The Data Menu

The commands in the Data menu operate on expressions in the Data window. You can set and show the context of expressions, change their display format, and lock expressions to keep them from being reevaluated.



Clear All Expressions This command clears the contents of the Data window and closes all your array and struct windows. It's especially useful if you don't want the debugger to restore the contents of your Data window the next time you use the debugger.

Set Context This command makes the selected statement in the Source window the context of the selected expression in the Data window.

Show Context This command highlights the statement that is the context of the expression selected in the Data window.

Decimal
Hexadecimal
Character
Pointer
Address
C string
Pascal string
Floating Point

These commands control how expressions appear in the Data window. The default format depends on the data type of the expression. The type of the expression also determines what other formats you can use.

The default formats are shown in italics.

Type	Formats Available
integers	*Decimal*, Hex, Char
unsigned	*Hex*, Decimal, Char
pointers	*Pointer*, Address, Hex, C String, Pascal String
arrays	*Address*, C String, Pascal String
structs	*Address*
unions	*Address*
functions	*Address*
floats	*Floating Point*

This is what the display formats look like:

Format	Example
Decimal	`4523345, -23576`
Hex	`0xA09E1487`
Char	`'c', 'TEXT'`
C String	`"abcdef\nghi\33"`
Pascal String	`"\pabcdef\nghi\33"`
Pointer	`0x7A7000 or 0x007A7000`
Address	`[] 0x09FE44, struct 0x08FC14`
Floating Point	`1961.0102`

The C string and Pascal string formats display non-printing characters in backslash form. Whenever it can, the debugger uses the built-in escape characters (`\n`, `\r`, `\b`); otherwise it uses `\nnn`, where nnn is an octal value.

Of course, you can use type casting to use formats that aren't normally available. For example, if you really wanted to see an integer, `i`, as a C string, you would type this expression: `(char * ) i`.

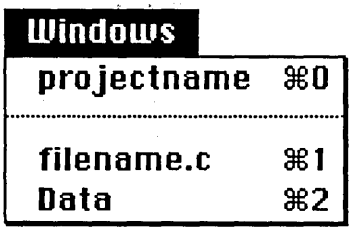
Lock

The debugger reevaluates all the expressions in the Data window every time execution stops. To keep an expression from being reevaluated, select it, then choose **Lock**. When an expression is locked, a small lock icon appears next to it.

To unlock an expression, select the expression, and choose the **Lock** command again.

The Windows Menu

The commands in the Windows menu work on the source debugger's windows.



projectname The real name of this command is the name of your project. Choose this command to bring THINK C to the foreground and make the project window the active window.

filename.c The real name of this command is the name of the file displayed in the Source window. When you choose this command, the Source window becomes the active window.

Data This command makes the main Data window the active window.

Language Reference 22

THINK C is a conforming implementation of the C programming language as defined in the ANSI C standard, *American National Standard for Information Systems—Programming Language—C X3.159-1989.* This chapter describes how THINK C implements those aspects of the language that the standard denotes as implementation defined.

The section numbers in this chapter correspond to the sections in the ANSI standard.

Contents

Introduction

This chapter describes the way THINK C implements the behavior that the standard identifies as "implementation defined." It also documents consistent behavior in THINK C that the standard identifies as "undefined behavior." These behaviors represent conforming extensions. This chapter is not a substitute for the ANSI standard.

Implementation-defined behavior

The standard defines implementation-defined behavior as:

> behavior, for a correct program construct and correct data, that depends on the characteristics of the implementation and that each implementation shall document.

Implementation-defined behavior covers such things as the way error messages are reported, the number of significant characters in identifiers, the format for integers and floating point numbers, and so on.

Undefined behavior

The standard defines undefined behavior as:

> behavior, upon use of a nonportable or erroneous program construct, of erroneous data, or of indeterminately valued objects, for which the standard imposes no requirements.

In most cases, undefined behavior is ignored, generates a diagnosed error, or results in runtime error. This section uses the notation "(Conforming Extension)" for instances when THINK C behaves in a predictable manner for cases that the standard specifies as undefined. If you are writing portable C code, your program should not rely on the behavior described that way.

Setting ANSI conformance

The default options settings for THINK C are not ANSI conformant. To make THINK C an ANSI conformant compiler, use the **Option...** command in the **Edit** menu, choose the "Language Settings" page, and click on the "ANSI Settings" button. Any options that affect ANSI conformance appear on that page.

About the standard libraries

For more information about implementation-defined behavior or THINK C extensions to the standard libraries described in section 4.0 of the standard, refer to the *Standard Libraries Reference* that came with your THINK C package.

The section numbers in this chapter correspond to the sections in the ANSI standard.

Language Reference

2.1.1.2 Translation Phases

The last line of a file does not need to end with a new-line character. (Conforming Extension)

2.1.1.3 Diagnostics

Errors that occur during translation are reported in a Macintosh alert box. The offending line is highlighted in a text editing window. Link errors are reported in a "Link Errors" window.

When the preprocessor encounters an `#error` directive, the words "#error directive" appear in a Macintosh alert box, and the line containing the `#error` directive is highlighted in a text editing window.

2.1.2.2.1 Program Startup

If the `ccommand` function is not used, `argc` is set to 1, and `argv[0]` is the empty string. If the `ccommand` function is used, `argc` and `argv` are set according to the values provided in the `ccommand` dialog box. See the description of `ccommand` in the *Standard Libraries Reference* for more information.

2.1.2.3 Program Execution

In programs that use the console package, the console window is an interactive device. In Macintosh programs, the interactive device is output-only.

2.2.1 Character Sets

The source and execution characters include the full Macintosh character set.

2.2.1.2 Multibyte Characters

There are no shift states for multibyte characters. Multibyte characters are one byte long.

2.2.4.2.1 Sizes of Integral Types <limits.h>

The number of bits in a character in the execution character set is eight.

3.1.2 Identifiers

All characters in an identifier are significant. The number of significant characters in an identifier with external linkage is 255. Case is significant in identifiers with external linkage.

3.1.2.2 Linkages of Identifiers

The same identifier may have internal and external linkage. Internal linkage takes precedence over external linkage. (Conforming Extension)

```
extern int foo;          /* this foo has      */
                         /* external linkage  */
static int foo = 63;     /* this foo has      */
                         /* internal linkage  */
```

3.1.2.5 Types

Integers are represented as 2's complement binary numbers. The sizes of integer types are:

Type	Bytes
`char`	1
`short`	2
`long`	4
`int`	2 or 4

You can select the size of the type `int` with the **Options...** command.

The limits for the integer types are given in the header file `<limits.h>` and documented in the *Standard Libraries Reference.*

THINK C uses five different representations for floating point values. Depending on the options set, different representations map to different types. The floating point representations are:

- 4 byte IEEE single precision
- 8 byte IEEE double precision
- 10 byte SANE extended precision
- 12 byte MC68881 extended precision
- 12 byte universal extended precision

These formats, except THINK C Universal, are documented in detail in the *Apple Numerics Manual, Second Edition* (Addison-Wesley) by Apple Computer and *M68000 Family Programmer's Reference Manual.*

The universal floating-point format is a THINK C extension that lets THINK C use the same format whether the program uses MC68881 code generation or SANE. The universal format is the same as the MC68881 extended format, but it duplicates the exponent part in the second word.

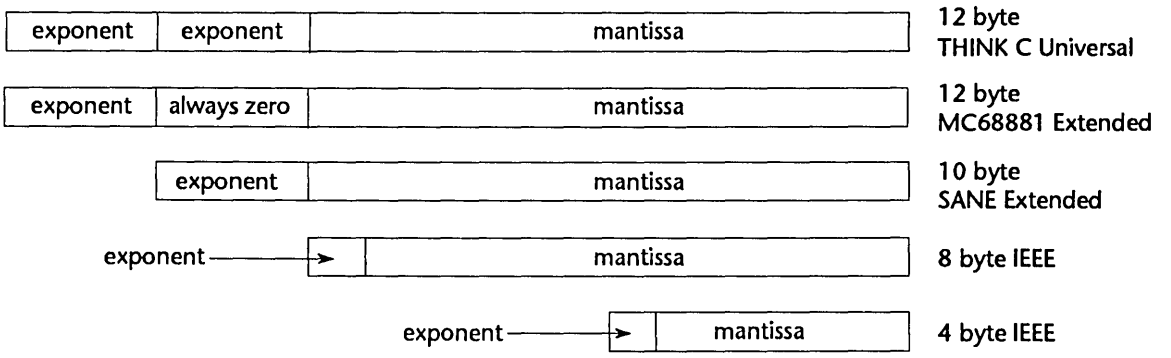


Figure 22-1 Floating-point formats

This table shows how floating-point types map to floating-point formats.

Type	Format	Option settings
`float`	4 byte IEEE	Any option setting
`short double`	8 byte IEEE	"THINK C" language extension on
`long double`	10 byte SANE	"Generate 68881 instructions" off "Native floating-point format" on
`long double`	12 byte MC68881	"Generate 68881 instructions" on "Native floating-point format" on.
`long double`	12 byte universal	"Native floating-point format" off.
`double`	`long double`	"8-byte doubles" off.
`double`	8 byte IEEE	"8-byte doubles" on.

3.1.3.4 Character Constants

If a string literal begins with the sequence `\p` or `\P` it is treated as a Pascal string. The `\p` or `\P` is replaced with the length of the string. The type of Pascal strings is `unsigned char []`, or optionally, `char []`. Pascal strings are not null terminated.

The mapping of characters in the source character set to the execution character set is one-to-one. Multibyte characters are 1 byte long and map one-to-one with single byte characters.

The basic execution character set consists of all 256 Macintosh characters. There are no integer character constants or escape sequences that cannot be represented in the basic execution character set.

An integer constant may contain 1, 2, or 4 characters from the execution character set. The value of a 2-character constant `'AB'` is `0x4142` and of type `short`. The value for a 4-character constant `'ABCD'` is `0x41424344` and of type `long`.

The backslash character \ is ignored for all unspecified escape sequences. (Conforming Extension)

3.1.7 Header Names

In header name preprocessing, header names are treated as character strings with different delimiters. The characters ', \, ", and /* are allowed between the < and > delimiters, and the characters ', \, and /* are allowed between the " delimiters. (Conforming Extension)

3.2.1.2 Signed and Unsigned Integers

If an integer is converted to a shorter signed integer, the low-order bits are retained and the high order bit of the shorter integer is treated as the sign bit.

If an unsigned integer is converted to a signed integer of equal length, and the value cannot be represented in the signed integer, there is no change in representation, but the high bit becomes the sign bit.

3.2.1.3 Floating and Integral

When a value of an integral type is converted to a floating type, if the value being converted is in the range of values that cannot be represented exactly, the result is rounded in the current rounding mode. The default rounding mode is round to nearest, and may be changed by the program.

3.2.1.4 Floating Types

If a floating value being converted is in the range of values that can be represented, but cannot be represented exactly, the result is rounded in the current rounding mode. The default rounding mode is round to nearest, and may be changed by the program.

3.3 Expressions

Bitwise operations on signed integers are carried out as if they were unsigned.

3.3.2.3 Structure and Union Members

If a member of a union object is accessed using a member of a different type, the data stored at that location are treated as if they were a member of the accessing type.

3.3.3.4 The sizeof Operator

The type of the `sizeof` operator is `size_t`, which is defined as `unsigned long`.

3.3.4 Cast Operators

A pointer to a function may be converted to a pointer to an object without losing the value of the pointer. (Conforming Extension)

When converting from a pointer to an integer, if the integer is 4 bytes, then it uses all the bits without a change of representation but interpreted as the type of the integer. If the integer is smaller than a pointer then the low order bytes are used.

3.3.5 Multiplicative Operators

The sign of the remainder is the same as the sign of the dividend.

a	b	a / b	a % b
5	3	1	2
-5	3	-1	-2
5	-3	-1	2
-5	-3	1	-2

Figure 22-2 Results of division and modulus operators

3.3.6 Additive Operators

Adding to or subtracting from a pointer that does not behave like a pointer to an element of an array is allowed. Memory is treated as a linear address space. Pointers that do not behave as if they point to the same array object may be subtracted. (Conforming Extension)

The type of integer required to hold the difference between two pointers is `ptrdiff_t` which is defined as `long`.

3.3.7 Bitwise Shift Operators

A right shift of a negative-valued signed integral type copies the sign bit in.

3.3.8 Relational Operators

Pointers may be compared using a relational operator even if they do not point to the same aggregate or union. (Conforming Extension)

3.5.1 Storage-Class Specifiers

The way THINK C allocates registers depends on the settings in the "Code Optimization" page of the **Options...** menu.

If the "Automatic Register Assignment" option is off, the following registers are available to register allocation:

Registers	Used for
`A2-A4`	Pointer types
`D3-D7`	Integral types, pointer types, and floats if "Generate 68881 instructions" is on
`FP4-FP7`	12-byte floating-point types if "Generate 68881 instructions" is on

If the project is not an application (code resource, device driver, or desk accessory), register `A4` is not available for register allocation.

If the "Automatic Register Assignment" option is on, and the "Honor 'register' first" suboption is on, THINK C places as many variables declared register into registers as described above.

The order that registers are allocated in is unspecified. Structs and unions are never placed in a register, even if they would fit into one.

3.5.2.1 Structure and Union Specifiers

A bit-field may be declared with any integral type.The size of the declared type determines the "word" size for that bit-field, so a "word" may be 8, 16, or 32 bits wide.

A sequence of bit-fields with the same word size are packed into a word. No bit-field may be wider than its word size. If a bit-field would straddle a word boundary, it is placed in the next word.

Bit-fields are assigned beginning with the high-order bit of a word. An unnamed field with a width of 0 "closes out" the current word. A bit-field with a different word size from the preceding bit-field causes this closing out to happen automatically, just as a non-bit-field member does. (Conforming Extension)

A plain `int` bit-field is treated as a `signed int`.

All structures and unions are padded if necessary to be even-size. Padding is inserted after a member only to meet the alignment requirements of the next member or at the end of the structure or union. Only odd-sized data items

(e.g. `char`, odd-sized array of `char`, char-size enums, etc.) do not need to be aligned.

3.5.2.2 Enumeration Specifiers

Enumeration types are of type `int` if the "enums are always ints" option is on. Otherwise the type of an enumeration type is the first one in the following list that is sufficient to hold all the values:

"4-byte ints" off	**"4-byte ints" on**
`char`	`char`
`int`	`short`
`long`	`int`

3.5.3 Type Qualifiers

In a compound assignment to a volatile-qualified type, THINK C may generate a fetch and a store. Otherwise each use of the volatile-qualified type results in a fetch or a store. If the volatile object is larger than 4 bytes, the compiler may need to generate multiple fetches or multiple stores.

3.5.4 Declarators

The maximum number of declarators is at least 12 but usually significantly greater. The actual number depends on the particular combination of declarators.

3.6.4.2 The switch Statement

The maximum number of case values in `switch` statements is limited only by available memory.

3.8.1 Conditional Inclusion

The token `defined` has special meaning only if it appears literally with an `#if` or `#elif` directive. (Conforming Extension)

The value of a single-character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Such character constants can have a negative value.

3.8.2 Source File Inclusion

The mechanism that THINK C uses to locate header files is described in Chapter 9, "Files & Folders."

The name in the `#include` directive is interpreted as a Macintosh file name. Case is not significant, and colons are treated as volume and directory separators.

3.8.3 Macro Replacement

A macro argument that consists of no preprocessing tokens is treated as no tokens. (Conforming Extension)

3.8.6 Pragma Directive

THINK C implements the following `#pragma` directives:

```
once            parameter
option          nooptimize
```

pragma once

When this directive appears in a header file, THINK C includes it only once even if there is more than one `#include` directive that includes the header file.

```
#pragma once
```

pragma parameter

This directive applies to a subsequent inline function definition and allows parameters to be passed in registers instead of on the stack. It specifies which register holds the return value and which registers parameters are passed in. The `#pragma parameter` directive must appear before the inline declaration.

Note

Inline function definitions are not normally part of the ANSI-conformant implementation of THINK C, but are enabled as a conforming extension if a `#pragma parameter` directive appears. For more information about inline function definitions, see "Inline function definitions" on page 177.

`#pragma parameter` *return-reg*$_{opt}$ *function-name* (*param-reglist*$_{opt}$)

Function-name is the name of a function that is subsequently defined inline. If the definition is not an inline definition, never defined, or already defined, the directive is ignored.

The optional *return-reg* can be `__A0`, `__A1`, `__D0`, or `__D1`.

The optional *param-reglist* is a parameter list made up of `__A0`, `__A1`, `__D0`, `__D1`, or `__D2`.

The inline definition must have a prototype and the prototype must not end with `...`. The return type must be an integer type or a pointer type. The return type must be 4 bytes long if the return is in an address register. The argument types may not exceed 4 bytes, except when the inline definition is declared `pascal`, in which case the address of the parameter is used. An address register can hold a 2-byte value for arguments only. In no case can an address register hold a 1-byte value.

Examples:

```
#pragma parameter __A0 NewHandleClear(__D0)
pascal Handle NewHandleClear(Size byteCount)
 = 0xA322;

#pragma parameter Delay(__A0,__A1)
pascal void Delay(long numTicks,long *finalTicks)
 = {0xA03B,0x2280};
```

pragma options

This directive allows programmers to turn certain options on and off from source.

```
#pragma options (option-list)
```

Option-list is a comma separated list of one of the following identifiers. If the identifier is given, the option is turned on. If `!` *identifier* is given, the option is turned off.

option	Meaning if on
`signed_pstrs`	`"\p..."` is `char []`
`virtual`	Methods are virtual by default
`indirect`	Classes are indirect by default
`pack_enums`	Select appropriate size for enum types
`trigraphs`	Decode trigraphs
`require_protos`	Function prototypes are required
`infer_protos`	Infer function prototypes
`check_ptrs`	Check pointer types
`mc68020`	Generate code for 68020
`mc68881`	Generate code for 68881
`macsbug_names`	Generate macsbug names
`long_macsbug_names`	Use long format for macsbug names
`profile`	Generate profiling calls
`honor_register`	Honor register declarations
`force_frame`	Always generate stack frames
`assign_registers`	Assign variables to registers

option	Meaning if on
defer_adjust	Defer and combine stack adjusts
redundant_loads	Suppress redundant loads
global_optimizer	Run global optimizer
gopt_induction	(global optimizer) Induction variables
gopt_cse	(global optimizer) Common subexpression elimination
gopt_loop	(global optimizer) Loop invariants
gopt_coloring	(global optimizer) Register coloring

For more information about using the options pragma directive, see "Accessing Option Settings in Your Code" on page 195.

pragma nooptimize

The nooptimize pragma directive allows the appearance of a call to a function to disable the global optimizer for the function that makes the call.

```
#pragma nooptimize (function-name)
```

Function-name must have already been declared as a function, otherwise it is an error.

Any function that calls a function that has been the subject of the nooptimize directive will not be optimized by the global optimizer, even if the option is on. For an example, look at the implementation of setjmp.

This directive should only be used for functions that have atypical flow-of-control properties.

3.8.8 Predefined Macro Names

The following predefined macro names can be the subject of #undef and #define preprocessing directives: __LINE__, __FILE__, __DATE__, __TIME__, __STDC__. They must be undefined with #undef before they can be defined with #define. (Conforming Extension)

The identifier defined is interpreted specially in context in #if or #elif preprocessing directives. It can be the subject of #define and #undef preprocessing directives. (Conforming Extension)

__DATE__ and __TIME__ are always available.

4.1.5 Common Definitions <stddef.h>

The following types are defined in `<stddef.h>`

Type	Definition
`NULL`	`((void *) 0)`
`size_t`	`unsigned long`
`ptrdiff_t`	`long`

4.2 Diagnostics <assert.h>

The diagnostic printed by assert is of the form "Assertion failed: *expression*, file *xyz*, line *nnn*". The message appears on `stderr`, and, if possible, is followed by the message "press return to exit".

4.3.1 Character Testing Functions

The character testing functions operate on the following ranges:

Function	Range
`isalnum`	`a-z, A-Z, 0-9`
`isalpha`	`a-z, A-Z`
`iscntrl`	`0x00-0x1F, 0x7F`
`islower`	`a-z`
`isprint`	`0x20-0x7E`
`isupper`	`A-Z`

4.5.1 Treatment of Error Conditions

The values returned by mathematics functions on domain errors are documented for each function in the *Standard Libraries Reference.*

The mathematics functions do not set `errno` to `ERANGE` on underflow range errors.

4.5.6.4 The fmod Function

If the second argument to `fmod` is 0, `fmod` returns 0 and sets `errno` to `EDOM`.

4.7.1.1 The signal Function

The set of signals, their semantics, the default handling of signals is described in the documentation of `signal` in the *Standard Libraries Reference.*

No implementation-defined blocking of signals is performed. The default handler is reset if a `SIGILL` signal is received.

4.9.2 Streams

The last line of a text stream does not require a terminating new-line character. Trailing blanks are not stripped off when writing to a text stream. No null characters are appended to a binary stream.

4.9.3 Files

The file position indicator of an append mode stream initially points to the end of the file. Zero-length files actually exist. A write on a text stream does not truncate beyond that point. Writing in place is allowed.

THINK C implements buffered and unbuffered I/O. Line buffered I/O is the same as fully buffered I/O.

The rules for composing valid file names are identical to Macintosh file-naming rules. In general, file names may have up 31 characters, and the `:` character separates directory names. See *Standard Libraries Reference*, Chapter 2, "Using the Standard Libraries," for more information.

The same file can be opened more than once if the Macintosh file system allows it.

4.9.4.1 The remove Function

It is an error to remove an open file.

4.9.4.2 The rename Function

It is an error to use `rename` to give a file a name that already exists.

4.9.5.2 The fflush Function

If you use `fflush` on a stream that is being used for input or update, and the most recent operation was input, the buffer is cleared, and the next read will go to disk. (Conforming Extension)

The `fflush` function always return the buffer to a neutral state, so this function can be used to switch between input to output regardless of the last I/O operation. (Conforming Extension)

4.9.6.1 The fprintf Function

If the conversion specification for the `fprintf` function contains a `#` flag and the conversion specifier is `s`, the matching argument is treated as a Pascal string. (Conforming Extension)

The output for the `%p` conversion in `fprintf` is the same as `%.8lX`, eight digit, hexadecimal integer that uses capital letters and leading zeros.

4.9.6.2 The fscanf Function

The input format for the `%p` conversion in `fscanf` is the same as `%lx`, a hexadecimal integer.

A dash (-) character that is neither the first nor the last character in the scanlist for `%[` conversion is treated as a range specifier. For example, `%[0-9]` means the range of characters from `'0'` to `'9'` inclusive.

4.9.9.1 The fgetpos Function

If `fgetpos` fails, `errno` is set to `ENODEV`.

4.9.9.4 The ftell Function

If `ftell` fails, `errno` is set to `ENODEV`.

4.9.10.4 The perror Function

The function `perror` prints the supplied error string, the string `"Error: "` and the value of `errno`.

4.10.3 Memory Management Functions

If the number of bytes of memory requested from `calloc`, `malloc`, `realloc` is 0 these functions return a pointer to a unique zero-size block of memory.

4.10.4.1 The abort Function

The `abort` function closes all open and temporary files.

4.10.4.3 The exit Function

The argument to the `exit` function is always ignored.

4.10.4.4 The getenv Function

The `getenv` function always returns `NULL`.

4.10.4.5 The system Function

The `system` function always ignores its argument and always returns zero.

4.11.6.2 The strerror Function

The string that `strerror` returns is `"Error: "` and the value of its argument.

4.12.1 Components of Time

The time functions are not aware of daylight savings time or the local time zone.

4.12.2.1 The clock Function

The `clock` function returns the number of ticks (60ths of a second) since the Macintosh was turned on.

THINK C Extensions

This section describes the extensions to ANSI C that are enabled when you set the "Language Extensions" option to "THINK C".

Inline assembly

The identifier `asm` is reserved for the inline assembler. For more information about the inline assembler, see Chapter 13, "Assembly Language."

pascal keyword

The identifier `pascal` is reserved to define functions that follow Pascal calling conventions. For examples, see "Working with Pascal Routines" on page 212. For a description of Pascal calling conventions, see "Pascal Calling Conventions" on page 269.

C++ style comments

Two slashes (//) introduce a comment. The comment ends at the new-line character.

short double type

The type `short double` is allowed. It is an 8-byte IEEE floating point number.

Identifiers after #else and #endif

An identifier after an `#endif` or `#else` preprocessing directive is ignored.

```
#ifdef DEBUG
   printf( "oops!\n");
#endif DEBUG
```

Inline function definitions

Inline function definitions are allowed. The `#pragma parameter` directive may be used to assign registers to parameters. An inline function definition consists of either one hexadecimal literal, or a series of comma-separated hexadecimal literals enclosed in braces. The hexadecimal literals represent machine language instructions.

```
#pragma parameter __A0 NewHandleClear(__D0)
pascal Handle NewHandleClear(Size byteCount)
 = 0xA322;

pascal void PrOpen(void)
     = {0x2F3C,0xC800,0x0000,0xA8FD};
```

When the compiler generates code for an inline function, it merely inserts the machine code.

Low memory global definitions

Global variables may be defined to refer to absolute memory addresses.

```
extern short MemErr : 0x220;
```

This construct is only legal in global scope. The `extern` storage-class specifier is optional. It was required in earlier versions of THINK C.

MC68881 unary inline functions

Unary inline functions for the MC68881 may be defined like this:

```
double atanh (double) : 0x0D;
```

The value after the colon is used as the seven bits of the second word of the MC68881 instruction. See <`math.h`> for examples.

Function prototypes

The parameter list `(...)` is allowed in a function prototype.

Dimensionless arrays allowed

Dimensionless arrays are allowed as the last member of `struct` definitions. The member does not contribute to the size of the struct. For example:

```
struct {
   short count;
   char data[];
} CountData;       /* sizeof(CountData) is 2 */
```

Void *

The type `void *` is compatible with function pointers and data pointers.

Predefined symbols

The preprocessor symbol `THINK_C` is defined as 5 in THINK C 5.0. It is defined as 1 in THINK C 4.0. It is not defined at all for prior versions.

THINK C Object Extensions

This section summarizes the extensions to ANSI C when the "Language Extensions" option is set to "THINK C + Objects". For more information about the way THINK C implements objects, see *Object-Oriented Programming*, Chapter 4, "Using Objects in THINK C."

Keywords

When the "THINK C + Objects" language extension option is on, THINK C reserves the following names as keywords:

```
class       delete
new         virtual
```

The following names are interpreted specially in context, but they are not keywords:

```
direct      indirect      inherited
operator    private       protected
public      this
```

THINK C

Appendices

Part Seven

A What's New
B Troubleshooting
C Error Messages
D Glossary

What's New

A

Just about all of THINK C has been rewritten for THINK C 5.0. The most important change is the new ANSI-conformant compiler.

Note

The section "Porting to THINK C 5.0" on page 458 gives you some guidelines on how to proceed. If you read nothing else in this appendix, read that section.

The other changes you'll find in this appendix include System 7 support, an optimizer, and new object extensions.

Contents

Upgrading THINK C

To upgrade your copy of THINK C, backup and delete all the files that came in the THINK C package and upgrades (but not your own projects and source files). Then follow the instructions in Chapter 2, "Installing THINK C 5.0." Since this upgrade includes many new or revised libraries, you should reinstall everything from scratch.

Compatibility with earlier releases

THINK C 5.0 automatically reads and converts projects created with THINK C 3.0 and later. All object code will be removed, so you will need to recompile the project. THINK C 5.0 can use libraries created with THINK C 3.0 and later. But be sure to use the new version of `MacTraps` and all other supplied libraries.

You will not be able to use projects or libraries from versions earlier than 3.0. Instead, create a new project and add your existing source files.

System 7 Compatible

THINK C 5.0 is System 7 compatible. It lets you and your applications take full advantage of the new system software. THINK C 5.0 does the following:

- Runs with 32-bit addressing
- Produces applications that run with 32-bit addressing (like all versions since THINK C 3.02)
- Runs with virtual memory
- Recognizes all Toolbox routines in *Inside Macintosh VI*
- Contains the THINK Class Library 1.1, with System 7.0 support.

THINK C lets you work with the alias of a project file. The project tree begins where the original project is. However, THINK C does not support aliases in these cases:

- Putting aliases in a project
- Including aliases in an `#include` statement
- Using an alias as a project's resource (`.rsrc`) file
- Inserting an alias of a folder in your THINK C tree or project tree.

100% ANSI-Conformance

THINK C 5.0 is 100% ANSI-conformant. In addition to function prototypes and new-style declarations, it now supports stricter type-checking and the `const` and `volatile` qualifiers.

Since THINK C 5.0 is ANSI-conformant, you may need to revise some programs that compiled under earlier versions of THINK C. To port your code to this new compiler, see "Porting to THINK C 5.0" on page 458.

To compile ANSI-compatible code, you have to set up some options in the **Options...** dialog. For more information, see "Compiling ANSI-Conformant Code" on page 182.

As in the previous version of THINK C, the `ANSI` library is a complete implementation of the ANSI standard library. It's described in the *Standard Libraries Reference*.

Optimizing Compiler

THINK C can optimize your code, so it runs faster and takes up less space. The global optimizer, which adds an additional pass over your code, can do the following:

- Induction variable elimination
- Common sub-expression elimination
- Code motion
- Register coloring.

The THINK C compiler can also perform these optimizations, without the global optimizer:

- Defer & combine stack adjusts
- Suppress redundant loads
- Automatic register assignment.

Other Compiler & Linker Features

The THINK C 5.0 compiler is completely new. This section describes some of the most important features, including more control over the compiler, new ways to look at your code, the project prefix, and options to make porting code easier. And the linker lets you write programs with virtually no limit on the number of routines or global variables.

For information on more new compiler features, see Chapter 10, "The Compiler." To port your existing code to the new compiler, see "Porting to THINK C 5.0" on page 458.

More control over the compiler

The **Options...** dialog includes many more options than earlier versions of THINK C. You now have control over more aspects of the THINK C compiler, including the following:

- How to optimize your code
- Which language extensions to use
- The sizes of `int` and `double`
- How to represent floating-point numbers

For more information, see "Using the Options... Dialog" on page 179.

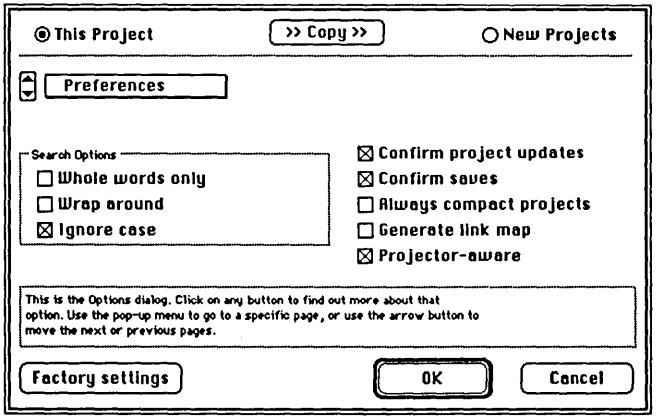


Figure A-1 The Options... dialog

Your can also test and change these options in your code with the `#pragma options` and `__option` directives. For more information, see "Accessing Option Settings in Your Code" on page 195.

New ways of looking at your code

THINK C 5.0 lets you look at your code and finished applications in three new ways to help debug your programs and understand how they work:

- To debug your macros, use the **Preprocess** command in the **Source** menu and see the preprocessor output for your file.
- To see how THINK C compiles your code, use the **Disassamble** command in the **Source** menu and see the assembly code your file produces.
- To see a map of what's in your finished application, use the "Generate link map" option in the Preferences page of the **Options...** dialog.

For more information on these reports, see "THINK C Reports" on page 167.

Including preprocessor code throughout your project

The project prefix lets you include some text in all the source files in your project. It's as if you put the code at the beginning of all your source files. For example, you could put the line `#define DEBUG` in the project prefix, and the symbol DEBUG is defined in all your project's files. You set the project prefix on the Prefix page of the **Options...** dialog. For more information, see "Using the Project Prefix" on page 181.

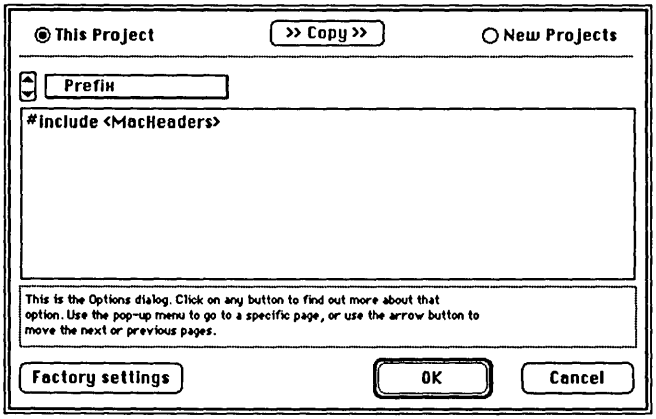


Figure A-2 The Prefix page of the **Options...** dialog

Fewer limits on large programs

In previous versions of THINK C, large programs could run out of space for global data or the jump table. In THINK C 5.0, you have an almost unlimited amount of space for global data and a jump table with the "Far CODE" and "Far DATA" options in the **Set Project Type...** dialog. For more information, see "Using a larger jump table with Far CODE" on page 100 and "Using more global data with Far DATA" on page 100.

Easier to port programs

Porting a program to THINK C can be especially difficult if a program assumes that the type `int` or `double` is a different size from THINK C's. THINK C 5.0 includes two options that let you change the size of those types and port programs more easily: "4-byte ints" and "8-byte doubles." For more information, see "Porting code from other compilers" on page 193.

Debugger Remembers Settings

By default, the THINK C debugger remembers its state when you leave the debugger. The debugger remembers all of your breakpoints, the expressions

in the Data window, and the positions of all your debugger windows. For more information, see "Saving the Debugger State" on page 242.

New Editor Features

The THINK C editor has new features to help you move around files quicker.

New function key shortcuts

The THINK C editor uses more function keys to help you move inside your files faster. This table describes what they do:

Press this key...	**To do this...**
Command-Left-arrow	Move the insertion point left one word.
Command-Right-arrow	Move the insertion point right one word.
Forward Delete (⌦)	Delete the next character.
Home	Scroll to the beginning of the file.
End	Scroll to the end of the file.
Page Up	Scroll to the previous screen.
Page Down	Scroll to the next screen.
Option-Page-Up	Scroll to the left.
Option-Page-Down	Scroll to the right.
Option-Home	Scroll all the way to the left
Option-End	Scroll all the way to the right

Go To Line...

The **Go To Line...** command in the **Edit** menu lets you move to a specific line in your file. For more information, see "Moving to a specific line" on page 132.

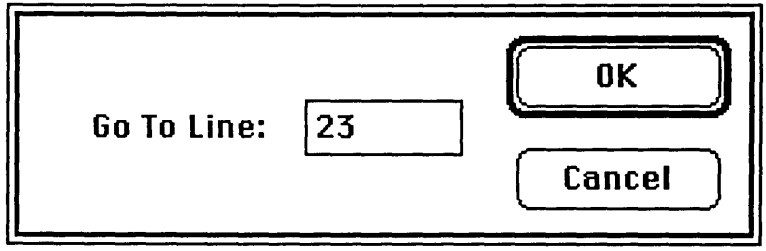


Figure A-3 The **Go To Line...** dialog

Markers

Markers let you move around your program quickly. They work like bookmarks in a book. You can place markers at the beginning of each routine, or

just at certain key points in your program. For more information see "Using Markers" on page 134.

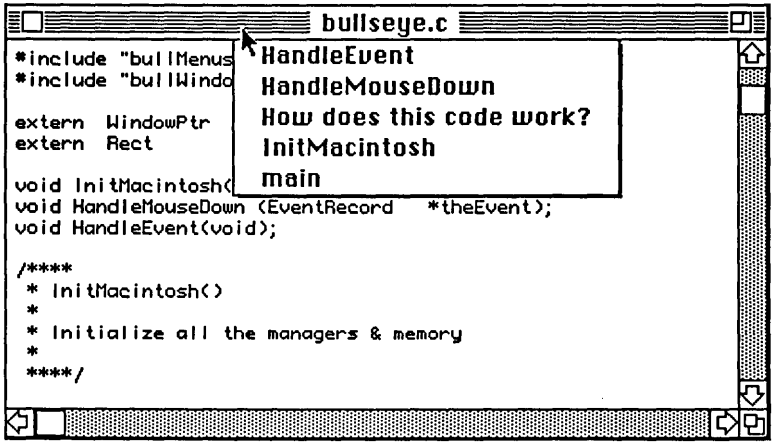


Figure A-4 The marker pop-up menu

New Add... dialog

The **Add...** dialog now lets you add several files to your project at once. For more information, see "Using Libraries" on page 273.

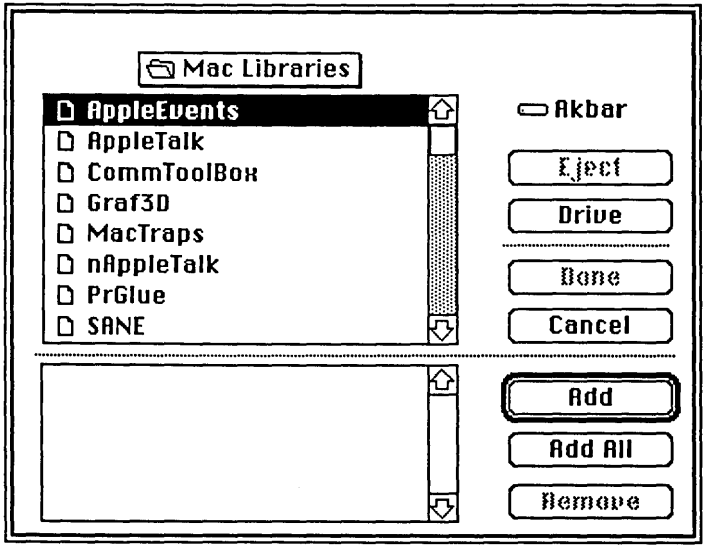


Figure A-5 The Add... dialog

Calling Macintosh Toolbox Routines

THINK C 5.0 handles Toolbox routines differently to take advantage of the new project prefix option and to be more compatible with the interfaces from Apple.

Note

Since THINK C 5.0 supports ANSI C's stricter type-checking you may need to revise some of your code. For more information, see "Porting to THINK C 5.0" on page 458.

Using MacHeaders

THINK C 5.0 no longer has a "<MacHeaders>" option. Instead, to use `MacHeaders`, make sure the line `#include <MacHeaders>` is in the project prefix on the Prefix page of the **Options...** dialog. When you compile, THINK C acts as if the project prefix is at the beginning of all your files. You still can't #include more than one precompiled header in your project.

When you create a new project, THINK C automatically inserts the line `#include <MacHeaders>` in the project prefix. When you convert a project from a previous version of THINK C, THINK C includes the line `#include <MacHeaders>` only if you had checked the "<MacHeaders>" option.

Using MacTraps

Since System 7.0 adds many new routines to the Macintosh Toolbox, the glue for the Toolbox is now in two libraries: `MacTraps` and `MacTraps2`. Old programs need only `MacTraps`. But if you revise your program to take advantage of System 7.0, it also needs `MacTraps2`. `MacTraps` contains glue for routines in *Inside Macintosh I-V*, definitions for QuickDraw globals, and the Gestalt glue. `MacTraps2` contains glue for routines in *Inside Macintosh VI* and less common routines.

Using Toolbox interface files

For more information on calling Macintosh Toolbox routines, see Chapter 11, "Working with the Toolbox."

Beginning with this version, THINK C uses virtually the same header files that Apple provides with its MPW C compiler. In some cases, where the compilers differ, THINK C has its own header files or uses modified versions of the header files. For more information, see "The Macintosh Header Files" on page 208

If your project uses `MacHeaders`, you don't need to change any of your sources. The `MacHeaders` that comes with THINK C 5.0 uses the new files. If you have a custom version of `MacHeaders` or include some Toolbox

header files on your own, you need to update your `#include` statements to use the new file names.

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

This file...	Contains the routines described in...
`Events.h`	Vol. I, Chapter 8, "The Toolbox Event Manager"
`Windows.h`	Vol. I, Chapter 9, "The Window Manager"
`Menus.h`	Vol. I, Chapter 11, "The Menu Manager"

This table will help you update your `#include` statements. However, the new file doesn't always contain all the definitions that were in the old file. If THINK C complains that a function isn't prototyped, look for it in another file.

Old File	New File
`Appletalk.h`	same
`asm.h`	same
`Color.h`	`Quickdraw.h`
`ColorToolbox.h`	`Quickdraw.h`
`ControlMgr.h`	`Controls.h`
`DeskBus.h`	same
`DeskMgr.h`	`Desk.h`
`DeviceMgr.h`	`Devices.h`
`DialogMgr.h`	`Dialogs.h`
`DiskDvr.h`	`Disks.h`
`EventMgr.h`	`Events.h`
`FileMgr.h`	`Files.h`
`FontMgr.h`	`Fonts.h`
`HFS.h`	`Files.h`
`IntlPkg.h`	`Packages.h`
`ListMgr.h`	`Lists.h`
`MacTypes.h`	`Types.h`
`MemoryMgr.h`	`Memory.h`
`MenuMgr.h`	`Menus.h`
`MultiFinder.h`	`Memory.h`
`nAppleTalk.h`	`AppleTalk.h`
`OSUtil.h`	`OSUtils.h` (note the s)
`PackageMgr.h`	`Packages.h`
`pascal.h`	same
`PrintMgr.h`	`Printing.h`
`PrintTraps.h`	same

Old File	New File
Quickdraw.h	same
ResourceMgr.h	Resources.h
SANE.h	same
ScrapMgr.h	Scrap.h
ScriptMgr.h	Script.h
SCSIMgr.h	SCSI.h
SegmentLdr.h	SegLoad.h
SerialDvr.h	Serial.h
SetUpA4.h	same
SlotMgr.h	Slots.h
SoundDvr.h	Sound.h
SoundMgr.h	Sound.h
StartMgr.h	Start.h
StdFilePkg.h	StandardFile.h
SysErr.h	Errors.h
TextEdit.h	same
TimeMgr.h	Timer.h
ToolboxUtil.h	ToolUtils.h
Video.h	same
VRetraceMgr.h	Retrace.h
WindowMgr.h	Windows.h

New Command-key Equivalents

The Command-key equivalents for **Close**, **Print...**, and **Select All** now follow the common Macintosh conventions. These are the new Command-key equivalents:

Command	Key
Close	Command-W
Print...	Command-P
Select All	Command-A

These commands now have command-key equivalents:

Command	Key
Options...	Command-; (Semicolon)
Go To Line...	Command-, (Comma)
Mark...	Command-M
Browser	Command-J

To free up Command-keys, these commands have different Command-key equivalents from what they had in previous versions of THINK C.

Command	New Key	Old Key
Find Again	Command-G	Command-A
Replace	Command-=	Command-P
Replace and Find Again	Command-H	Command-W
Debug	Command-I	Command-G
Get Info	none	Command-I
Make...	Command-\	Command-M

More Object Extensions

THINK C 5.0 makes it easier to write object-oriented programs with more object extensions, a class browser, and an enhanced class library.

More object-oriented programming features

THINK C includes more object-oriented extensions, including:

- The `class` keyword
- Both virtual and non-virtual methods
- Access control, with `public`, `private`, and `protected`
- Allocators and deallocators (also called `operator new` and `operator delete`)
- Constructors and destructors
- Class functions (also called static member functions)
- Class variables (also called static members).

For more information, see *Object-Oriented Programming Manual*, Chapter 4, "Using Objects with THINK C."

Class Browser

The Class Browser window shows the hierarchy of your classes and helps you to explore it. To display the Browser, choose the **Browser** command in the **Source** menu. For more information, see *Object-Oriented Programming Manual*, Chapter 6, "The Class Browser."

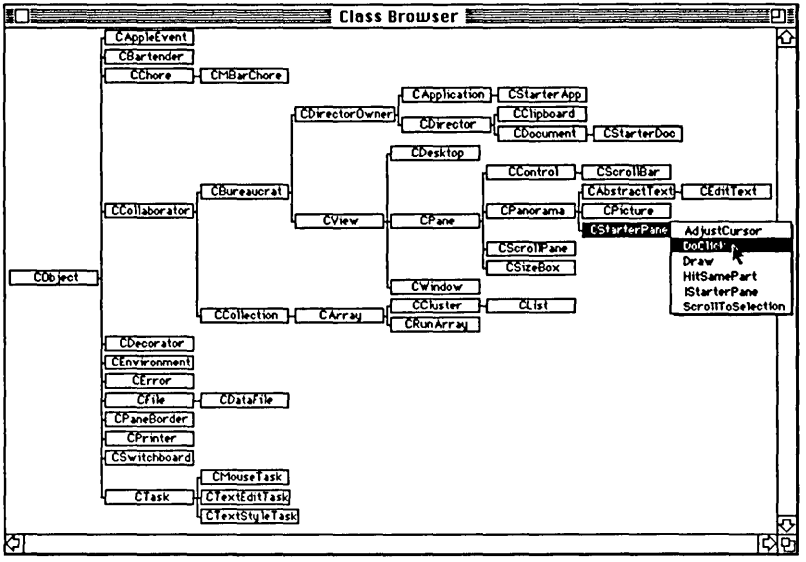


Figure A-6 The Class Browser

Expanded THINK Class Library

THINK C 5.0 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 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 *Object-Oriented Programming Manual*, Chapter 7, "Using the THINK Class Library."

Rewritten Manual on the Standard Libraries

If you use THINK C's standard libraries (like `ANSI` or `unix`), look at the completely revised *Standard Libraries Reference*. It contains more detailed information on how THINK C implements the ANSI standard library and incudes example programs for almost all the functions in the libraries.

Porting to THINK C 5.0

THINK C 5.0 includes a completely new compiler and reorganized header files. If you were careful to write easily portable code, you'll need to revise very little. If you didn't write easily portable code, you'll need to revise much more.

THINK C is now 100% conformant with the ANSI standard for the C language. You can disable its stricter type checking and new features, such as trigraphs, in the **Options...** dialog. For more information, see "THINK C Extensions" on page 174 and "Type Checking" on page 188.

This section describes changes that are especially important for THINK C and Macintosh programmers. For more general advice on ANSI C, read a book on ANSI C, such as *The C Programming Language, Second Edition* by Kernighan and Ritchie, or (better still) a book on porting, such as *Portability and the C Language* (Hayden Books) by Rex Jaeschke.

Double check your Toolbox function calls

In THINK C 5.0, the arguments in your Toolbox function calls must be type-compatible with the function's formal arguments. It's no longer enough for your arguments to be the same size. They now must be the right type, too.

For more information on the "Strict Prototype Enforcement" option, see "Checking pointer types" on page 188.

You can disable the stricter enforcement by turning off the "Strict Prototype Enforcement" option in the Language Settings page of the **Options...** dialog. However, you'll probably find that stricter enforcement saves you time in the long run by catching more bugs. Disable the option only if you're porting code from a previous version of THINK C and you're sure it ran properly.

Stricter prototype enforcement is especially noticeable with arguments that are declared as pointers. It's no longer enough for the argument to be four bytes long. You must now make sure that you cast the argument to a pointer type. For example, the last argument to `GetNextWindow()` must be a pointer, so you must change a call like this

```
GetNewWindow(resID, 0L, -1L);
```

to one of these:

```
GetNewWindow(resID, 0L, (void *) -1);     // OK
GetNewWindow(resID, 0L, (Ptr) -1);        // OK
GetNewWindow(resID, 0L, (WindowPtr) -1); // BEST
```

When you pass a callback routine to a Macintosh Toolbox function, you must prototype it properly. For example, to use a modal dialog filter, you

must declare a filter of type `ModalFilterProcPtr`, which Apple declares like this, in `Dialogs.h`:

```
typedef pascal Boolean (*ModalFilterProcPtr) (
   DialogPtr theDialog, EventRecord *theEvent,
   short *itemHit );
```

You must prototype your filter like this:

```
pascal Boolean MyFilter (DialogPtr myDialog
   EventRecord *myEvent, short *myItem);
```

And call `ModalDialog()` like this:

```
ModalDialog(MyFilter, anItem);
```

If you want even stricter prototype enforcement for your Toolbox calls, comment out the `#pragma options` directive in `Mac #includes.c`, as shown below, and recompile it.

```
// comment out this line for full prototypes
// #pragma options(!check_ptrs)
```

Now you must be sure that your arguments are *exactly* the right pointer type, not just any pointer type. For example, the call to `GetNewWindow()` *must* look like this:

```
GetNewWindow (resID, 0L, (WindowPtr) -1L );
```

long and Point are not the same type

The types `long` and `Point` are the same size, but they are not compatible types. If you want to pass a `long int` as a `Point`, you must cast it. For example, this code won't work:

```
Rect r = { 0, 0, 10, 10 };

if ( PtInRect(0x00050006, &r) ) { ... }
                    // ERROR: 0x00050006 is a
                    //        long literal,
                    //        not a Point
```

Unfortunately, the correction isn't as simple as you might think. You cannot cast a `long` literal to a `Point`. You must assign the value to a temporary variable and cast the variable. Also, you cannot cast a `long` directly to a `Point`. You must take the address of the `long`, cast the address to

(Point *), and dereference it with *. For example, you can change the example above to this:

```
Rect r = { 0, 0, 10, 10 };
long p = 0x00050006;

if ( PtInRect(*(Point *)&p, &r) ) { ... }
                        // OK: p is cast correctly.
```

Or, better yet, to this:

```
Rect r = { 0, 0, 10, 10 };
Point p = { 0x0005, 0x0006 };

if ( PtInRect(p, &r) ) { ... }
                        // BEST: p is a Point
```

short and int are not the same type

In THINK C, int and short int are the same size by default, but they are not compatible types. For example, using this function prototype with this function definition will give you an error:

```
int square (int);
...
short square (short x)  // ERROR:int doesn't
{                       //       match short
   return x*x;
}
```

Note

For more portable code, use short instead of int. Not all compilers for the MC68000 have the same size int, but they all have the same size short.

Beware of promotable types

As the ANSI standard requires, THINK C uses argument promotion to be compatible with old programs. In argument promotion, THINK C promotes a short or char argument to int, and promotes a float or short double argument to double. THINK C performs argument promotion in these two cases:

- Calling a function without a prototype
- Using an old-style function declaration.

Argument promotion can cause a lot of trouble if you mix old-style definitions and new-style declarations. Your best defense is to avoid mixing them.

Either leave your old declarations alone or convert all of them. Don't go halfway.

Argument promotion can also cause trouble if you call a function without a prototype. For example, the following program won't work:

```
main()
{
   short j, i = 2;
   j = square(i);
}

short square(short x)    // ERROR: Declaration
{                        //        doesn't match
   return x*x;           //        THINK C's
}                        //        expectation.
```

When you first use `square()`, THINK C assumes that it returns an `int` and that its argument isn't a promotable type. When THINK C sees how you actually declare `square()`, it assumes you made a mistake and gives you an error.

The best solution to this problem is to use a prototype, so THINK C doesn't create a declaration, like this:

```
short square(short);     // Use a prototype so
                         // there's no confusion.
main()
{
   short j, i = 2;
   j = square(i);
}

short square(short x)    // RIGHT: Declaration
{                        //        matches
   return x*x;           //        prototype.
}
```

Argument promotion can also cause trouble if you prototype a function that you declare with an old-style declaration. In this example, the prototype and declaration don't match since THINK C promotes x from `short` to `int`.

```
int square(short);
...
int square(x)
   short x;          // ERROR: THINK C promotes x
{                    //        to int, so it does
   return x*x;       //        not match prototype.
}
```

For more information on "Require prototypes," see "Type Checking" on page 188. For more information on the Options... dialog, see "Using the Options... Dialog" on page 179.

The best solution is to convert the old-style declaration to a new-style declaration, like the one above. However, if your source files contain a lot of old-style declarations, you have another solution. If you turn on the "Require prototypes" and "Language Extensions" options in the Language Settings page of the **Options...** dialog, THINK C does not promote arguments in old-style declarations. THINK C can do this because, with "Require prototypes" on, it knows you've prototyped all your functions properly.

Function prototypes must have a return type

You *must* include a return type in a prototype. Previous versions of THINK C assumed that the return type was `int`. This was accepted as a prototype in previous versions of THINK C:

```
square (int);      // ERROR: No return type
```

You must include the `int` declaration:

```
int square (int); // RIGHT: Includes return type
```

You can still leave out the return type in a function definition, and THINK C assumes it returns an `int`. However, it's best to always include a return value for more readable code.

Declaring function arguments

THINK C is more strict when you use functional arguments. The function you pass must be declared exactly like the argument in the declaration. For example, this code won't compile:

```
void foo(void);
void bar(int);
void bar(char);

void DoFunc( void f(void) );
                  // A function that takes
                  // another function as an
                  // argument.

main()
{
   DoFunc(foo);   // OK:    foo() matches the
                  //        prototype for the
                  //        argument
   DoFunc(bar);   // ERROR: bar() doesn't match
                  //        the prototype.
   DoFunc(baz);   // ERROR: baz() also doesn't
                  //        match the prototype.
}
```

If your argument must match functions with different argument lists, declare the functional argument with `(...)`. However, it will not match functions that use arguments with promotable types. For example, if you define `DoFunc()` like this:

```
void DoFunc( void f(...) );
```

it can take `foo()` and `bar()` as arguments, but not `baz()`.

You can declare functional prototypes with `(...)` only if the "THINK C" extensions option is on. For more information see "THINK C Extensions" on page 174.

Get rid of extra punctuation

THINK C 5.0 does not allow a semicolon after the closing brace in a function definition For example, this function definition is wrong:

```
int square (int x)
{
   return x*x;
} ;                // ERROR: Semicolon after
                   //        last brace.
```

Troubleshooting B

Troubleshooting is the most time-consuming part of writing any program. This appendix describes how to solve some common problems you might have while using THINK C. Among the problems discussed are

- Crashing
- Bomb alerts, like the one in Figure B-2
- The "out of memory" bug alert
- Unexpected function results.

This appendix does not discuss bug alerts with error messages, like the one in Figure B-1. Appendix C, "Error Messages," describes these messages and suggests how to fix the errors. This chapter does discuss how to avoid bomb alerts, like the one in Figure B-2, and the "out of memory" bug alert.

Figure B-1 A bug alert, explained in Appendix C, "Error Messages."

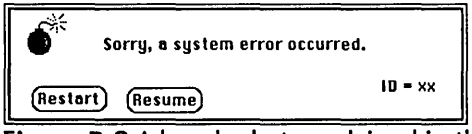


Figure B-2 A bomb alert, explained in this appendix.

Contents

How to Find Your Solution

To find the solution to your problem, start with the section "First Try This..." below. It describes how to solve some problems that can crop up no matter what you're doing.

If those solutions don't work, determine when your problems occurs and look at the appropriate section:

- "Launching THINK C and Opening Projects" on page 469
- "Compiling a Project" on page 470
- "Crashing While Running Your Project" on page 471
- "Getting Unexpected Function Results" on page 473
- "Using the Debugger" on page 476.

If you can't find a solution in the appropriate section, try reading the other sections. Sometimes, you don't notice a problem until well after its cause has passed.

As a last resort, read "When All Else Fails..." on page 477. It tells you how to call Symantec Technical Support.

First Try This...

This section describes how to solve the most common problems that can show up no matter what you're doing in THINK C.

- Check for virus infection
- Investigate virus alerts
- Check for INIT conflicts.

Check for virus infection

Viruses cause a large number of unpredictable problems that you can easily confuse with programming errors. If you program frequently, you need antiviral software, like SAM (Symantec Antivirus for Macintosh), Virex, or Disinfectant. Be sure to read the documentation and use the software diligently.

A common virus is `nVIR`, especially in computer labs or any place where many people use one machine. The `nVIR` virus is easy to find without special software. Open your THINK C application with ResEdit and look for any `nVIR` resources. If you find any, get an antivirus program to eliminate it.

Viruses frequently cause these problems:

- A "Don't Panic" dialog. A classic symptom of the nVIR virus.
- Extra beeping when launching or quitting an application. An nVIR symptom.
- Crashing when you quit an application. An nVIR symptom.
- Getting a bomb alert with ID=-39. An nVIR symptom.
- The "Out of Memory" bug or bomb alert. Viruses sometimes take up memory and do nothing else. If you can't find any other possible causes, look for a virus.

Investigate virus alerts

If your antivirus software detects suspicious activities, it will alert you frequently when you use THINK C. THINK C creates code resources from your source files and adds them to files, usually applications and desk accessories you're creating. Adding code resources is an activity that antivirus software finds suspicious.

If your antivirus software tells you that THINK C is adding or changing a resource to a file, check for a virus. If you have no virus, continue what you were doing and let THINK C make the change. If you can teach your antivirus software to let some programs make changes, you'll want to teach it to let THINK C add and change resources.

Check for INIT conflicts

Poorly behaved INITs (or Startup Documents) also cause a large number of unpredictable problems that are easily confused with programming errors.

If your program has a problem that you can't find in your code, reboot your Macintosh without any INITs. You can do this two ways:

- Remove all the INITS from your system folder and reboot.
- Create a new system folder, put the System and Finder from your old system folder in it, and reboot.

If the problem disappears, you know an INIT helped cause it.

To find a troublesome INIT, remove half your INITs and reboot. If there's no problem, you know the INIT is in the half you removed. If there still is a problem, you know the INIT is in the half you let remain. Split the half with the troublesome INIT again and continue this process until you find it.

Old version of these INITs frequently cause problems with THINK C. If you have any problems, make sure you're using the most recent versions.

- MacroMaker
- Automac
- MasterStrokes
- SuperClock
- early versions of Easy Access
- Radius FPD and Radius TPD
- QuickFolder
- Staircase
- Stepping Out II

Launching THINK C and Opening Projects

This section describes how to solve problems that happen when you launch THINK C or open a project, either by choosing **Open Project...** in THINK C or by double-clicking on a project in the Finder.

Look for your problem in this list, and read its suggested solution. If you can't find your problem here, try going through all the solutions.

- Crashing. Read "Low memory" on page 469.
- The "Out of Memory" bug alert. Read "Low memory" on page 469.
- An error alert with ID=-192. Read "Missing resource error" on page 470.
- An error alert with ID=-39. Read "Check for virus infection" on page 467.
- A "Don't Panic" alert. Read "Check for virus infection" on page 467.
- Extra beeping. Read "Check for virus infection" on page 467

Low memory

If you're using the THINK C debugger, see the "Memory Considerations" on page 246. Otherwise, you may have too small a partition for THINK C or a corrupted project file. Try these solutions:

- Change THINK C's partition size. When you work with a large project, especially one that uses the THINK Class Library, THINK C may need more memory than it usually does. To change the partition size, select THINK C in the Finder, choose **Get Info...**

from the **File** menu, and edit the value in the box labeled "Application Memory Size" (or "Current Size" in System 7).

- 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. Sometimes, a buggy project accidentally writes to its project file and corrupts it. 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.
- Reinstall THINK C. A buggy project may damage your THINK C application. If you think your THINK C application is damaged, reinstall THINK C on a different disk or on a different folder on your floppy. If you can get your work done with the new copy, use it instead of the old one.

Missing resource error

The Macintosh system errors -192 mean that THINK C couldn't find a resource. Here are some solutions for that problem:

To learn how to edit `MacHeaders`*, see "Editing the MacHeaders file" on page 165.*

- Use libraries made with THINK C 5.0. Don't use a project or library, like `MacHeaders`, `MacTraps`, and `ANSI`, made with an old version of THINK C. If you are using your own version of `MacHeaders`, precompile it again with THINK C 5.0. Otherwise use the new versions of these libraries supplied with THINK C 5.0.
- Use THINK C 5.0 to open projects made with THINK C 5.0. If you have two versions of THINK C, you might accidentally launch the old version when you double-click on a project made with the new version. Remove the old version, or double-click on THINK C 5.0 and open your project there.
- Reinstall GateKeeper. If you use the GateKeeper INIT, it may be improperly installed. Reinstall it again, according to its instructions.
- Don't move THINK C when it's running. If you move THINK C into a different folder while it's running, it won't be able to find its files.

Compiling a Project

This section describes solutions to some problems that happen when you compile a project. This section does *not* explain the error messages that appear in bug alerts. They're explained in Appendix C, "Error Messages."

Look for your problem in this list, and read its suggested solution. If you can't find your problem here, see Appendix C, "Error Messages."

- A bomb alert with ID=-192. Read "Missing resource error" on page 470.
- A bomb alert with ID=-39. Read "Check for virus infection" on page 467.
- Crashing while building an application. Read "Corrupted resource file" below

Corrupted resource file

If you think your resource file may be corrupted, try these solutions:

- Let ResEdit fix it. Open the resource file with ResEdit, which tries to fix corrupted resource files.
- Use a released version of ResEdit. Be sure to use the ResEdit included with THINK C. Sometimes, beta versions of ResEdit are buggy.
- Rebuild it. If ResEdit can't fix your file, rebuild it from scratch.

Crashing While Running Your Project

This section describes how to fix problems that cause your program to crash or display a bomb alert.

Look for your problem in this list, and read its suggested solution. If you can't find your problem here, look through all the solutions.

- A "bus error" (ID=1) bomb alert. You tried to access an address that doesn't exist. Read "Pointer errors" on page 472.
- An "address error" (ID=2) bomb alert. You tried to access memory illegally (that is, you try to access a word or longword on an odd-byte boundary). Read "Pointer errors" on page 472.
- A "stack collision with heap" (ID=28) bomb alert. Your program's stack became too large and started to overwrite your program's

heap. Read "Stack errors" on page 473 and "Pointer errors" on page 472.
- An "out of memory" (ID=25) bomb alert. Your program ran out of memory. Read "Pointer errors" on page 472 and "Stack errors" on page 473.
- Damaging your hard disk. You may have written to the I/O address space and corrupted your disk. Read "Pointer errors" on page 472.
- Crashing when you call a function the first time. You may not have enough memory to load in the function's segment. Read "Pointer errors" on page 472 and "Stack errors" on page 473.
- Crashing when you call a function you called before. You may have use an uninitialized pointer that contains the return address for the function. Writing to the pointer corrupts the function. Read "Pointer errors" on page 472.

Pointer errors

Using pointers improperly can cause a large number of problems. If you're having problems with pointers, try these solutions:

- Initialize pointers. When you declare a pointer, THINK C doesn't initialize it or allocate space for what it points to.
- Check array bounds. THINK C doesn't check whether your array subscripts are within the array bounds.
- Check return values. A function that allocates memory or loads a resource may fail and return null. If you use the pointer, you might corrupt low memory.
- Check your resource file. If a function that loads a resource returns null, maybe it couldn't find the resource file. Make sure your resource file is in the same folder as your project and has the same name as your project with `.rsrc` appended.
- Increase your program's partition size. If a function that allocates memory returns null, your program may really be out of memory. Choose **Set Project Type...** to see if your partition is large enough for your memory requests.
- Prototype your functions. You might be passing an argument that's not a pointer to a function that expects a pointer. If you prototype your functions, THINK C displays a bug alert when you make this mistake. To make sure you include prototypes, turn on the "Strict Prototype Enforcement" option and choose

To learn more about resource files, see "Using resource files with projects" on page 96.

To learn more about changing the partition size, see "Setting the partition size and SIZE resource flags" on page 101.

To learn more about the "Require Prototypes" option, see "Enforcing prototype use" on page 189.

"Require prototypes" in the Languages Settings page of the **Options...** dialog.

- Check your arguments. You might pass the wrong argument to a Toolbox routine. For example, calling `DisposHandle()` with the handle to a resource will crash your program. You should use `ReleaseResource()`.
- Pass pointers to **`var`** parameters. If a Pascal function has a **`var`** parameter, pass it a pointer to the argument.

Stack errors

When your application's stack gets too big, it starts overwriting the application's heap. Every sixtieth of a second, the Macintosh Operating System checks for a stack overflow. If one occurs, it brings up a "stack collision with heap" (ID=28) bomb alert. But you may not get a clear sign that the stack is overflowing. Before the Operating System can catch the overflow, it might cause another problem or even crash your project with a "bus error" (ID=1) or "address error" (ID=2) bomb alert.

If you think your stack might be overflowing, try one of these solutions:

- Use less stack space. Use fewer or smaller local variables. Allocate large variables on the heap.
- Increase the stack size. Use `SetApplLimit()` (in *Inside Macintosh II*) to give your project a larger stack.

Getting Unexpected Function Results

This section describes some reasons why functions don't always return what you expect. To find a solution to your problem, read the section that deals with the function you're having trouble with:

If you can't find a solution in the appropriate section, try reading the others and the "Pointer errors" on page 472.

Using your own functions

If one of your own functions isn't returning what you expected, try this solution:

- Prototype your functions. You might be passing an argument that's not the right type. If you prototype your functions, THINK

To learn more about the "Require Prototypes" option, see "Enforcing prototype use" on page 189.

C displays a bug alert when you make this mistake. To make sure you include prototypes, turn on the "Strict Prototype Enforcement" option and choose "Require prototypes" in the Languages Settings page of the **Options...** dialog.

Using the ANSI library

If you have trouble with any `ANSI` library functions, try these solutions:

- Don't profile `ANSI`. Many functions in `ANSI` won't work if the library is profiled.
- Include the right header file. If you don't #include the right header file, THINK C may let you call your function but return invalid results.
- Reinstall the `ANSI` library. If all else fails (including the solutions below), you may have a corrupted `ANSI` library. Reinstall it from your original disks.

If you have trouble with the ANSI I/O functions, try these solutions:

- Check the result of `fopen()`. Make sure it's not null.
- Include `stdio.h`. If you don't #include `stdio.h`, THINK C may let you call the functions but return invalid results. Even if you don't call any I/O functions, you need to #include `stdio.h` if you declare any variables of type `FILE` or `FILE*`.

If you have trouble with the ANSI memory allocation functions, try these solutions:

- Include `stdlib.h`. If you don't #include `stdlib.h`, THINK C may let you call the functions but return invalid results.
- Check your return values. Make sure the allocation function doesn't return null.
- Double-check your arguments. Make sure that the arguments that should be of type `size_t` are `size_t`. Remember that multiplying two integers gives you an integer, not a long integer, even if the result doesn't fit in an integer. To make the result a long integer, cast one of the integers to be a long integer.
- Increase your application's partition size. If a memory allocation function returns null, you might really be out of memory. Choose **Set Project Type...** to see if your partition is large enough for your memory requests.
- Use the right format specifier in `printf()`. When you print the value of a pointer, use either the pointer specifier `%p`, or one of the `long int` specifiers, like `%ld` or `%lx`.

To learn more about changing the partition size, see To learn more about using `printf()`*, see the* Standard Libraries Reference*, Chapter 4, "Using printf and scanf."*

Using Macintosh Toolbox functions

Failing to initialize the Macintosh Toolbox managers is a common source of problems. This function takes care of all the initializations most programs need:

```
void InitMac(void)
{
    InitGraf( &thePort );
    InitFonts();
    FlushEvents( everyEvent, 0 );
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(0L);
    InitCursor();
    MaxApplZone();
}
```

If you still have trouble with the Toolbox functions, try these solutions:

- Double-check your arguments. Make sure that all the parameters are the correct size, of the correct type, and in the correct order. Remember that a Pascal function with a **var** parameter expects a pointer to the argument.
- Check the return values. Check whether functions that allocate memory or get resources return null. Check whether any function returns an error code. These error codes are described with the functions in *Inside Macintosh*.
- Don't use `AddResFile()`. Instead, use a resource file in the same folder as your project. If you use `AddResFile()` while running under THINK C, you corrupt your project's header and you must rebuild your project from scratch
- Fix your resource file. You may have a corrupted resource file. Open it with ResEdit, which tries to fix corrupted resource files. If ResEdit can't fix it, rebuild it.
- Check the name of your resource file. Make sure it has exactly the same name as your project with `.rsrc` appended. Remember that spaces count. For example, the resource file for `My Appl.π` should be `My Appl.π.rsrc`, not `MyAppl.π.rsrc` or `My Appl.rsrc`.
- Check your resource ID numbers. Make sure the ID numbers in the resource file and the ID numbers in your source code match.
- Clean up before sublaunching. Before you launch another application from your application, call `_exiting()` (in the `ANSI` library).

To learn more about resource files, see "Using resource files with projects" on page 96.

To learn more about using `exiting()`*, see the* Standard Libraries Reference.

Using math functions

If you have trouble printing the result of any math function, try this:

To learn more about using `printf()`*, see* Standard Libraries Reference*, Chapter 4, "Using printf and scanf."*

- Use the right format specifier in `printf()`. When you print floating point numbers, use `%g` for the best results. Remember that `%f` rounds to six places after the decimal, so small values print as `0`.

Using the Debugger

This section describes how to solve problems with the THINK C Debugger. For more information on how to use the debugger, see Chapter 12, "The Debugger." If you're having trouble with low memory, see the "Memory Considerations" on page 246.

- Use MultiFinder. The debugger needs MultiFinder to work.
- Make sure you have enough memory. For a small project, you need 2Mb of RAM to use the debugger. For a large project, you may need more.
- Select **Use Debugger**. The **Use Debugger** option in the **Project** menu must have a check mark by it.
- Check where the debugger is. The debugger must be in the same folder as THINK C.
- Make sure there's debugger information for `main()`. Make sure that there is a gray diamond next to the file that contains `main()` and any file you want to debug.
- Turn off the "Use second screen" option. The debugger may be displayed on a second monitor that doesn't exist or is turned off. Turn off the "Use second screen" option in the Debugging page of the **Options...** dialog.

To learn more about changing the partition size, see "Setting the partition size and SIZE resource flags" on page 101.

- Use a smaller partition for your program. If you don't have enough memory for THINK C, the debugger, and your project, you may be able to use a smaller partition for your project. Choose **Set Project Type...** to see if your partition is too big.

To learn more about precompiled headers, see "Precompiled Headers" on page 164.

- Use a precompiled header. Using a precompiled header helps make the debugger tables smaller. If you already use the `MacHeaders` file, you can put more header files in it.
- Check your INITs. Some INITs, like MacroMaker, Automac, and MasterStrokes, interfere with the debugger's display. Remove them, reboot your Macintosh, and try again. See "Check for INIT conflicts" on page 468.
- Turn off debugging for some files. If there's not enough memory for the debugger, turn off debugging for some files.

When All Else Fails...

If none of the solutions in this appendix solve your problem, you may need to call Symantec Technical Support. Before you do, be sure you do the following:

- Isolate your problem. Finding a bug in a large program is difficult. If you think you've found the code causing your problem, write a small program that uses that code.
- Know the System version. To find it, go to the Finder and choose **About the Finder** in the **Apple** menu.
- Know the THINK C version. To find it, launch THINK C and choose **About THINK C** in the **Apple** menu.
- Call from a phone near the computer. The support analyst may need to ask you about the program.
- Have the problem in front of you. Be ready to re-create your problem for the support analyst. Have THINK C running and your project open.
- Be patient.

To find the phone number for Technical Support:

- If you're in North America, look at the *Customer Service Plan* booklet included in your THINK C package.
- If you're outside North America, ask your local Symantec office or distributer.

Note

Other Symantec products (like MORE, SAM, and Norton Utilities) have different Technical Support numbers. Read the *Customer Service Plan* booklet carefully for the right number.

B Troubleshooting

Error Messages C

This appendix is a guide to the error messages THINK C generates.The error messages are in alphabetical order. Error descriptions marked "Assembly" are generated by the inline assembler. Error messages marked "Debugger" are generated by the THINK C Debugger.

8-bit reference to label *'symbol'* out of range

(Assembly) You've used a symbol that's out of range for an assembly language instruction that expects an 8-bit operand. You get this error if you try to do a short branch to a label that's over 127 bytes away from the branch instruction, or use such a label in a PC-relative with index addressing mode. Because the inline assembler optimizes branches to `goto`s, you'll also get this error if the *final* destination of the branch is over 127 bytes away. Example:

```
asm {
            bra.s @foo
            jmp   @foo(d0.w)
            ...     /* over 127 bytes */
foo:

kansas:     bra.s @cLabel
}

cLabel:     goto Oz
            ...     /* over 127 bytes from kansas */
Oz:         printf("We're not in kansas anymore!");
```

Keep in mind that `return`, `continue`, and `break` are treated just like `goto` in the inline assembler.

'symbol' has not been declared

You used the identifier *symbol* before you declared it.

'symbol' is not an argument

You declared *symbol* as if it was an argument in a function definition, but it is not in the parameter list. Example:

```
void function(void)
short left_out_of_parameter_list;  /* Illegal */
{
}
```

'symbol' was not placed in a register

(Assembly) THINK C couldn't place a variable in a register and you tried to use it as a register in an inline assembly language section. Example:

```
void function(void)
{
   register short a, b, c, d, e, f;

   asm {
      moveq #10,f    ; f isn't in a register
   }
}
```

"filename" is not a text file

There are three ways to get this error:

- You tried to include more than one precompiled header.
- A file name in an `#include` statement refers to a file that is not a text file.
- You Option-double-clicked on a symbol to find its definition, and the symbol is defined in a library.

access to *'symbol'* denied

For more information on access control, see Object-Oriented Programming, *Chapter 4, "Using Objects with THINK C."*

The member *symbol* of a class is `private` or `protected`, or was inherited `private`, and it is not accessible. Remember, if you declare a class with the `class` keyword, its members and inheritance are `private` by default. Example:

```
class A {
   short i;
   void foo(void);
};

class B : A {
public:
   short j;
   void bar(void);
};
```

```
class C : B {
public:
   short k;
   void xyzzy(void);
};

class D : public B {
public:
   short k;
   void xyzzy(void);
};

void function(void)
{
   A *p = new A;
   B *q = new B;
   C *r = new C;
   D *s = new D;

   p->i = 0;    // NO: i is private.
   q->j = 0;    // OK: j is public.
   r->j = 0;    // NO: j is inherited private.
   s->j = 0;    // OK: j is inherited public.
}
```

access to 'delete' denied

For more information on access control, see Object-Oriented Programming, Chapter 4, "Using Objects with THINK C."

You tried to `delete` an object, and its class defines its destructor as private or protected. Note that all members of a class declared with the `class` keyword are `private` if no access specifier is provided. You might declare constructors and destructors `private` to control allocation and deallocation through `static` member functions. Example:

```
class ClassC {
public:
   static void destroy(ClassC *);
private:
   ClassB   *b;
   ~ClassC(void);
};

void ClassC::~ClassC(void)
{
   delete b;
}

void ClassC::destroy(ClassC *c)
{
   delete c;       // OK.
}
```

```
void function(void)
{
   register C *p;

   delete p;       // NO: Destructor is private.
   C::destroy(p); // OK.
}
```

access to 'new' denied

For more information on access control, see Object-Oriented Programming, Chapter 4, "Using Objects with THINK C."

You tried to create an object with `new`, and its class defines its constructor as `private` or `protected`. Note that all members of a class declared with the `class` keyword are `private` if no access-specifier is provided. You can declare constructors and destructors `private` to control allocation and deallocation through `static` member functions. Example:

```
class ClassC {
public:
   static ClassC *make(void);
private:
   ClassB   *b
   ClassC();
};

ClassC::ClassC(void)
{
   b = new ClassB
}

ClassC *ClassC::make(void)
{
   return(new ClassC);   // OK.
}

void function(void)
{
   register ClassC *p;

   p = new ClassC;       // NO: Constructor is
                         //     private.
   p = ClassC::make();   // OK.
}
```

addressing mode requires 68020

For more information on using the inline assembler, see Chapter 13, "Assembly Language."

(Assembly) You're trying to use an MC68020 addressing mode, but MC68020 inline assembly is not enabled. Example:

```
asm {
   clr.w (100000,a0)        ; NO
   clr.w (10000,a0)         ; OK: Optimized to
                            ;     10000(a0)
   clr.w (1000,a0,d0.w)     ; NO
   clr.w (100,a0,d0.w)      ; OK: Optimized to
                            ;     100(a0,d0.w)
   clr.w 100(a0,d0.w*2)     ; NO: Scale factor
                            ;     must be 1.
}
```

argument list is inappropriate here

You declared a function with an old-style argument list but did not provide a function body. Usually, you specified a prototype for a function which uses a type name that is undefined. Example:

```
short foo(Str255);    // NO: If Str255 isn't
                      //     defined.

#include <Types.h>
short foo(Str255);    // OK
```

argument to function does not match prototype
argument to function *'symbol'* does not match prototype

For more information on type checking and prototypes, see "Type Checking" on page 188.

The sole argument to a function does not match the prototype. The prototype may be explicit or inferred. Example:

```
void print(char *);
void function(void)
{
   char *msg = "hello, world\n";
   Str255 s;

   print(s);    // NO: Str255 is 'unsigned char'
   print(msg); // OK.
   error(s);    // OK: Infers
                //     int error(unsigned char *)
   error(msg); // NO: Doesn't match inferred
                //     prototype.
}
```

argument #*n* to function does not match prototype
argument #*n* to function *'symbol'* does not match prototype

For more information on type checking and prototypes, see "Type Checking" on page 188.

The given argument to a function does not match the prototype. The prototype may be explicit or inferred.

'break' not in loop or switch

The `break` statement must be inside a loop (`while`, `do-while`, `for`) or a `switch`. Example:

```
void function(void)
{
   short i;
   break;          // NO

   while(1)
      break;       // OK

   do { break; }   // OK
      while(1);

   switch(i){
   case 1:
      break;       // OK
   }
}
```

call of non-function

An expression in the function call position does not evaluate to a function or a pointer to a function. Example:

```
void function(short (*f_ptr)())
                              // f_ptr is a pointer
                              // to a function.
{
  short i, j, k, result;
  result = i(j+k);            // NO: May be missing
                              //     something
                              //     after i.
  result = (*f_ptr)(i,j);     // OK
  result = f_ptr(i, j);       // OK: Same as above
}
```

can't allocate stack temporary

The compiler tried to allocate a temporary variable beyond the first 32K of the local stack frame. THINK C uses temporary variables to store intermediate values in SANE floating-point operation and the return values of functions returning `structs`. You should reduce the size of the local stack frame

(by making large arrays `static`, for example) or move the calculations into another function. Example:

```
void function(void)
{
   char a[32762];
   float x = 2.0;

   x = x * x + 3.0;  // NO: Needs temp
                     //     beyond -32768(a6)
}
```

can't assign to array

You cannot assign, increment, or decrement arrays. Example:

```
void function(void)
{
  char a1[4];
  char a2[4];
  char a3[4];

  if (a1 == a2)// OK: Compares arrays' addresses
      a1++;    // NO: Can't increment arrays
  a1 = 0;      // NO: Can't assign to an array
  a1 = a2;     // NO: Can't assign arrays
}
```

can't assign to function

You cannot assign, increment, or decrement functions. Example:

```
void function(void)
{
  short f1();
  short f2();
  short f3();

  if (f1 == f2)// OK: compares functions'
               //     addresses
    f1++;      // NO: can't increment functions
  f1 = 0;      // NO: can't assign to a function
  f1 = f2;     // NO: can't assign functions
}
```

can't call constructor *'symbol'* directly

You can invoke a class's constructor only by using new. You can't call it like a member function. Example,

```
class C {
public:
   C();
};

void function(void)
{
   register C *p, *q;

   p = new C;  // OK
   q = p->C(); // NO
}
```

can't concatenate normal/wide strings

You can concatenate only wide strings with wide strings or normal strings with normal strings. Example:

```
char s1[] = "hello,"   "world".  // OK
char s2[] = "hello,"   L"world". // NO
char s3[] = L"hello,"  "world".  // NO
char s4[] = L"hello,"  L"world". // OK
```

can't do that with multi-segment project

Your project is multi-segment, and what you're doing is allowed only with single-segment projects. You see this message when you

- Use the **Build Library...** command on a multi-segment project
- Use a multi-segment project as a library,
- Use the **Build Device Driver...**,**Build Desk Accessory...**, or **Build Code Resource...** command on a multi-segment project when the "Multi-Segment" option is not on.

can't load STRS in this project

You're trying to load a library into a project. The library has "Separate STRS" option on, and the project has the "Separate STRS" option off. You need to rebuild the library with the "Separate STRS" off, or turn the option on in the project that's using the library.

can't modify 'const' location

You are trying to assign to a `const` variable, or to a location referred to by a pointer to a `const`. Example:

```
const short x = 0;// constant short
const short *p;   // pointer to constant short
short * const q;  // constant pointer to short

void function(void)
{
   x = 1;    // NO
   p = 0;    // OK
   *p = 0;   // NO
   q = 0;    // NO
   *q = 0;   // OK
}
```

can't open file *"filename"*

THINK C cannot open the file in an `#include` statement. Its name may be misspelled, it may not exist, or it may be in the wrong tree. For more information, see Chapter 9, "Files & Folders."

can't run background-only project

For more information, see "Setting the partition size and SIZE resource flags" on page 101.

You can't run your project under THINK C if the Background Only bit is set in its `SIZE` resource. You set that bit in the SIZE Flags menu in the **Set Project Type...** dialog.

can't take address of method *'symbol'*

You can't take the address of a method unless it is `static`. Unlike C++, THINK C does not support pointers to non-`static` methods. Example:

```
class C {
public:
   void g(void);
   static void f(void);
} *cp;

void foo(void)
{
   void *p;

   p = &cp->g; // NO.
   p = &C::f;  // OK: f is static.
}
```

'case' not in switch

You used the `case` keyword outside of a `switch` block.

character literal must be 1, 2, or 4 bytes

A character literal must contain exactly 1, 2, or 4 characters. Example:

```
char c1 = 'A';          // OK
char c2 = 'AB';         // OK
char c3 = 'ABC';        // NO
char c3 = 'ABC ';       // OK
char c3 = '\n\n\n\n';   // OK
```

class declarations must appear outside any function

You must declare a class globally. You cannot declare it within the body of a function.

class must have a name

Unlike a `struct`, `union`, or `enum`, a `class` declaration must have a tag name. Example:

```
typedef class {       // NO.
   void foo(void);
} C;

class C {             // OK: Compiler inserts
   void foo(void);    //     implicit
};                    //     'typedef class C C'.
```

code overflow

You have more than 32K of code or data in one file. Break your file up into smaller files.

constant required

THINK C expected a constant but didn't find one. Example:

```
#define aConstant  12
void function(void )
{
  short h;
  short x[2];              // OK: 2 is a constant
  short y[h];              // NO: h isn't a
                           //     constant
  enum {
     color = h,            // NO: enum values must
                           //     be constant
     size = aConstant      // OK
  } attributes;

  struct {
      unsigned illegal_bitfield : h;    // NO
      unsigned legal_bitfield : size;   // OK
  } bits;

  short z[size];           // OK: Array dimension
                           //     is a constant
 }
```

code segment too big

See "link failed" on page 506.

contains a function call

(Debugger) The expression in the Data window contains a function call. In Figure C-1, entering `rand()` displays an error message, but entering `rand` displays the address of the function.

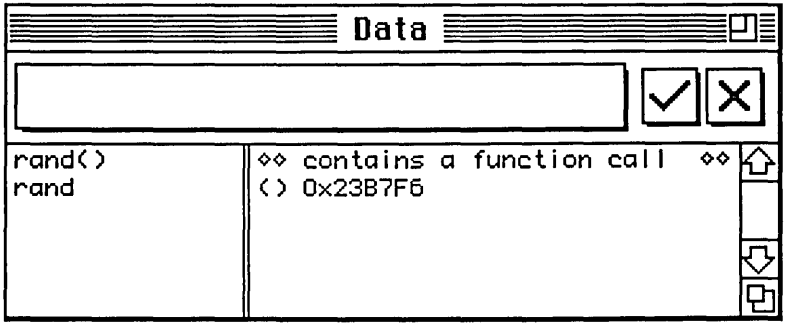


Figure C-1 The "contains a function call" error message

'continue' not in loop

The `continue` statement must be inside a loop (`while`, `do while`, `for`). Example:

```
void function(void)
{
  short i;
  continue;           // NO

  while(1)
      continue;       // OK

  do { continue; }    // OK
      while(1);

  switch(i){
  case 1:
      continue;       // OK
  }
}
```

could not evaluate constant expression

An expression that's in a position where THINK C requires a constant (such an enumeration constant or array size) is too large for the compiler to evaluate. Example:

```
enum { foo = 1 << 33 }; // NO.

void function(void)
{
   short x;
   x = 1 << 33;          // OK: Yields 0 at
}                        //     runtime.
```

data segment too big

See "link failed" on page 506.

declaration of *'symbol'* does not match #pragma parameter

There is a `#pragma parameter` directive for a function named *symbol*, and the number or types of arguments in the prototype do not match those in the pragma. Example:

```
#pragma parameter __A0 myfunc(__D0)

void myfunc(long);              // NO
short myfunc(long);             // NO: Can't return
                                //     short in A0
void *myfunc(long, ...);        // NO: Can't use
                                //     '...'
long myfunc(Rect r);            // NO: Can't pass
                                //     struct in D0
pascal long myfunc(Rect r);     // OK: Passes address
                                //     of Rect
```

declarator too complex

The declaration statement is too complex. You're unlikely to get this error unless you're compiling a computer-generated program.

'default' not in switch

You used the `default` keyword outside of a `switch` block.

displacement too large for this addressing mode

(Assembly) The displacement portion of an addressing mode is out of range for that mode. Usually, you used an indexed addressing mode with a displacement less than -128 or greater than 127. Example:

```
asm {
   clr.w 200(a0,d0.w)     ; NO.
   clr.w 0xFE(a0,d0.w)    ; NO: Hex numbers are
                          ;     unsigned.
   clr.w -0x02(a0,d0.w)   ; OK.
   clr.w 100000(a2)       ; NO: Displacemnt must
                          ;     fit in 16 bits.
   clr.w (100000,a2)      ; OK, If MC68020 is on.
}
```

duplicate 'case'

Within a `switch` block, there are two or more `case` expressions with the same value.

duplicate 'default'

You have more than one `default` in a `switch` block.

duplicate register in argument list

You used the same register more than once in the argument list of a `#pragma parameter` directive. Example:

```
#pragma parameter __A0 foo(__D0, __D0); // NO
#pragma parameter __A0 bar(__A0);       // OK
```

escape sequence doesn't fit in a byte

The octal or hexadecimal escape sequence specifies a number which is too large to fit in a byte. Example:

```
short x = '\777'; // NO: octal 777 = decimal 511
```

'enum *symbol*' already defined

There is already an enum with the tag *symbol*. Example:

```
enum Days { Monday, Tuesday, Wednesday,
            Thursday, Friday };
enum Days { Mothers, Fathers, Secretaries,
            Memorial, Labor };
```

'enum *symbol*' already in use

The enum tag *symbol* is already used as a `struct` or `union` tag. Example:

```
typedef struct X { short x; float y; };
typedef enum X { a, b, c, d };
```

enumeration constant too large

For more information on the "enums are always ints" option, see "enums of any size" on page 176. For more information on the "4-byte ints" option, see "Porting code from other compilers" on page 193.

The value of an enumeration constant is too large for a 4-byte `int`. This error happens only if the "enums are always ints" option is on and "4-byte ints" option is off. Values that cannot be represented in 32 bits bring up the error "could not evaluate constant expression" on page 490.

#error directive

The preprocessor came across an `#error` directive. You can use the `#error` directive to report on any potential problems that the preprocessor can detect. Example:

```
#if __option(trigraphs)
   #error Can't use trigraphs with File Manager.
#endif

#define FILE_TYPE    '????'
#define FILE_CREATOR '????'
```

expression too complex

For more information on the `#pragma options` *directive, see "Accessing Option Settings in Your Code" on page 195.*

An expression is too complex. Usually, there are not enough scratch registers to evaluate a complex addressing expression. Try splitting the expression into subexpressions. You can also disable automatic register allocation for the function so more registers are available. To disable automatic register allocation, use this line:

```
#pragma options(!assign_registers)
```

first argument to function does not match prototype
first argument to function *'symbol'* does not match prototype

For more information on type checking and prototypes, see "Type Checking" on page 188.

The first argument to a function does not match the type in the prototype. The prototype may be explicit or inferred.

floating constant too large

A floating-point constant was too large (or too small) to be represented. For various limits of floating-point types, see *Standard Libraries Reference*, Chapter 2, "Using the Standard Libraries." Example:

```
long double x = 1.2e100000;
                // NO: Maximum exponent is 4931.
```

has side-effects

(Debugger only) You entered an expression that has a potential side effect: =, ++, or --. In Figure C-2, the debugger won't evaluate `i++` and `i+=1` because they have side effects.

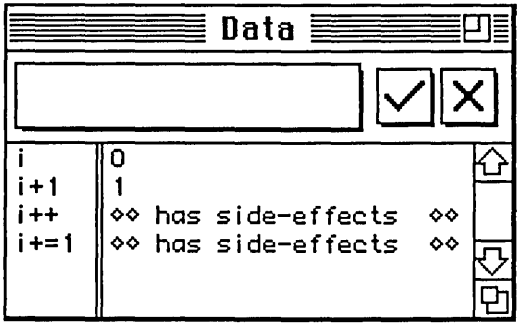


Figure C-2 The "has side-effects" error message

illegal #elif
illegal #else

Every `#else` or `#elif` must have a matching `#if`, `#ifdef`, or `#ifndef`. Example:

```
#ifdef name
#else
#else              // NO: No matching #ifdef
#endif
#elif condition    // NO: No matching #if
```

illegal #endif

Every `#endif` must have a matching `#if`, `#ifdef`, or `#ifndef`.

illegal application of '&'

The address-of operator was used on an object (such as a register variable or a bitfield) that doesn't have an address. It is not an error to take the address of an inline function. The compiler generates a reference to a function with the same name, and brings up the error "link failed" on page 506. Example:

```
pascal void Debugger(void) = 0xA9FF;
void function(void)
{
  register short i;
  short *p = &i;          // NO: i is a register
  void *q = &Debugger;    // OK: But probably not
}                         //     what you meant.
```

illegal application of 'pascal'

Only functions or function types can be declared `pascal`. Example:

```
pascal short f(void);     // OK.
pascal short i;           // NO.
pascal short *p(void);    // OK: pascal function
                          //     returning ptr to
                          //     short.
pascal short (*q)(void);// OK: pointer to pascal
                          //     function
                          //     returning short.
```

illegal application of 'sizeof'

You cannot use `sizeof` with a function or a function type. Example:

```
short f(void), (*g)(void);

short a = sizeof(f); // NO
short b = sizeof(g); // OK: b = sizeof pointer
short c = sizeof(*g);// NO
```

illegal array bounds

The bounds of an array can't be negative or zero. Examples:

```
short a[-7];    // NO: bounds can't be negative
short b[0];     // NO: bounds can't be zero
```

illegal array element type

You can't declare an array of functions, or of `void` objects. However, you can declare an array of pointers to functions or of pointers to `void`. (`void *x[]` is a C idiom for an array of generic pointers.) Example:

```
short array_of_functions[]();                 // NO
void abyss[SIZE];                             // NO
void *generic_ptr_array[SIZE];                // OK
short (*array_of_ptrs_to_funcs[SIZE])();      // OK
```

illegal cast

You cannot cast `structs` or `unions` to other types. You can cast numerical values or pointers to other numerical values or pointers. Example:

```
typedef struct {short v, h;} Point;
void function(void)
{
  Point p;
  long l;
  p = (Point) l;      // NO
  p = *(Point *) &l; // OK
}
```

illegal character *'character'*

A character that's not part of the C character set (such as `@` or `$`) appears in your source file outside of a comment or string. If the character isn't a printable ASCII character, the error message shows it's hexadecimal value. Example:

```
asm {
   dc.w  $A9FF        ; NO: Use 0xA9FF.
}
```

illegal floating-point operation

You cannot use some of the C operators on floating point values. Example:

```
void function(void)
{
  double d1;
  double d2;
  double d3;
  d3 = d1 % d2; // NO: Can't do floating modulo
  d3 = d1 << 3; // NO: Can't do floating shifts
  d3 = 3 << d1; // NO: Can't do floating shifts
}
```

illegal function return type

A function cannot return an array or a function. However, a function can return a pointer to an array or a pointer to a function. Example:

```
typedef char Str255[256];
typedef short (* function_ptr)();

Str255 function_returning_array();          // NO
Str255 *function_returning_ptr_to_array();// OK
short function_returning_function()();      // NO
function_ptr  f_returning_ptr_to_f();       // OK
```

illegal initialization

If the THINK C project is a code resource, desk accessory, or device driver, you cannot initialize an address in global or static memory. Example:

```
static short i;
static short *p = &i;  // NO: Not in application
```

illegal near reference: *'symbol'*

See "link failed" on page 506.

illegal operation on struct
illegal operation on union

You cannot use many C operators on `structs` or `unions`. Example:

```
short function(void)
{
  struct {
    short x;
  } s1, s2, s3;
  short i;

  if (s1 == s2)    // NO: Can't compare structs
    s1++;          // NO: Can't increment structs
  s1 = 0;          // NO: Can't assign short to a
                   //     struct
  s1 = s2;         // OK: struct assignment is
                   //     allowed
  s3 = s1 + s2;    // NO: Can't add structs
  i = s1 - s2;     // NO: Can't add structs
  i = (long)s1;    // NO: Can't cast a struct
  return(s1)       // NO: return does an implicit
}                  //     cast
```

illegal pointer arithmetic

You cannot perform some arithmetic operations on pointers. You cannot assign integers to pointers with the exception of the constant zero. In addition, you cannot compare pointers to integers, again with the exception of the constant zero. In all cases, you can use a cast to force the operation when necessary. Example:

```
void function(void)
{
  short *short_ptr;
  short i;
  char *p1, *p2;
  long result;

  short_ptr = 0x220;              // NO
  short_ptr = 0;                  // OK
  short_ptr = (short *) 0x220;// OK: Address of
                                  //     MemErr
  result = p1 + p2;               // NO: Can't add
                                  //     pointer
  result = p1 / p2;               // NO: Can't
}                                 //     divide
                                  //     pointers
```

illegal preprocessing directive

These are the valid preprocessing directives:

```
#if          #ifdef          #ifndef
#elif        #else           #endif
#include     #define         #undef
#line        #error          #pragma
```

illegal size for bitfield

Bitfield sizes must be less than or equal to the number of bits in the type specifier. Labeled bitfield sizes must be greater than zero. Unlabeled bitfield sizes of 0 force a word alignment. Example:

```
struct bitfields{
  char c : 9;      // NO: chars are 8-bits long
  short i : 17;    // NO: shorts are 16-bits long
  short zero : 0; // NO: Bitfield size must be
                   //     over 0.
  short good : 11;// OK
  short : 0;       // OK: Forces word alignment
  long l : 33;     // NO: longs are 32-bits long
  short z : -4;    // NO: Bitfield size must be
};                 //     over 0.
```

illegal type for bitfield

These are the only types allowed for a bitfield:

```
char                    short int
int                     long int
unsigned char           unsigned short int
unsigned int            unsigned long int
```

You cannot use any other type for a bitfield. Example:

```
struct bitfields{
  char c : 5;           // OK
  unsigned int i : 5;   // OK
  long l : 17;          // OK
  double d : 10;        // NO
  void *p : 8;          // NO
};
```

illegal type for field

A field may not be a function, unless it is a field of a class. Example:

```
struct S {
   short f();  // NO: S is not a class
};

struct T : indirect {
   short f();  // OK: T is a class
};
```

illegal use of class type

You must declare an object as a pointer. Also, you can't use `sizeof` with a class. Example:

```
void function(void)
{
   ClassName   badObject;              // NO
   ClassName   *goodObject;            // OK
   char        a[sizeof(ClassName)];   // NO
}
```

illegal use of void

You cannot use most of the C operators on `void` values. You cannot declare variables of type `void`. Example:

```
void f(void)
{
  short i, j;
  i = (void)j + 5; // NO: Can't add a void.
  i = f() - 3;     // NO: f() returns void.
}
```

immediate value out of range

(Assembly) An immediate operand is outside the range of permissible values for the instruction. Example:

```
asm {
   addq.l#10, d0   ; NO: ADDQ/SUBQ range is 1..8
   lsr.l #10, d0   ; NO: LSR/LSL range is 1..8
   moveq #128, d0 ; NO: MOVEQ range is -128..127
}
```

#include files nested too deeply

For more information on `#pragma once`*, see "Once-only Headers" on page 154*

You tried to compile a file that had `#include` directives nested more than 50 deep. Usually, one of your header files file includes another header file which (eventually) includes the original file again. You can use the `#pragma once` directive to inhibit recursive processing of include files.

incomplete macro call

THINK C reached the end of file before a macro call completed. Example:

```
#define macro(arg1, arg2) (arg1 > arg2)
macro(x,          // NO
```

incorrect base register

(Assembly) You're probably referencing a global through a register other than A5 (for applications) or A4 (for desk accessories, device drivers, or code resources). You don't need to provide a base register to use a global variable in assembly language. Example:

```
short theGlobe;       // a global

void OffBase(void)
{
   asm{
      move  theGlobe(a2), d0  // NO
      move  theGlobe(a5), d0  // OK: But
                              //     unneeded
      move  theGlobe, d0      // OK: Much better
   }
}
```

instruction requires 68020
instruction requires 68881

For more information on writing assembly for the MC68020 or MC6881, see "Writing processor-specific assembly code" on page 252

(Assembly) You're using an MC68020 or MC68881 instruction but MC68020 or MC68881 inline assembly is not enabled. Example:

```
asm 68000 {
   extb.l   d0    ; NO: EXTB.L requires 68020
   ext.w    d0
   ext.l    d0    ; OK: Use EXT.W/EXT.L on 68000
}
```

integer constant required

An integer constant is required, but isn't there. Example:

```
#define aConstant  12
void function(void)
{
  enum {
     shape = -2.4,        // NO: enum values
                          //     must be int.
     size = aConstant     // OK
  } attributes;

  short w[3.14159];       // NO: Constant must
                          //     be an int.
  short z[size];          // OK: Array dimension
}                         //     is an int.
```

integer constant too large

The absolute value of a decimal integer constant must be less than or equal to `4294967295` ($2^{32}-1$). A hexadecimal integer constant must range from `0x0` to `0xffffffff` inclusive. Example:

```
unsigned long a = 4294967296;
                     // NO
unsigned long c = 0x100000000;
                     // NO: Larger than 32-bits
```

invalid 68881 unary operator *'symbol'*

You must declare an MC68881 unary operator to take an extended-precision argument and return an extended precision result. See `math.h` for examples of MC68881 unary operators. Example:

```
// The "8-byte double" option is on.

float acos(float) : 0x1C;                 // NO
double acos(double) : 0x1C;               // NO
long double acos(long double) : 0x1C;     // OK
```

invalid access specifier *'symbol'*

symbol is not `public`, `private`, or `protected`.

invalid addressing mode

(Assembly) You specified an operand to an assembly language instruction which is not a valid MC68000 or MC68020 addressing mode. Example:

```
asm {
   clr.w    (d0)+        ; NO
   move.w   (pc),d0      ; NO
   clr.w    -2(d0,a0)    ; NO
   clr.w    -2(a0,d0)    ; OK
}
```

invalid constructor

You tried to declare a return type for a constructor, or tried to declare a member function with the same name as its class. Example:

```
class C {
public:
   void C();    // NO: Constructor can't
                //     have return type
   C();         // OK
};
```

invalid declaration

A syntax error is in a declaration. Example:

```
short i j;      // NO: Missing ','
```

invalid declaration of 'operator delete'

The prototype for `operator delete` must be like this:

```
void operator delete(void *);
```

invalid declaration of 'operator new'

The prototype for `operator new` must be like this:

```
void *operator new(size_t);
```

If you are not using `stddef.h`, you can use `long` or `unsigned long` in place of `size_t`.

invalid destructor

You tried to declare arguments for a destructor. Example:

```
class C {
public:
   ~C(long);    // NO
   ~C(void);    // OK: But void is unneeded.
   ~C();        // OK: Much better.
};
```

invalid floating-point literal

(Assembly) A quoted floating-point literal is not a legal SANE floating-point number. Example:

```
asm {
   dc.x  "nan0"       ; NO
   dc.x  "nan(0)"     ; OK
}
```

invalid function prototype

For more information on the "Language Extensions" option, see "THINK C Extensions" on page 174

You used a bad function prototype. Usually, the prototype contains only `...` and the "Language Extensions" option is off.

invalid redeclaration of *'symbol'*

You declared an identifier twice, and the second redeclaration is incompatible with the first. Example:

```
extern short x;
long x;                 // NO: x is redeclared
                        //     differently
extern short y;
short y;                // OK: y is redeclared as
                        //     the same type
short z;
short z;                // NO: Can only have one
                        //     defining instance
typedef short IsARose;
IsARose IsARose;        // NO
```

invalid redefinition of macro *'symbol'*

You tried to `#define` an already defined macro with a different definition. You can always redefine a macro to the same thing. If you want to actually change the definition of the macro, you must `#undef` it first. Example:

```
#define NULL 0L
#define NULL 0L              // OK
#define NULL ((void *) 0)    // NO
#undef NULL
#define NULL ((void *) 0)    // OK
```

invalid number

You specified an illegal integer or floating-point number. Example:

```
short x = 0xA9FG;            // NO
float f = 1e*37;             // NO: But +37 or -37
                             //     is OK.
```

invalid register *'symbol'*

The name *symbol* is not one of the permitted register names for a `#pragma parameter` directive. You can use only these registers:

```
A0          A1
D0          D1              D2
```

invalid register list

(Assembly) The register list contains duplicates, or the list is in the wrong order. Example:

```
asm {
   movem a0/a0,-(sp) ;NO
   movem d7-d1,-(sp) ;NO
   movem d1-d7,-(sp) ;OK
}
```

invalid scale factor

(Assembly) In an MC68020 indexed addressing mode, you can apply a scale of only 1, 2, 4, or 8 to the index register. Example:

```
asm 68020 {
   clr.w 20(a0,d0.w*0)          ; NO
   clr.w 20(a0,d0.w*3)          ; NO
   clr.w 20(a0,d0.w*(3+1))      ; OK
   clr.w ([20,a0],d0.w*8,10)    ; OK
}
```

invalid size

(Assembly) In an assembly language instruction, you used a bad size specifier or the wrong size specifier for what you're trying to do. Example:

```
asm {
   move.b   a0,a1     // NO: Can't move bytes to
                      //     address registers.
   move.q   a0,a1     // NO: .q is not a legal
}                     //     size.
```

invalid storage class

You applied incompatible attributes to a data item, implicitly or explicitly. Example:

```
register short aGlobal; // NO: register global
extern auto short y;    // NO: contradictory
                        //     storage class
```

invalid type specification

You used an illegal or ambiguous combination of type keywords when declaring a data item. Example:

```
unsigned double d;          // NO
long struct {short i;} s;   // NO
```

invalid use of class specifier

You tried to declare a member of a class outside of a `class` declaration. Example:

```
class C {
public:
   static short i;
   short j;
   void foo(void);
};

short C::j = 0;          // NO
short C::i = 0;          // OK: Initializes class
                         //     variable.
void C::foo(void);       // NO
void C::foo(void) {...}  // OK
```

invalid use of 'operator delete'

You tried to declare `operator delete` outside of a `class` declaration. To define `operator delete` for a class, you must prefix it with the class name and `::`. Example:

```
class C {
   void operator delete(void *);
};

void *operator delete(void *p) {...}      // NO
void *C::operator delete(void *p) {...}   // OK
```

invalid use of 'operator new'

You tried to declare `operator new` outside of a `class` declaration. To define `operator new` for a class, you must prefix it with the class name and `::`. Example:

```
class C {
   void *operator new(size_t);
};

void *operator new(size_t n) {...}        // NO
void *C::operator new(size_t n) {...}     // OK
```

jump table too big

See "link failed" on page 506.

label *'symbol'* is already defined

The label *symbol* has been defined twice within the function.

last argument to function does not match prototype
last argument to function *'symbol'* does not match prototype

The last argument to a function does not match the type of the argument in the prototype. The prototype may be explicit (it *must* be explicit if the "Require Prototypes" option is set) or inferred (if the "Infer Prototypes" option is set).

link failed

The linker couldn't link your program. Usually, the linker fails due to an undefined symbol or to a multiply defined symbol (a symbol defined more than once in a project). To find out which files contain the offending sym-

bols, use the **Check Link** command in the **Project** menu. The Link Errors window will display the names of the files in parentheses.

- **code segment too big** One or more of your code segments has exceeded the 32K limit. See "Segmentation" on page 97.
- **data segment too big** You declared more than 32K of global and static data in your project. Use memory allocation to create large data structures or turn on the "Far DATA" option in **Set Project Type...** dialog. See "Using more global data with Far DATA" on page 100.
- **illegal near reference: '*symbol*'** A 16-bit reference was made to a variable or function which was not in the first 32K of the data area or jump table, probably from a library. Recompile the library using the Far Code or Far Data options. See "Mixing Near CODE and DATA with Far CODE and DATA" on page 101.
- **jump table too big** If the "Far CODE" option is off, your jump table is over 32K. If the "Far CODE" option is on, your jump table is over 256K. See "Using a larger jump table with Far CODE" on page 100.
- **multiply defined: '*symbol*'** *symbol* was defined more than once.
- **resource too big** For drivers, DRVR + DATA resource is > 32K. For multi-segment drivers, DATA + JUMP > 32K. For code resources, CODE + DATA + JUMP > 32K. For multi-segment code resources, CODE + DATA + JUMP > 32K for segment containing `main()`, CODE > 32K for other segments.
- **undefined: '*symbol*'** You referred to *symbol* without defining it.
- **undefined: *class*::** You didn't define the first method you declared in *class*.
- **undefined: ?*class*::*method*** You declared a method in your class declaration but didn't define it. You might get this message if you defined *method*, but you didn't define the first method you declared in *class*.

lvalue required

An lvalue is an expression that refers to an object in memory that you can store to and examine. Example:

```
short short_f(void);
short *pshort_f(void);
void function(void)
{
  short i;
  short *p = &i;

  // Operand of ++ must be an lvalue
  7++;              // NO
  short_f()++;      // NO
  pshort_f()++;     // NO

  // Left operand of an assignment
  // must be an lvalue.
  short_f() = i;  // NO
  pshort_f() = i; // NO
  *pshort_f() = i;// OK: Produces an lvalue
  (*p)++;         // OK
  (*pshort_f())++;// OK
}
```

macro argument *'symbol'* appears more than once

The formal parameters in a macro definition must appear only once. Example:

```
#define macro(again,again)   (again+7)   // NO
```

memory critical - proceed at your own risk

Now would be a *really* good time to save your files and quit. Before you work on your project again, you may want to increase the size of THINK C's partition with the Finder's **Get Info...** command.

missing ';'

THINK C expected a semicolon, but couldn't find one. Sometimes THINK C cannot determine that a semicolon is missing, and displays the error "syntax error" on page 513.

missing argument name

The argument list in a function definition did not name all the arguments. Example:

```
void foo(short i, short j, float);  // Prototype
void foo(short i, short j, float) { ... }   // NO
void foo(short i, short j, float f) { ... } // OK
```

missing #endif

Every `#if`, `#ifdef`, and `#ifndef` must have a matching `#endif`.

multiply defined: *'symbol'*

See "link failed" on page 506.

no files in project

Your project must contain at least one file for what you're trying to do.

no members defined

You defined a `struct` or `union` without members. Example:

```
struct memberless_struct{ };   // NO
union memberless_union { };    // NO
```

no methods defined

A class definition must have at least one method.

```
struct BadName : indirect {
   short instanceVar;    // NO: No methods
};                       //     declared
```

no such member *'symbol'*

In a `.` or `->` expression, the right operand is not the name of a member of the class or struct for the left operand. Example:

```
void function(void)
{
  struct s1_struct{ short member_1; }s1, *p1;
  struct s2_struct{ short member_2; }s2, *p2;
  s1.non_member;      // NO: non_member is not in
                      //     s1_struct
  p1 -> non_member;   // NO: non_member is not in
                      //     s1_struct
  s1.member_2;        // NO: member_2 is not in
                      //   .   s1_struct
}
```

no such option *'symbol'*

For more information on the `#pragma options` *directive, see "Accessing Option Settings in Your Code" on page 195.*

You used the identifier *symbol* as an argument to `#pragma options` or `__option`, but it is not one of the valid options. Example:

```
#pragma options(!optimize)           // NO
#pragma options(!global_optimizer)   // OK
```

nothing was declared!

For more information on the "Language Extensions" option, see "THINK C Extensions" on page 174

You tried to compile an empty source file. If the "Language Extensions" option is off, source files must contain at least one declaration.

object pointer required

The operand of a `delete` or `__classof` operator has a type which is not a pointer to an object. Example:

```
void function(void)
{
   void *p = new A;   // p is a generic pointer
   A *q = new A;      // q is an object pointer

   delete p;          // NO
   delete q;          // OK
   delete (A*)p;      // OK
}
```

object pointer required for *'symbol'*

Because of the visibility rules for classes and their members, there are other situations where you must use an object pointer. For more information, see Object-Oriented Programming, *Chapter 4, "Using Objects with THINK C."*

You are trying to access a non-static member of a class, and you did not specify an object pointer. Example:

```
class C {
public:
   static short i;
   short j;
   static void f(void);
   void g(void);
};

void foo(C *cp)
{
   C::i = 0;    // OK: 'i' is static.
   C::j = 0;    // NO
   cp->j = 0;   // OK
}

void C::g(void)
{
   C::j = 0;    // OK: 'this' is implied.
}
```

option *'symbol'* can not be adjusted

For more information on the `#pragma options` *directive, see "Accessing Option Settings in Your Code" on page 195.*

Some of the compiler options can not be set or cleared with the `#pragma options` directive during compilation. Usually these are project-wide options, such as `int_4` or `a4_globals`. You can change these options for the entire project from the **Options...** and **Set Project Type...** dialogs.

option *'symbol'* can not be adjusted within a function

With the `#pragma options` directive, you cannot set some compiler options within the body of a function. Move the pragma before the body of the function. Example:

```
void function(void)
{
#pragma options(check_ptrs)    // NO
   ...
}

#pragma options(check_ptrs)    // OK
void function(void)
{
   ...
}
```

out of memory

THINK C ran out of memory. Close open windows to reclaim more memory. If you get this message when you try to run the debugger, try making your application's partition size smaller. See"Memory Considerations" on page 246.

pointer required

You're using an operator that requires a pointer with an expression that's not a pointer. Example:

```
void function(void)
{
  short x, result;
  result = *x;           // NO: x isn't a pointer
  result = x->a_member; // NO: x isn't a pointer
}
```

pointer types do not match

You tried to assign, compare, or subtract incompatible pointers. In THINK C, pointers are more strictly typed than some other C compilers. In assignments and comparisons, the two pointer types must match or be of type `void *`.

When subtracting two pointers, the types must match and not be `void *`. You can cast pointers to force compatibility. Example:

```
void function(void)
{
  char *char_ptr;
  int *int_ptr;
  void *void_ptr;
  int result;
  long difference;
  char_ptr = int_ptr;                          // NO
  char_ptr = (char *)int_ptr;                  // OK
  void_ptr = int_ptr;                          // OK
  int_ptr = void_ptr;                          // OK
  result = (char_ptr == int_ptr);              // NO
  result = ((int *)char_ptr == int_ptr);       // OK
  result = (char_ptr == (void *)int_ptr);      // OK
  result = (void_ptr == int_ptr);              // OK
  difference = char_ptr - int_ptr;             // NO
  difference = char_ptr - (char *)int_ptr;     // OK
}
```

project is not 32-bit compatible

For more information on the Partition box and SIZE flags menu, see "Setting the partition size and SIZE resource flags" on page 101.

You're trying to run your application under THINK C with System 7.0 in 32-bit mode, but you didn't set the 32-Bit Compatible flag in your application. To set that flag, use the SIZE Flags menu in the **Set Project Type...** dialog.

project requires a larger memory partition

You need to increase the project's partition size. To set the partition size, use the Partition box in the **Set Project Type...** dialog.

prototype required
prototype required for function *'symbol'*

For more information on the "Require prototypes" option, see "Type Checking" on page 188.

When the "Require prototypes" option is on, you must provide a function prototype for each of your functions. You may have defined a prototype, but forgot to put a semicolon (`;`) at the end. Also, in a `class` declaration a prototype is required for `operator new` and `operator delete`, regardless of the setting of the "Require prototypes" option.

resource too big

See "link failed" on page 506.

redundant application of 'const'
redundant application of 'pascal'
redundant application of 'volatile'

You repeated one of the qualifiers `const`, `pascal`, or `volatile` in a type declaration.

second argument to function does not match prototype
second argument to function *'symbol'* does not match prototype

For more information on type checking and prototypes, see "Type Checking" on page 188.

The second argument to a function does not match the prototype. The prototype may be explicit or inferred.

static function *'symbol'* referenced but not defined

You declared a function `static` and referred to it, but you did not define it in the same source file. Previous versions of THINK C treated these as if they were declared `extern`, but now it strictly enforces the declaration.

'struct *symbol*' already defined

There is already a `struct` with that tag. Example:

```
typedef struct X { short x; };
typedef struct X { short y; };
```

'struct *symbol*' already in use

The `struct` tag is already used as a `struct` or `union` tag. Example:

```
typedef struct X { short a, b, c; };
typedef union X { short x; double y; };
```

switch value must be integral

A switch expression must be of type `char`, `int`, or `long` or an `unsigned` variant. Example:

```
char *p;
float f;
switch(p){...}      // NO: switch of pointer
switch(f){...}      // NO: switch of float
```

syntax error

There was a syntax error. These are some common errors:

- Too many }'s
- A label without :
- Malformed expressions
- Mismatched parentheses
- A missing semi-colon.

third argument to function does not match prototype
third argument to function *'symbol'* does not match prototype

For more information on type checking and prototypes, see "Type Checking" on page 188.

The third argument to a function does not match the prototype. The prototype may be explicit or inferred.

too many arguments

THINK C allows you to have up to 31 arguments in a function definition. Example:

```
f(arg1, arg2, ..., arg31, arg32)    // one too many
{                                   // args
}
```

too many initializers

The number of initialization values exceeds the expected number of data items specified in the declaration of the data structure. Example:

```
char *directions[4] = {"north", "east",
      "south", "west", "lost" };          // NO
struct { short a,b,c; } x[2] =
      { 1, 2, {3, 4 } };                  // NO
```

too many segments

Applications can have no more than 254 segments. Multi-Segment desk accessories, device drivers, and code resources are limited to 30 segments.

undefined label *'symbol'*

You tried to `goto` the label *symbol*, but you haven't defined that label within the function.

undefined *'symbol'*

See "link failed" on page 506.

unexpected end-of-file

THINK C reached the end of a source file before a C language construct was completed. Example:

```
main(
   // End of file encountered
   // before the close parenthesis.
```

'union *symbol*' already defined

There is already a `union` with that tag. Example:

```
typedef union X { short x; double y; };
typedef union X { short y; double x; };
```

'union *symbol*' already in use

The `union` tag is already used as a `struct` or `union` tag. Example:

```
typedef union X { short x; double y; };
typedef struct X { short a, b, c; };
```

unknown class *'symbol'*

You use the identifier *symbol* as a class name but it is not defined or is not as a class. Example:

```
short C;
void function(void)
{
   void *p = new A;   // NO: A is not defined.
   C::f();            // NO: C is not a class.
}
```

unknown CPU

For more information on writing processor-specific code, see "Writing processor-specific assembly code" on page 252.

(Assembly) You specified a CPU in an `asm` statement that was not one of the recognized CPUs. Example:

```
asm 68010 {     ; NO: Only 68000, 68020, 68030
   rtd   #8     ;     68040, 68881, and 68882
}               ;     are permitted.
```

unknown enumeration *'symbol'*

An enumeration declaration refers to an enumeration tag that you have not defined. Example:

```
enum unknown colors;     // NO
```

unknown instruction

(Assembly) The inline assembler doesn't recognize the mnemonic you used. Usually, you forgot the period in a mnemonic. Example:

```
asm {
   movew    d0,d1     ;NO
   move.w   d0,d1     ;OK
}
```

unknown struct *'symbol'*
unknown union *'symbol'*

You must define a `struct` or `union` before declare an instance of it. However, you can declare a pointer to an undefined `struct` or `union` since

THINK C knows the size of a pointer and doesn't need to know the size of undefined `struct` or `union` is not needed. Example:

```
struct not_previously_defined s;          // NO
struct not_previously_defined *p;         // OK
short i = sizeof(p);                      // OK
short j = sizeof(*p);                     // NO
struct link_list_element {
  struct link_list_element *next;         // OK
  struct link_list_element recursive;     // NO
};
```

unterminated comment

THINK C reached the end of a source file before it reached the end of a comment.

unterminated quote

Either a character constant or a string constant is missing its end quote. Example:

```
void function(void)
{
  long file_type;
  function("hello world);  // NO: missing "
                           //     after world
  file_type = 'TEXT;       // NO: missing '
                           //     after TEXT
  file_type = 'TEXT';      // OK
}
```

value has no members

In a `.` or `->` expression the left operand was not a `struct/union`. Example:

```
void function(void)
{
  short non_struct, *short_ptr;
  struct s1_struct { short member_1; } s1, *p1;

  non_struct.member_1;  // NO: non_struct is not
                        //     a struct */
  short_ptr->member_1;  // NO: short_ptr is not
}                       //     a struct pointer
```

'void' function must not return a value

A function declared `void` cannot return a value. Use a `return` statement without a value. Example:

```
void in_partners_suit(condition)
short condition;
{
   if (condition)
      return;     // OK
   else
      return(1); // NO
}
```

wrong number of arguments to function
wrong number of arguments to function *'symbol'*

You called a function with an explicit or inferred prototype with the wrong number of arguments. To write a function that takes a variable number of arguments, use `...` in your prototype. Example:

```
typedef void (*actionProc)(char *s);
(*actionProc)(aString, anInt);          // NO

void error(char *s);
error("invalid command %s", aString);   // NO

void msg(char *s, ...);
msg("invalid command %s", aString);     // OK
```

wrong number of arguments to macro *'symbol'*

You called a macro with the wrong number of arguments. Example:

```
#define twice(x)   (x+x)
void function(void)
{
  short i;
  i = twice(3,4);     // NO: macro called
}                     //     with 2 args.
```

wrong number of operands

(Assembly) You probably forgot something. Example:

```
asm {
   add.w     d0        ; NO
   add.w     d0,d0     ; OK
}
```

wrong type(s) of operand(s)

(Assembly) Some instructions require operands of a specific type. Example:

```
asm {
   movea    d1,d0     ; NO: MOVEA can move only
}                     ;     into address regs
```

Glossary D

A

A5 The register that points to the application space for the currently running application.

ANSI (The American National Standards Institute) An organization that defines standards for everything from the C programming language to scientific equipment to the sinks in public rest rooms. The ANSI standard for C specifies the syntax and interpretation of the language and includes a set of functions that a compiler should support.

ANSI Library Contains the functions in the library defined in the ANSI standard.

APDA (Apple Programmer's and Developer's Association) Apple's in-house membership organization that distributes technical information and products to developers.

AppleTalk Apple's networking protocol.

application globals The globals that your application defines.

application parameters Information that the system stores in the application space about a running application.

application space Memory that's available for dynamic allocation by applications.

array window In the THINK C Debugger, a window that displays the contents of an array. It is similar to a **data window**.

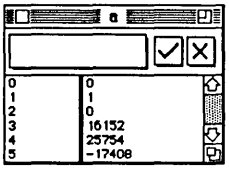


Figure D-1 An array window

assembler An application that compiles assembly language into machine code. See also **inline assembler**.

auto-indenting Describes an editor that indents lines for you automatically. Whenever you start a new line in the THINK C editor, it's indented by the same amount of space as the line above it.

auto-make The part of THINK C that compiles your program for you. Using the information in your project file, it compiles only the files that have been changed since the last compilation.

auto mode The mode in which the THINK C Debugger pauses at breakpoints to update the source and data windows. To enter auto mode, hold down the Option or Command key as you press any of the status panel buttons (except Stop). To exit auto mode, press the Stop button. See also **breakpoint** and **status panel**.

B

breakpoint A statement at which you tell the THINK C Debugger to stop execution when it's running your program. See also **simple breakpoint**, **conditional breakpoint**, and **temporary breakpoint**.

Bundle bit Lets the Finder know that your application has a `BNDL` resource. Set it with the **Set Project Type...** dialog. The `BNDL` resource maps file types to their proper icons.

C

C++ A programming language based on C that includes support for object-oriented programming, operator overloading, and more. THINK C contains object-oriented extensions that are compatible with C++.

call chain See **subroutine call chain**.

callback routine A routine that you pass as an argument to a Macintosh Toolbox routine.

CODE component One of the four components a project has for each source file or library. Contains the executable code generated for the file. Compare with **DATA component**, **JUMP component**, and **STRS component**.

code resource A resource that contains a program's code. Applications have code resources of type `CODE`, and device drivers and desk accessories have code resources of type `DRVR`. Some other types of code resources are `XCMD`, `cdev`, `INIT`, and `WDEF`. Compare with **code segment**.

code segment An individual `CODE` resource, which is part of the code of a Macintosh application. Segments are loaded in and out of memory by the Segment Loader.

conditional breakpoint A breakpoint at which the THINK C Debugger stops only if a condition is true. See also **breakpoint**. Compare with **simple breakpoint** and **temporary breakpoint**.

console A Macintosh window that behaves like a glass TTY or dumb terminal. See also **console package**.

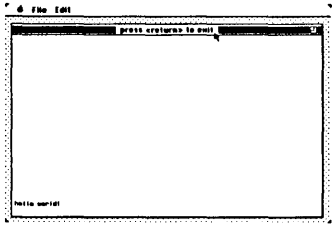


Figure D-2 A console

console package The part of the ANSI library that lets you use simple windows, called consoles, for input and output. If your program uses the streams `STDIN`, `STDOUT`, or `STDERR` and doesn't directly call Toolbox routines, this package creates a console and handles the Macintosh interface for you. This package lets you port UNIX or MS-DOS programs easily. See also **console**.

current function In the THINK C Debugger, the function that THINK C is executing. It's displayed in the lower left corner of the source window. See also **source window**.

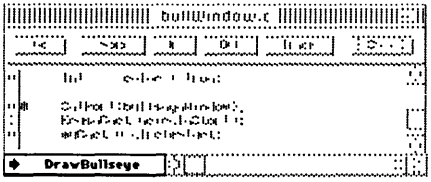


Figure D-3 The name of the current function

current statement In the THINK C Debugger, the statement that the debugger is about to execute. A black arrow, the current statement arrow, appears to the left of it.

current statement arrow In the THINK C Debugger, the arrow that appears to the left of the **current statement**.

D

DA Shell A small program that simulates an environment for your desk accessory. Useful for debugging your desk accessories. Included with THINK C

DATA component One of the four components a project has for each source file or library. Contains the global and static variables for the file. See also **CODE component**, **JUMP component**, and **STRS component**.

data window In the THINK C Debugger, the window you use to display and change the values of your variables.

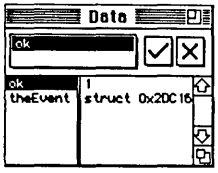


Figure D-4 The data window

deselect button The button with an "X" in the THINK C Debugger's data window. Press it to erase the contents of the entry field. See also **data window** and **entry field**.

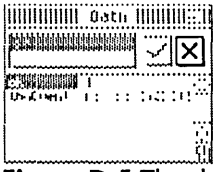


Figure D-5 The deselect button

desk accessory A small program that is implemented as a device driver and can be used at the same time as an application. Its type is `DFIL`, and its creator is DMOV.

Desk Manager The part of the Macintosh Toolbox that handles desk accessories.

device A part of the Macintosh or a piece of external equipment that transfers information into or out of the Macintosh. The most common ones are disk drives, modems, and printers.

device driver A program that controls the exchange of information between an application and a device. See also **device**.

Device Manager The part of the Macintosh Toolbox that supports device input and output.

driver See **device driver**.

E

enter button The button with a check mark in the THINK C Debugger's data window. Press it to enter the contents of the entry field. See also **data window** and **entry field**.

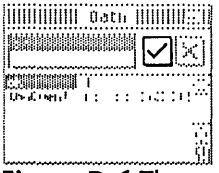


Figure D-6 The enter button

entry field In the THINK C Debugger, the field in the upper left corner of the data window. Here you enter the names of variables you want to view or new values for variables you want to change.

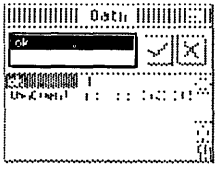


Figure D-7 The entry field

F

Far CODE Code that contains functions that you refer to with 32-bit absolute addresses instead of 16-bit relative addresses. It lets you have a jump table as large as 256K

Far DATA Code that contains global data that you refer to with 32-bit absolute addresses instead of 16-bit relative addresses. It lets you have an unlimited amount of global variables.

File Manager The part of the Macintosh Toolbox that handles file input and output.

function prototype A function declaration that describes the types of the function's arguments. This is a function prototype:

```
int foo (int x, char y);
```

G

global optimizer The part of THINK C that optimizes your code so that it's smaller and faster. These are the types of optimizations it performs: induction variable elimination, common sub-expression elimination, code motion, and register coloring.

Grep A powerful pattern-searching feature in the THINK C editor. Based on the `grep` command found in many UNIX systems.

H

header file A file that contains function prototypes and variable declarations. It does not contain source code to implement functions. You include it in files that use what it declares.

heap The area of memory in which space is dynamically allocated and released on demand, using the Memory Manager. See also **zone size**.

I

#include (v) To put a file in an `#include` statement. For example, to #include the file `stdio.h` in your source file, put this statement in your source file:

```
#include <stdio.h>
```

(adj) Describes any file you use in an #include statement. See also **header file**.

inline assembler The part of THINK C that lets you include assembly language for the MC68000, MC68020, and MC68881 in your C programs.

inline declaration Declares an inline function that lets you embed machine code in a C program. The machine code in the declaration is executed instead of a call to a function.

J

JUMP component One of the four components a project has for each source file or library. Contains the jump table for the file. See also **jump table**. Compare with **CODE component**, **DATA component**, and **STRS component**.

jump table A table that contains one entry for every function in an application. The Segment Loader uses this to find a function in one segment when a function in another segment refers to it.

L

library A collection of compiled functions that you can use in many programs.

Link Errors window When you compile a program, this window displays the names of undefined and multiply defined routines.

List Manager The part of the Macintosh Toolbox that helps you create, display and manipulate lists.

low memory globals Globals stored at the bottom of the Macintosh memory space. They are available to all running applications, but you should manipulate them only with Toolbox routines.

M

MacHeaders A precompiled header file containing the most common declarations used in Macintosh programs. See also **precompiled header file.**

Macintosh Programmer's Workshop (MPW) A software development environment for the Macintosh made by Apple. A fully configured system contains a shell (with a command-line interface much like UNIX), a C compiler, a Pascal compiler, and an assembler.

Macintosh Toolbox The software in the Macintosh ROM that implements the Operating System and the User Interface Toolbox. The Operating System handles low-level operations like I/O and memory management. The User Interface Toolbox helps you implement the standard Macintosh user interface in your application.

Macsbug A low-level debugger from Apple that uses a command-line interface. It can show exactly what's in memory as your program runs, but it can't show you high-level information, like variable names.

marker A placeholder you can put in a source file. You place it with the **Mark...** command, and you go to it by Command-clicking on the title bar of the source window and selecting its name from the pop-up menu.

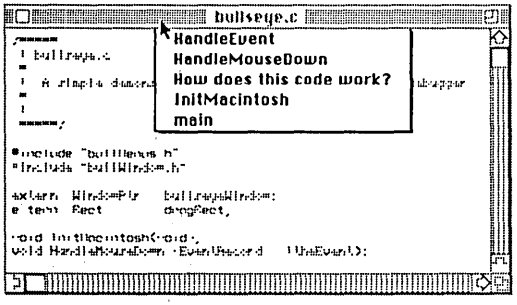


Figure D-8 The marker pop-up menu

MC68000 (1) The microprocessor used in the Macintosh Classic, Plus, Portable, SE, 128K, 512K, and 512K enhanced. (2) A generic term for the microprocessors in the MC68000 family, including the MC68000, MC68020, and MC68030.

MC68020 The microprocessor used in the Macintosh LC and II. In addition to what the MC68000 provides, this provides faster processing, an ex-

tended instruction set, and full 32-bit addressing. With the MC68851 coprocessor, this lets you use **virtual memory**.

MC68030 The microprocessor used in the Macintosh SE/30, IIci, IIcx, IIfx, IIsi, and IIx. In addition to what the MC68020 provides, this lets you use virtual memory without the MC68851. See also **virtual memory**.

MC68851 An coprocessor for the Macintosh II that handles paged memory management and lets you use virtual memory. Sometimes called the *Paged Memory Management Unit* or *PMMU.* See also **virtual memory**.

MC68881 A coprocessor that provides high-speed support for processing scientific computations. Sometimes called the *floating-point unit* or *FPU.*

MPW See **Macintosh Programmer's Workshop**.

multi-segment Describes any application, desk accessory, device, or code resource that contains more than one code segment.

O

.o files The binary files produced by MPW compilers. They can be used as THINK C libraries.

object code The machine language that the THINK C compiler generates from your source files. THINK C stores it in your project.

optimizer See **global optimizer**.

P

partition size The amount of memory an application gets when running under MultiFinder. You set it in the **Get Info...** dialog in the Finder. Also called *Application Memory Size.*

Pascal string A string that begins with `"\p"` or `"\P"`, like this:

```
"\pThis is a Pasal string"
```

THINK C replaces the `\p` with a byte containing the length of the string. Use with Pascal routines, such as Toolbox routines.

PMMU See **MC68851**.

pointer A variable that contains the address of something in memory, like another variable or a function.

precompiled header file A header file in a format THINK C can use readily, making it faster to load than other headers. Must contain only declarations and preprocessor symbols. See also **header file**.

Printing Manager The part of the Macintosh Toolbox that lets programs print on any kind of printer with the same interface.

profiler The part of THINK C that collects timing statistics about your functions. To use it, turn on the Profile option in the Code Generation section of the **Options...** dialog.

program Anything you can compile with THINK C: an application, desk accessory, device driver, or code resource.

prototype See **function prototype**.

project A file that keeps track of all the files in your program. It contains the object code for your files, maintains the dependencies between them, and remembers what needs to be recompiled. This is the heart of the THINK C development environment.

project prefix Text that THINK C includes in all the source files in your project. It's as if you put the text at the beginning of all your source files. You set in this prefix in the Prefix page of the **Options...** dialog.

project tree The folder your project is in and all the folders in it. Keep the files you use in only one project here.

project window The window that displays the files in your project. Here you can select files to edit and debug. See also **project**.

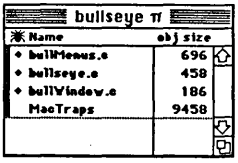


Figure D-9 A project window

Q

QuickDraw The part of the Macintosh Toolbox that performs all the graphic operations on a Macintosh screen.

R

ResEdit An application that lets you interactively create and edit resources. Included with THINK C.

resource A piece of data stored in the resource fork of a Macintosh file. Common resources are windows, menus, fonts, and executable code. See also **resource fork**.

resource fork One of the two forks in any Macintosh file. It contains resources, such as menus, fonts, icons, and executable code. To access it, use the Resource Manager. See also **resource**. Compare with the **data fork**.

resource file (1) A file that contains the resources your project uses. When you run a program in the environment, THINK C opens this file so you can use its resources. When you build your project, THINK C copies the resource file into your finished program. (2) Another name for the **resource fork** of a Macintosh file.

S

SANE See **Standard Apple Numerics Environment**.

SADeRez An application that decompiles the resources in a file and creates a resource description file. Included with THINK C.

SARez An application that compiles a resource description file to create resources. Included with THINK C.

segment See **code segment**.

Segment Loader The part of the Macintosh Toolbox that loads the code of an application into memory, either as a single unit or divided into dynamically loaded. See also **code segment**.

selected statement In the THINK C Debugger, the statement you selected in the source window by clicking on it. It is highlighted. If you don't click on a statement, the current statement is the selected statement. Compare with **current statement**. See also **source window**.

Serial Driver A device driver that controls communication, via serial ports, between an application and serial devices.

simple breakpoint A breakpoint at which the THINK C Debugger always stops. See also **breakpoint**. Compare with **conditional breakpoint** and **temporary breakpoint**.

smart linking A technique THINK C uses to remove as much unused code from your final program as possible. THINK C removes code for all unused functions.

source files The text files that contain your source code.

source window In the THINK C Debugger, the window that contains the source code being executed and the **status panel**.

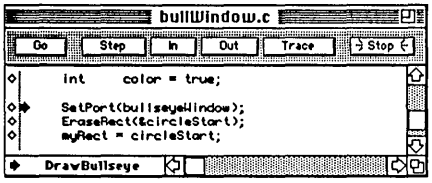


Figure D-10 The source window

stack (1) Any area of memory that is allocated and deallocated in a last-in first-out order, like a stack of trays in a cafeteria. (2) A specific stack in the MC68000 that contains information on each routine in the subroutine call chain. Each time you call a routine, a stack frame is added that contains the routine's parameters, variables, and return address. When you return from a routine, its frame is removed from the stack.

stack frame A frame in the MC68000 stack that contains a routine's parameters variables, and return address.

stack size The maximum size of the stack in the MC68000. By default, this is 16K.

Standard Apple Numerics Environment (SANE) A system for performing floating-point arithmetic that Apple computers use. It meets all requirements for extended-precision, floating-point arithmetic as prescribed by IEEE Standard 754 and ensures that all floating-point operations are performed consistently and return the most accurate results possible.

statement marker In the THINK C Debugger, the diamond to the left of a line of source code. It shows that the line generates code.

status panel In the THINK C Debugger, the top of the source window. It contains six buttons that let you control the execution of your program.

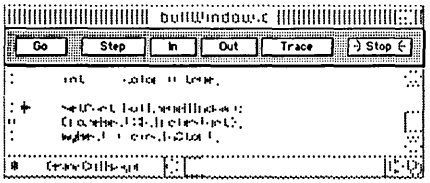


Figure D-11 The status panel

STRS component One of the four components a project has for each source file or library. If you use the Separate STRS option, this contains the string literals and floating-point constants for the file. Otherwise, they're in the DATA component and this component doesn't exist. See also **CODE component**, **DATA component**, and **JUMP component**.

string literal A sequence of characters enclosed by quotes.

struct window In the THINK C Debugger, a window that displays the contents of a `struct`. It is similar to a **data window**.

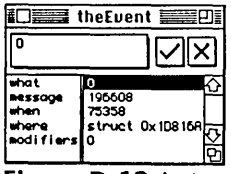


Figure D-12 A struct window

subroutine call chain The list of functions called to get to the current function. In the THINK C debugger, click on the current function indicator to see this. See also **current function**.

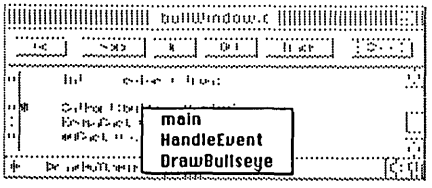


Figure D-13 A subroutine call chain

T

temporary breakpoint Lets you set a breakpoint and start execution in one step. The next time the THINK C Debugger comes to it, the debugger stops and clears it. See also **breakpoint**. Compare with **conditional breakpoint** and **simple breakpoint**.

THINK C tree The `THINK C Folder` and all the folders in it. If you use a file in more than one project, keep it in here.

TMON A low-level debugger that uses a graphical interface. It can show exactly what's in memory as your program runs, but it can't show high-level information, like variable names.

Toolbox See **Macintosh Toolbox**.

Toolbox routine Any routine in the Macintosh ROM.

trap intercept routine A routine that executes instead of a trap routine that implements a Toolbox routine.

U

UNIX A popular operating system for mainframe computers created by AT&T's Bell Labs. Bell Labs created C to write UNIX.

unix Library Contains many functions found in C compilers for UNIX that aren't in the ANSI standard.

V

vertical retrace task A task that executes during the vertical retrace interval, which occurs sixty times a second.

virtual memory Space on your hard disk that you can use as though it were extra RAM.

Z

zone See **heap**.

zone size The size of your application's heap.

Index

Entries in **bold face** refer to menu commands. Entries in `typewriter face` refer to functions, methods, variables, keywords, or files.

Symbols

A

B

C

D

E

F

G

H

I

J

N

O

P

Q

R

S

T

U

V

W

X

Z

Symantec Corporation
10201 Torre Avenue
Cupertino, CA 95014-2132
408/253-9600

05-30-00029
LSC019 4/91