SYMANTEC

THINK's LightspeedC™

The Professional's Choice

USER'S MANUAL

For Macintosh Plus,
Macintosh SE, and
Macintosh II computers

Credits

Software: Michael Kahl

User's Manual: Philip Borenstein

Product Manager: Diana Bury

Special thanks to Andrew Singer.

THINK Technologies Division
135 South Road
Bedford, MA 01730, USA
(617) 275-4800

Contents

THINK's LightspeedC

PART ONE

Getting Started

Welcome
1

Introduction

Welcome to THINK's LightspeedC. 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

If you need to get started quickly, read this chapter, the next chapter, and one of the tutorials.

If you're an experienced THINK C user

If you already use THINK's LightspeedC, you'll be pleased with the source level debugger and other improvements. Read "Notes for Experienced Users" at the end of this chapter.

Topics covered in this chapter:

- What is THINK's LightspeedC
- What you need
- What's in the package
- What's in the manual
- What you should know
- Notes for experienced users

What is THINK's LightspeedC

THINK's LightspeedC is a unique development environment for the Macintosh. It features a very fast compiler, a faster linker, an integrated 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 an #include 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 drives, and any kind of code resource. The standard C libraries include the standard I/O functions as well as 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 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 how you wrote it. No more translating assembly language back into C. The debugger lets you set breakpoints, step through your code, examine variables, and change their values while your program is running. And because the debugger runs under MultiFinder, you can edit your source files while you're debugging.

What You Need

THINK C works best when you have at least 2 megabytes (Mb) of RAM and a hard disk. It will also run with 1Mb of RAM and two 800K floppy drives. With only 1Mb you won't be able to take advantage of MultiFinder, and you won't be able to use the debugger. With a floppy based system, you won't be able to write very large programs.

How much RAM?

You can run THINK C on a Macintosh Plus, Macintosh SE, or Macintosh II. 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 1Mb 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.

Although you can use THINK C with two 800K floppy drives, it works much better when you use a hard disk.

Which System/Finder?

Use the latest System and Finder provided by Apple. At press time, this is System Tools 6.0 (System 6.0/Finder 6.1). THINK C requires at least System Tools 5.0 (System 4.2/Finder 6.0).

THINK C is designed to work best under MultiFinder. If you're using a standard Mac Plus with 1Mb RAM, however, you're better off *not* using MultiFinder.

What's in the Package

Your THINK C package consists of two double sided floppies, this manual, and the *Standard Libraries Reference.*

What's in the Manual

This manual is organized in five sections: Getting Started, Learning THINK C, Using THINK C, Reference, and the Appendices. Each chapter beings 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. The first one, "Hello World" shows you how to write a minimal program that uses the standard C libraries and introduces you to the basics of using THINK C.

The second tutorial, "MiniEdit," shows you 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.

The third tutorial, "Bullseye," shows you how to use THINK C's source level debugger. Read this chapter even if you already know how to use THINK C.

Using THINK C

This section contains eight chapters that describe the different components of THINK's LightspeedC.

Overview, Chapter 6, describes how THINK C works.

The Project, Chapter 7, describes 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, Chapter 8, describes 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, Chapter 9, tells you why you should follow the installation instructions in Chapter 2. This chapter describes how THINK C looks for files on your disk and how it names files.

The Compiler, Chapter 10, describes how THINK C compiles your source files. It also tells you how to call the Macintosh Toolbox

routines, how to generate code for the 68881 floating point coprocessor, how to use function prototypes, and how to port code from Unix machines.

The Debugger. Chapter 11, describes 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, Chapter 12, describes 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, Chapter 13, tells you 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.

THINK C Menus, Chapter 14, describes the THINK C menu commands.

Debugger Menus, Chapter 15, describes the source level debugger's menu commands.

Language Reference, Chapter 16, is a supplement to the C Language Reference (Appendix A) of Kernighan and Ritchie's *The C Programming Language*.

Appendices

This section contains appendices that describe the code profiler, some tips and troubleshooting suggestions, a list of error messages (with explanations), the RMaker resource compiler reference, and the index.

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 in ROM. The Macintosh ROM actually consists 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 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 to much on these aspects of programming.

If you're learning C from a book, or if you're using THINK C to do coursework, 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 the first edition of Kernighan & Ritchie's *The C Programming Language* (Prentice Hall) and Harbison & Steele's *C: A Reference Manual* (Prentice Hall). The second edition of *The C Programming Language* is an update that incorporates the proposed ANSI standard. These books assume that you're already an experienced programmer.

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 released 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.

There are now several good books that introduce you to programming the Macintosh and teach you some of the finer points of using the Macintosh Toolbox. No matter which books you choose to help you get started, *Inside Macintosh* is indispensable.

Inside Macintosh Volumes I-V (Addison-Wesley) is the official reference that describes the more than 600 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. At five 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.

Stephen Chernicoff's two volume set, *Macintosh Revealed* (Hayden Books), is a 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 book are written in Pascal, but they're not too difficult to translate to THINK C.

Scott Knaster, formerly of Apple Technical Support, is the author of two books about Macintosh programming. The first, *How to Write Macintosh Software* (Hayden Books), teaches you what's going on inside the Toolbox. This books also 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 writing Macintosh programs. It also contains information about the Macintosh II and the Mac SE. These books are more technical than *Macintosh Revealed* and are loaded with pictures, diagrams, and examples (as well as some awful jokes).

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, but most of the C examples are written in THINK C. (All of the programs described in the magazine are available on disk.)

Apple Programmer's and Developer's Association

The Apple Programmer's and Developer's Association (APDA) is an Apple-sponsored 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. Membership costs $20 per year.

For information about membership and products, contact APDA directly:

Apple Programmer's and Developers Association (APDA)
290 SW 43rd Street
Renton, WA 98055
(206) 251-6548

CompuServe

Symantec has a forum on CompuServe specifically for THINK C users. Simply type `GO THINK` at any ! prompt. You'll find discussions here about programming in general and THINK C in particular. The data libraries contain utilities as well as sources for some of the programs. When upgrades are ready, they're usually posted here first.

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

Notes for Experienced Users

If you've used THINK's LightspeedC before, you'll be pleased with the new features of this new release. This section describes the changes and enhancements.

Compatibility

THINK C version 3.0 reads and converts version 2.x projects automatically. Be sure to use the newer version of `MacTraps` or any other supplied libraries.

> **Note:** It's not enough to copy `MacTraps` to your disk. You also need to reload it into your project. Use the **Load Project** command in the **Source** menu, or choose **Make...** from the **Source** menu and click on the Use Disk button to mark `MacTraps` for reloading.

You will not be able to use projects from version 1.x. Create a new project, and add your existing source files instead.

String literals and floating point constants no longer live in the STRS component by default. Instead, THINK C places them in the DATA component. You can choose to have separate STRS in the **Set Project Type...** dialog.

If you use string literals or floating point constants in code resources, make sure you set up register `A4` correctly or your project may not work. See "Building Code Resources" in Chapter 7 to learn how to set up register `A4`.

New features

The source level debugger is the biggest new feature. Read Chapter 3, the Bullseye tutorial, to learn how to use the debugger. Read Chapter 11 to learn more about the source level debugger.

Precompiled headers are a major new feature. Precompiled headers work like regular #include files, but since they're in a form THINK C can use readily, they load faster. A standard precompiled header, `MacHeaders`, contains most of the declarations used in writing Macintosh programs. This means that you don't have to #include files like `QuickDraw.h` in every file. Read "Precompiled Headers" in Chapter 10. Your projects will build faster if you remove the #include statements.

The **Set Project Type...** dialog contains many new enhancements. You can set the type and creator for all files, MultiFinder attributes for applications, ask THINK C to build multi-segment desk accessories and device drivers, specify resource attributes for code resources, and more. Look up the **Set Project Type...** command in Chapter 14, THINK C Menus to learn the details for each type of project.

While you're in Chapter 14, look at the new options available in the **Options...** dialog. You can now ask for 68881 and 68020 code generation, choose not to have `MacHeaders` included automatically, and set options for the source level debugger.

Installing THINK C 2

Introduction

This chapter tells you how to install THINK's LightspeedC on your system.

Topics covered in this chapter:

- Installing on a hard disk system
- Installing on a floppy system

Installing THINK C on a Hard Disk System

This section 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.

Installation summary

First you'll create a development folder. The development folder will contain a folder for THINK C, the #include files, and the libraries. Then you'll create folders for each of your projects. The project folders will be in the development folder, but not in the THINK C folder.

The picture at the end of this chapter shows you what this disk layout looks like. You can use the picture to set up your disk, or you follow these directions.

Installation instructions

Start at the Finder and create a new folder. Name it `Development`.

Open the `Development` folder and create a new folder in it. Call the new folder `THINK C Folder`.

Now, copy all the files from disk `THINK C 1` to the `THINK C Folder`. At least these files should be in the `THINK C Folder`: (Note that for THINK C to work correctly, these files must be in the same folder.)

- THINK C (the application)
- MacHeaders
- Mac #includes folder
- Mac Libraries folder
- DAShell
- THINK C Debugger

Next, copy the `Libraries` folder from `THINK C 2` to the THINK C Folder. This folder contains the standard C libraries, the standard #include files, and the library sources. If you're running short on disk space, you can get rid of the Library Sources folder. They'll be on your original disk if you ever need them.

That's all there is to it. Be sure to read Chapter 9 to learn more about how THINK C treats files and folders.

Installing on a Floppy System

Although THINK C works best when you use a hard disk, it's possible to use it if you have two 800K floppy drives. If you use floppies, you probably like to put your application and a System folder on one floppy and all of your data files on another. Unfortunately, the THINK C system won't fit on the same floppy as the System. To use THINK C from floppies, your "data disk" will be your System disk, and THINK C will be on another floppy.

Making the System disk

First, create your System disk; just drag the System folder from one of your original Macintosh System disks to a blank floppy.

Next, use the Font/DA Mover to remove all the fonts except the ones needed by the system. Just select all the font names, and click on the Remove button. You'll get a warning dialog telling you the system fonts won't be removed. While you're still in the Font/DA Mover, remove all but one of the Desk Accessories. (The Calculator seems to be the smallest one).

Now remove the print drivers, the Control Panel files (General, Mouse, Sound, etc.). Leave the System, Finder, MultiFinder, and DA Handler.

You'll use this System floppy as your data disk. Use this disk to store your projects.

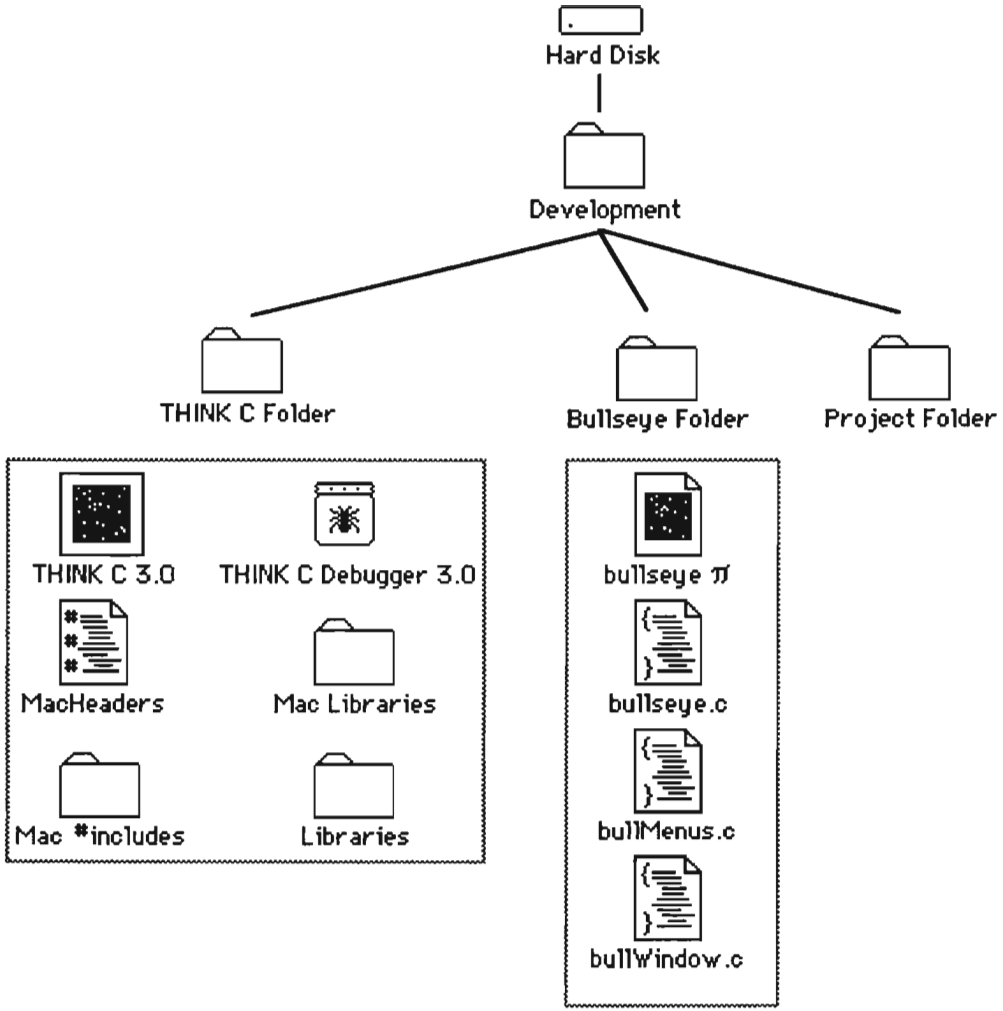
Making the THINK C disk

Now you're ready to make the THINK C disk. This disk will contain the THINK C application, the libraries, and the #include files.

First, copy all the files from the `THINK C 1` to a blank floppy. If you are planning on writing programs that use the standard C libraries, copy the `Libraries` folder from the `THINK C 2` disk to your floppy.

Disk Layout Diagram

This diagram on the next page 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.


Hard Disk
Development
THINK C Folder
Bullseye Folder
Project Folder
THINK C 3.0
THINK C Debugger 3.0
MacHeaders
Mac Libraries
Mac #includes
Libraries
bullseye π
bullseye.c
bullMenus.c
bullWindow.c

THINK's LightspeedC

PART TWO

Learning THINK C

Tutorial: Hello World 3

Introduction

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

Before you begin

Be sure you followed the instructions in Chapter 2 to put THINK C on your disk. The names of your folders may not match the pictures in this chapter. It's all right as long as you remember where you put your files.

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 folders contain the files `stdio` (not to be confused with the file `stdio.h`) and `MacTraps`.

Topics covered in this chapter:

- Creating a project
- Creating the source file
- Compiling the source file
- Adding the libraries
- Running the project
- Creating an application

Creating the Project

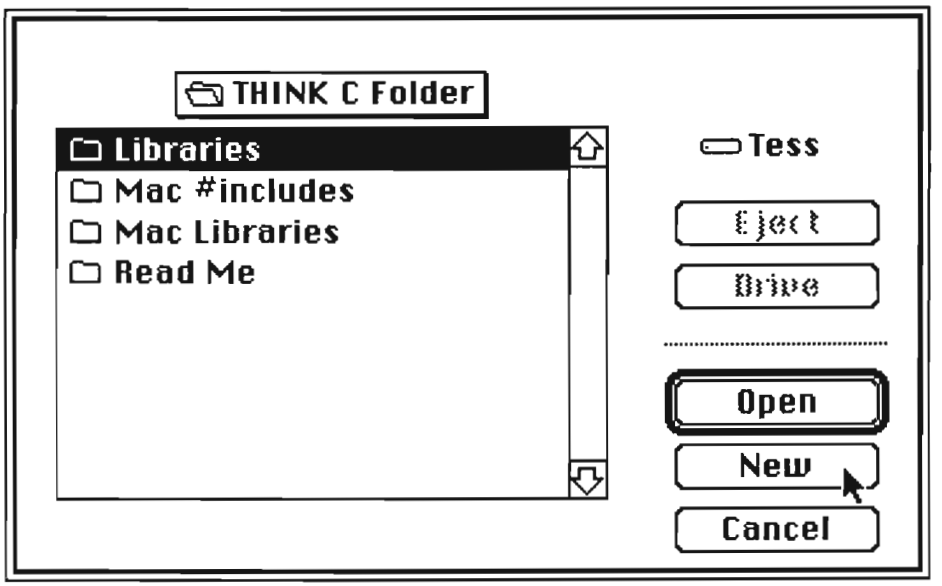
The first thing you need to do is create a folder for your project in the `Development` folder. This is the folder you'll use for all your development work.

Create a folder called `Hello Folder` in the `Development` 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 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.



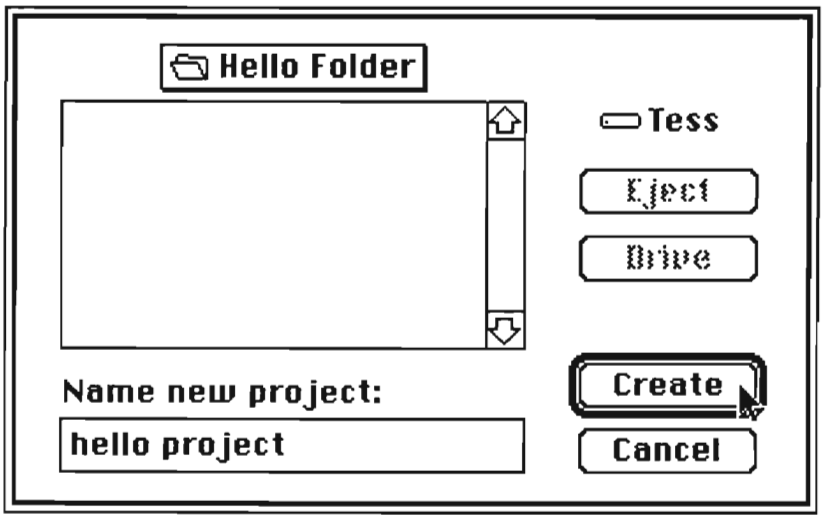
Since you're creating a new project, click on the New button.

You'll see another dialog box, one that lets you create projects.

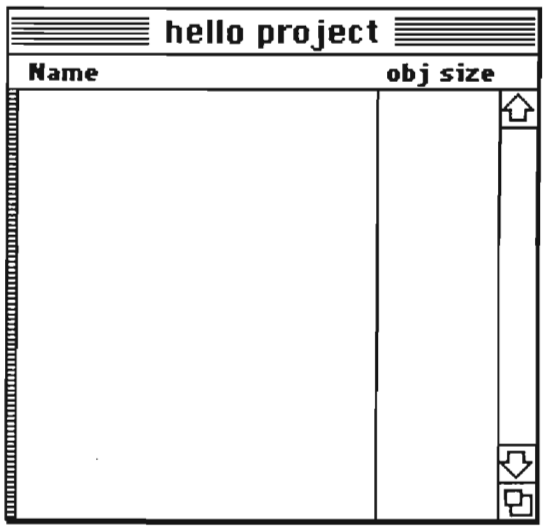
Move back to the `Hello Folder` you just created.

> **Note:** It's very important that you move to the `Hello Folder`.

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



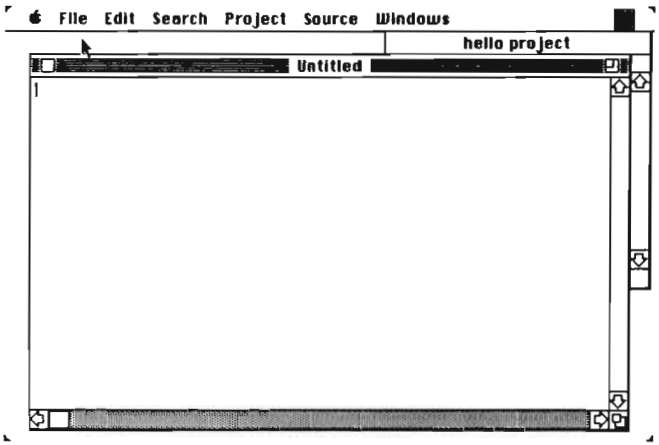
THINK C creates a new project document on disk and displays a project window:



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.



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

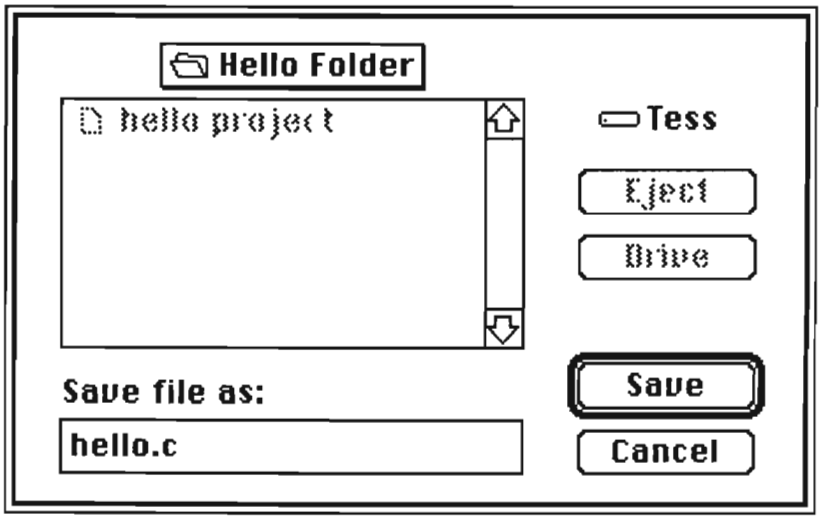
```
/*****
 * hello.c
 *
 * The hello world program for THINK C
 *
 *****/

#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. Triple click to select an entire line. If you have a keyboard with arrow keys, you can use them to move around your file.

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 scrollbar at the bottom of the window to see any text that goes past the right edge.

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

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.

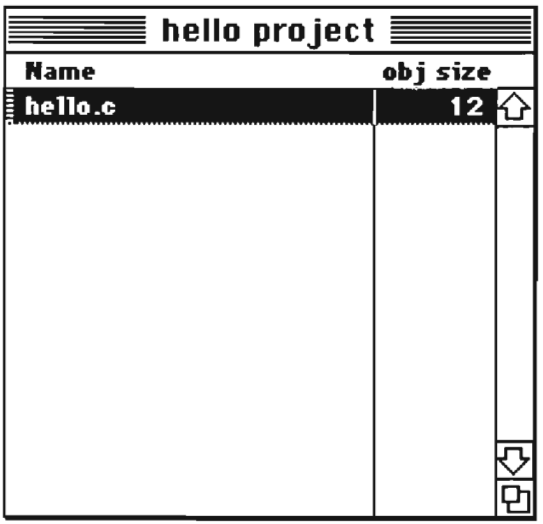


THINK C will only compile files that end in `.c`, but you can edit any text file with the THINK C editor.

Compiling the Source 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:



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 thing you can do is forget 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 #include'd file" like this one:

it means that THINK C wasn't able to find the #include file `stdio.h`. THINK C might not be able to 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 to make sure you installed THINK C correctly. Once you're sure everything is OK, start again from the beginning of this chapter.

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

Adding the Libraries

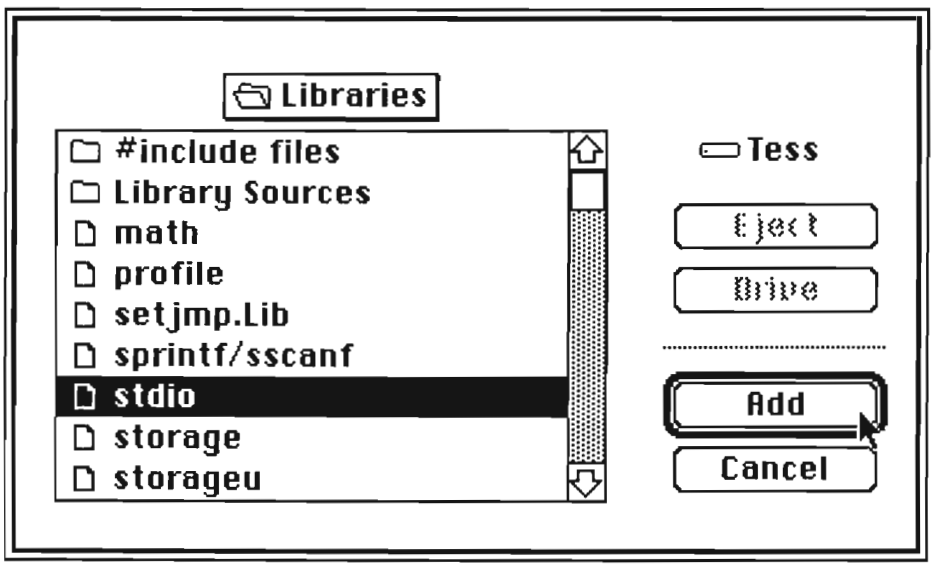
If you were to try to run your program now, you'd get linking errors because the project doesn't know where the `printf()` function is defined.

`Printf()` is a standard C input/output function, so it's in the `stdio` library. The next thing you need to do is to add the `stdio` library to your project.

> **Note:** It's easy to get confused. There is a file called `stdio.h` and another called `stdio`. Just remember that `stdio.h` is a text file that contains the definitions of constants and data structures used in the `stdio` library. You #include `stdio.h` only when you're using the `stdio` library.

To add the `stdio` library to your project, select **Add...** from the **Source** menu.

When you get the standard file dialog box, open the folder called `Libraries`. This folder contains all the libraries for Unix compatibility, including the `stdio` library. Select `stdio`, and click on the Add button. (If you're having trouble finding `stdio`, be sure you're not looking for `stdio.h`.)

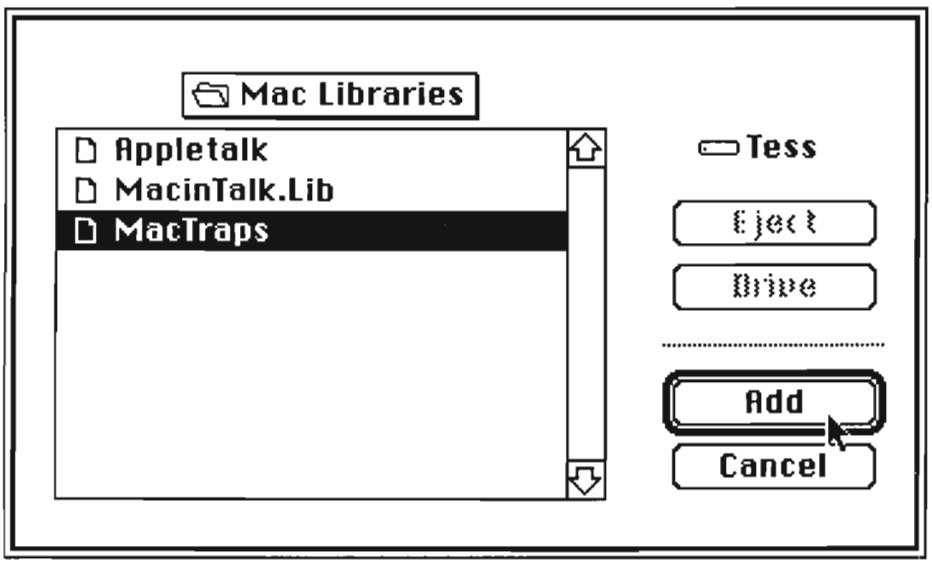


THINK C adds the name `stdio` to the project window and then puts up the standard file dialog box again. Don't click on any button just yet.

The `stdio` library is implemented in terms of Macintosh Toolbox routines. The `printf()` function, for instance, uses Toolbox routines to open a window and to draw a string to it. To make sure that `printf()` knows about the Toolbox routines, you'll need to add the `MacTraps` library to your project as well.

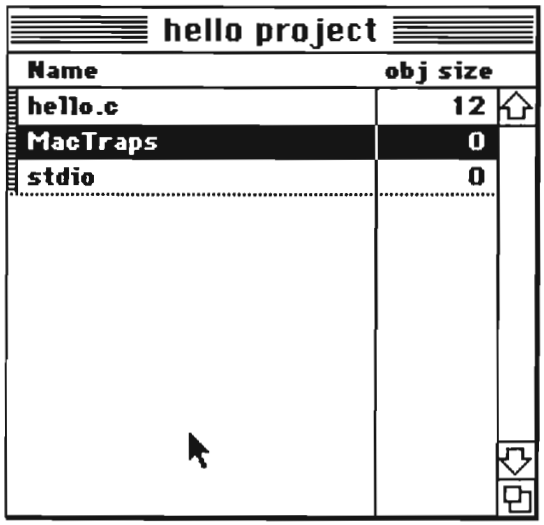
Since the standard file dialog is still up, use it to move to the `Mac Libraries` folder. This folder contains the libraries for interfacing to the Macintosh Toolbox.

Select `MacTraps` and then click on the Add button.



These are all the libraries you'll need for this project, so click on the Cancel button when the standard file dialog reappears after you add `MacTraps`.

Your project window should look like this:



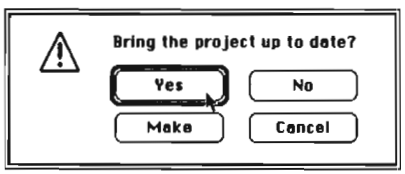
The object size for the libraries reads zero because when THINK C adds a library to your project, it doesn't load the code for it right away. This lets you add a whole set of libraries without waiting for them to load.

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

Running the Project

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.

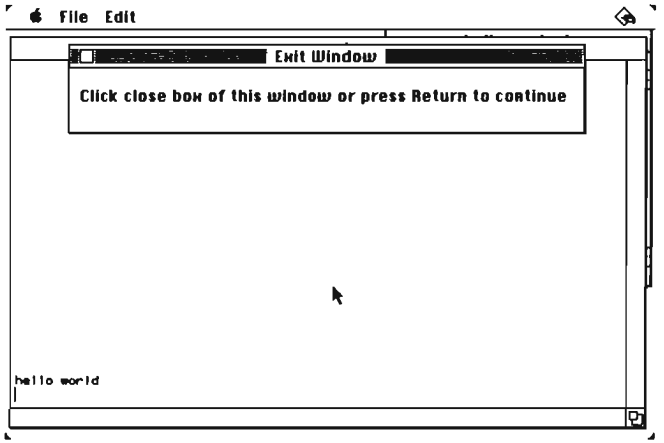
THINK C notices that the libraries need to be loaded, so it puts up a dialog box asking you if you want to bring the project up to date:



Click on the Yes button. THINK C goes to disk to load the code for the libraries. It may take THINK C a little time to load the libraries. Once they're loaded into the project, though, THINK C doesn't need to load them 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.

This program uses the `stdio` library, so all output from `printf()` calls goes to a window called console. The console window emulates a generic terminal screen. You'll see the "hello world" string at the bottom of this window.

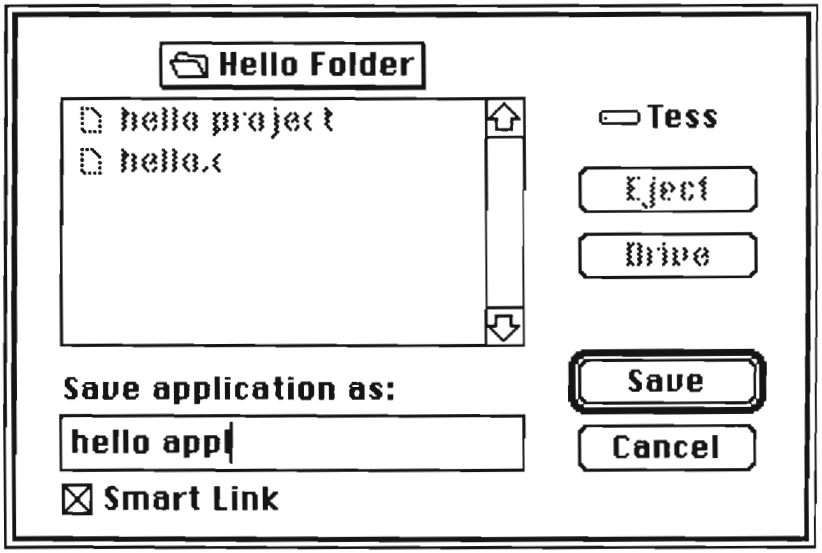


To exit the program, press Return or click on the close box of the Exit window. Like the console window, the exit window appears only when you use the `stdio` library.

Creating the Application

As you develop a large application, you'll 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.

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, quit THINK C to run your application. Use the **Quit** command in the **File** menu. If you're using MultiFinder, 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 and follow the tutorial there.

Tutorial: MiniEdit 4

Introduction

This chapter shows you how to use the more advanced features of THINK's LightspeedC. 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.

Copy the folder `MiniEdit Folder` from disk `THINK C 2` to your THINK C folder. This folder contains all the files you'll need to follow this example.

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`.

Topics covered in this chapter:

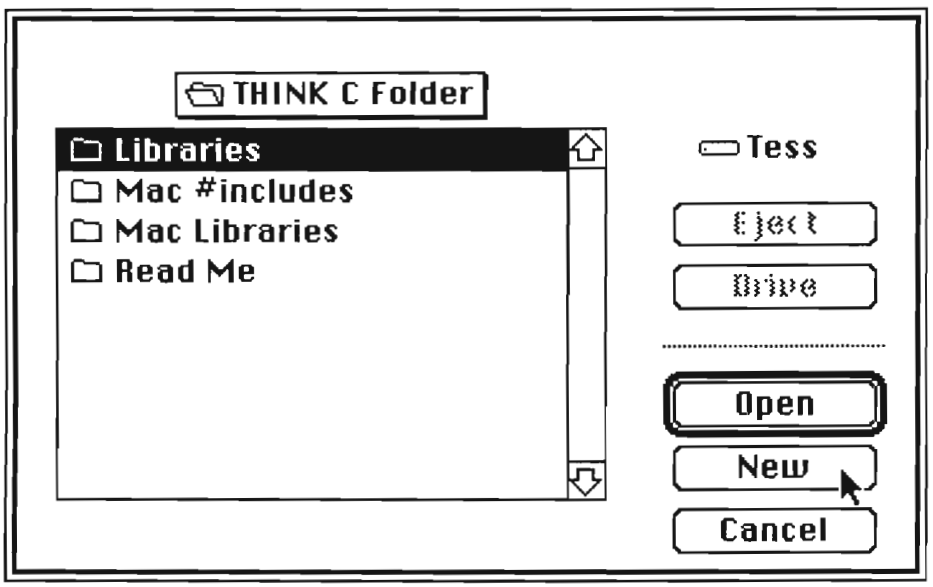
- Creating a project
- Adding the source files and libraries
- Compiling and running the project
- Fixing a bug
- Running the project again
- Creating an application
- Using a resource file
- Finishing up

Creating the Project

Make sure you copy the entire `MiniEdit Folder` from disk `THINK C 2`. This folder contains the source files you need to create the MiniEdit application as well as the application's resource file.

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.

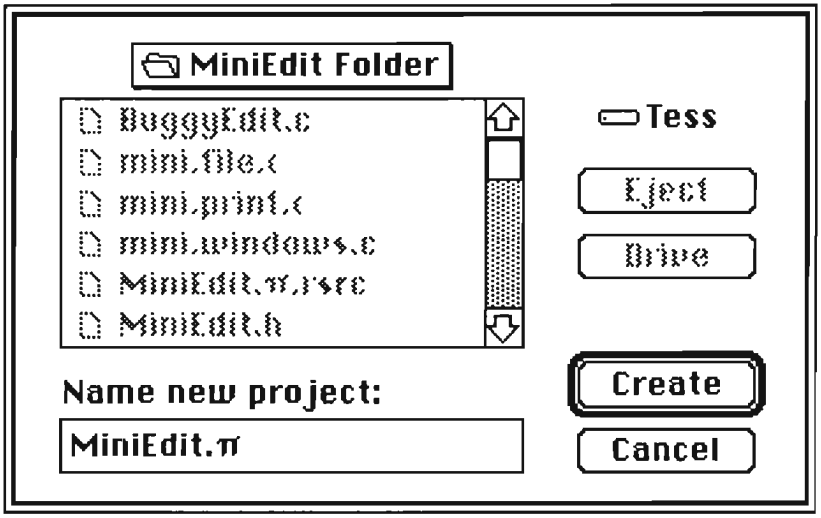
Open the THINK C Folder, and double click on the THINK C icon to begin. You'll see a dialog box that asks you to open a project.



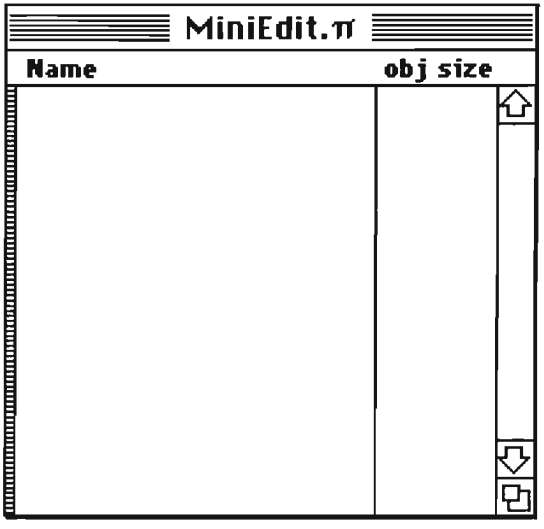
Since you're creating a new project, click on the New button.

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`. (To make a π, type Option-p.)

> **Note:** It's very important to move back to the `MiniEdit Folder`.



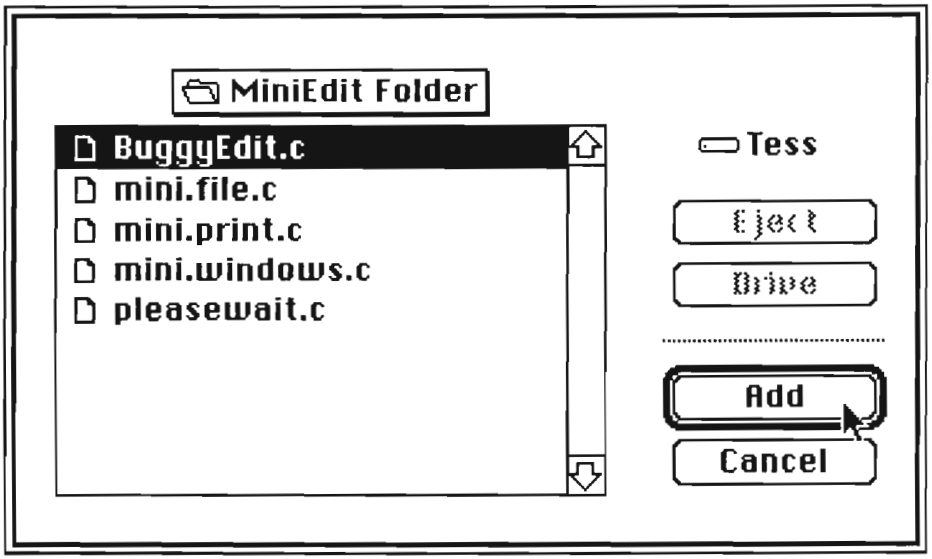
Click on the Create button. THINK C creates a project document on disk, and displays an empty project window.



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` you copied from disk `THINK C 2`.

Adding the Source Files and Libraries

Select **Add...** from the **Source** menu. You'll see a standard file dialog that lets you add source files and libraries.

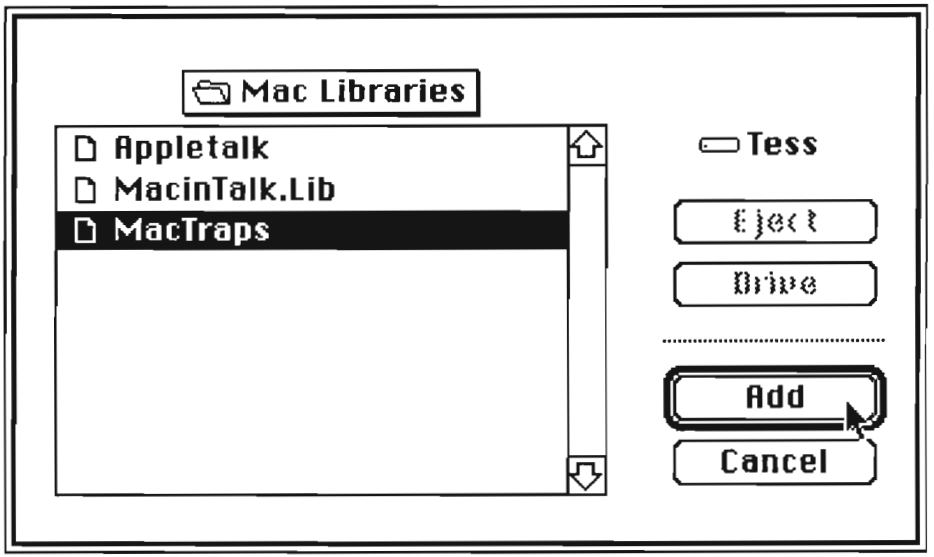


Double click on the first file in the file list, `BuggyEdit.c`. THINK C adds the file name to the project window and displays the standard file dialog again. Add all of the source files in the `MiniEdit Folder` to the project:

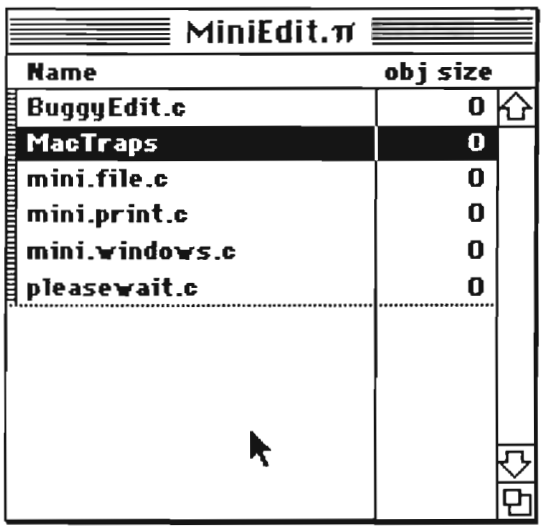
```
BuggyEdit.c
mini.file.c
mini.print.c
mini.windows.c
pleasewait.c
```

When you've added the last file, `pleasewait.c`, do not click on the Cancel button.

Move to the Mac Libraries folder, and add the `MacTraps` library.



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 this:



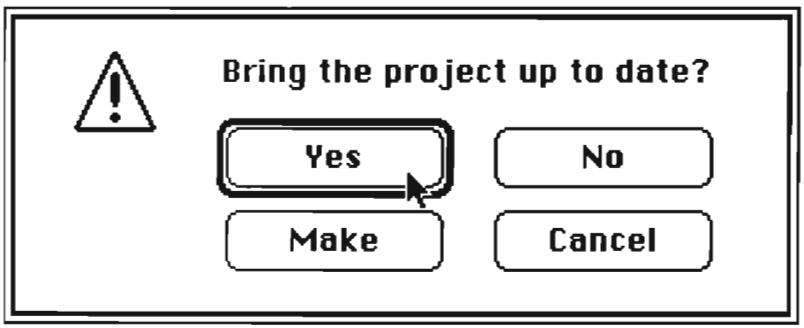
The **obj size** column displays the object size in bytes for each file. The sizes are all zero because you haven't compiled any files or loaded the `MacTraps` library.

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:



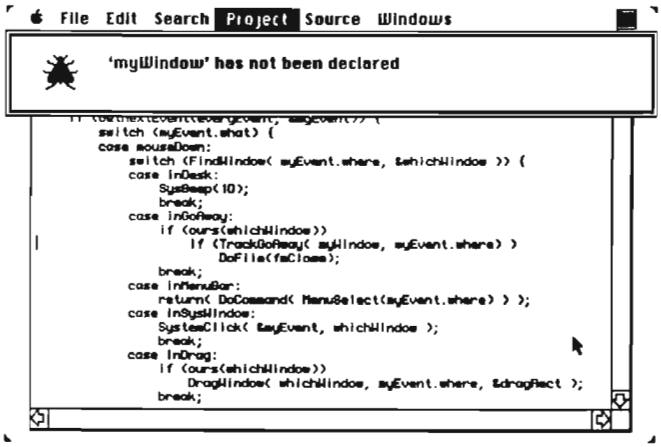
Click on the Yes button.

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.

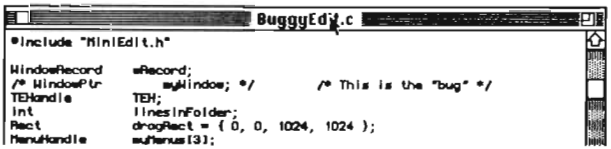
Fixing a Bug

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. The insertion point is at the beginning of the line that contains the error. In this example, THINK C complains that a variable hasn't been defined.



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

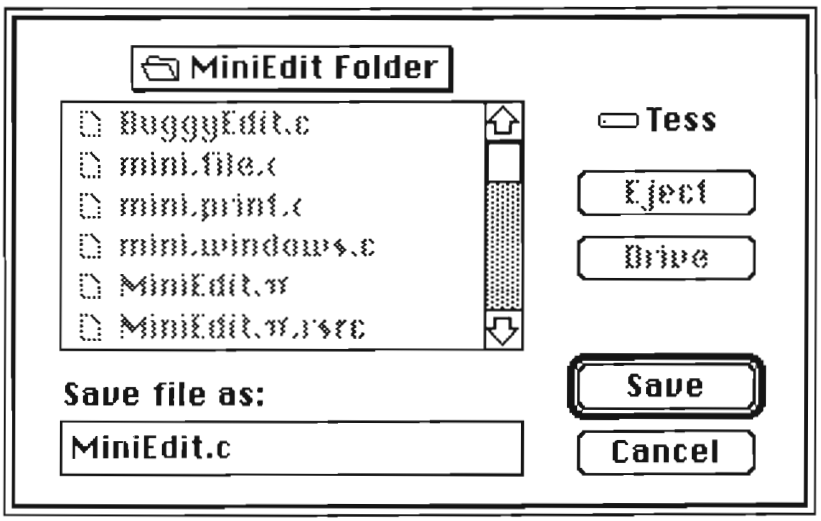
Scroll toward the beginning of the file, and you'll see that the declaration for `myWindow` is commented out:



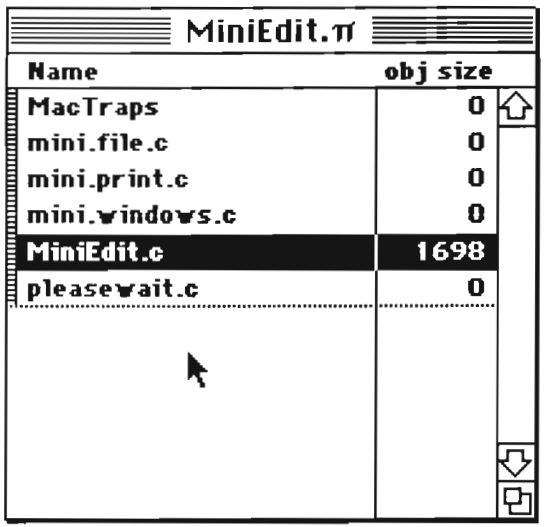
Remove the comments surrounding the declaration of `myWindow`.

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`. (Make sure you're in the `MiniEdit Folder`.)



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 project window as well. The file's object code is now associated with the new name.



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

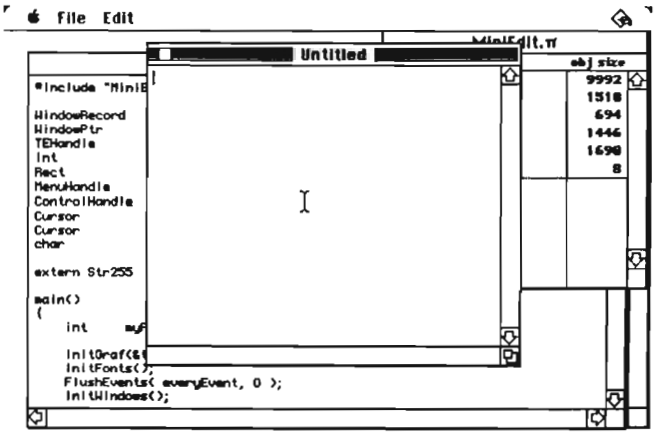
Now that you've fixed the bug, you can try running the project again.

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.

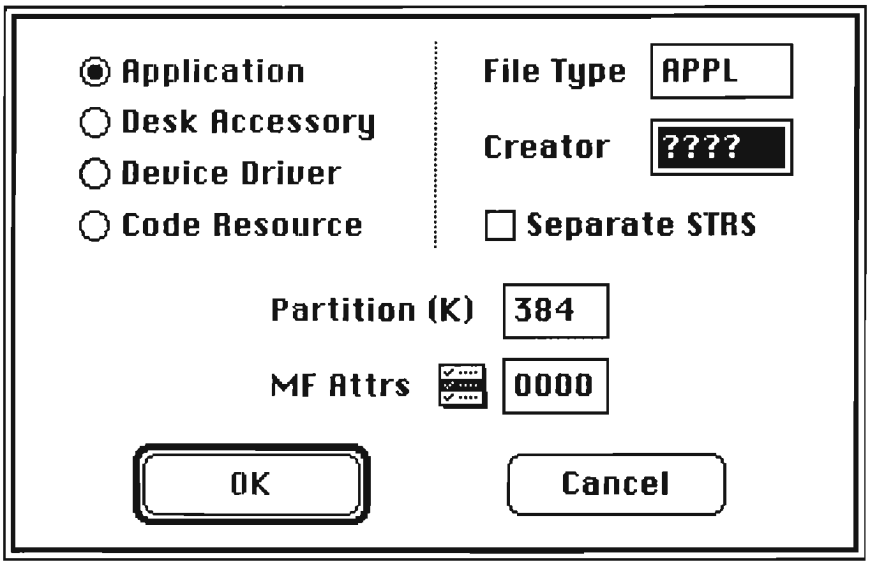


If you're using MultiFinder, THINK C launches your project in its own partition, so you can shift from your project back to THINK C. If you're not using MultiFinder, THINK C launches your project as if you had started it from the Finder. When you quit running your project, THINK C starts up again automatically.

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

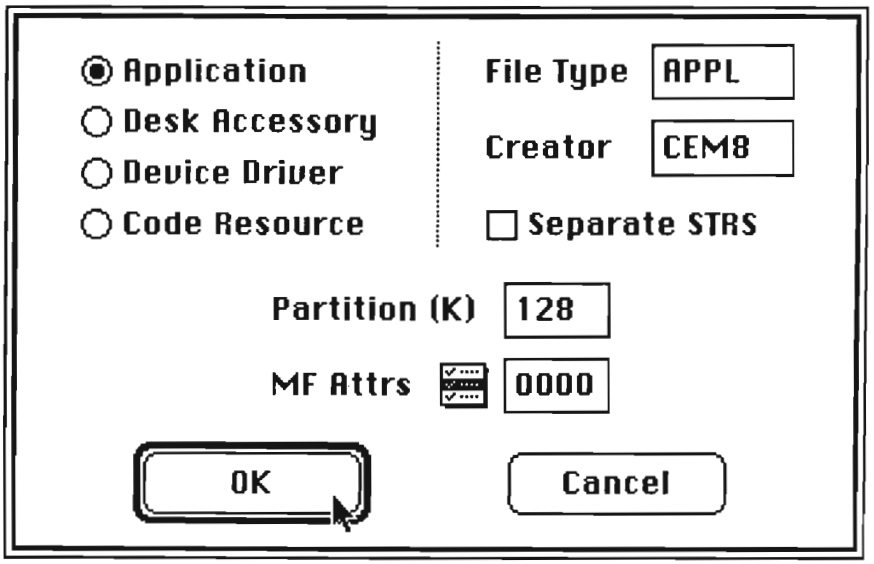
Turning your THINK C project into an application is easy. Choose the **Set Project Type...** command from the **Project** menu. You'll see this dialog box:



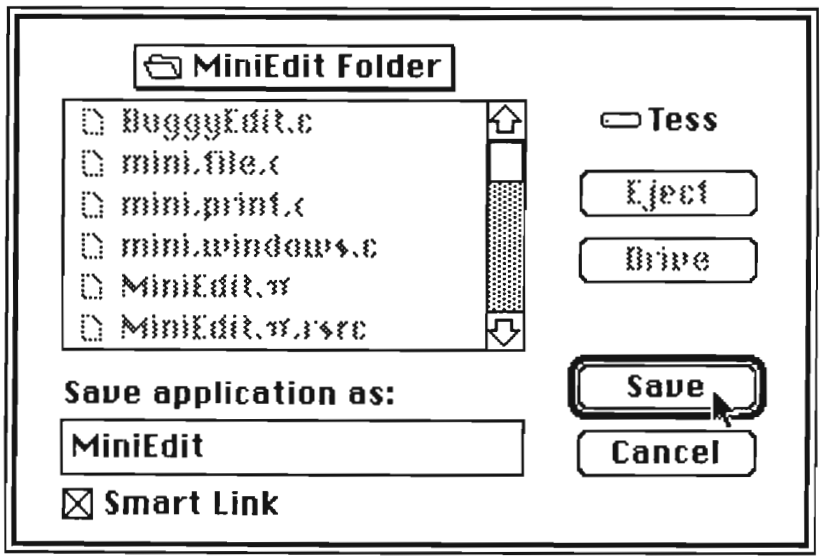
Set the creator to CEM8. This will ensure that your application will have the right icon when you build it. (CEM8 doesn't stand for anything. It's just unlikely that any other application on your disk has that signature.)

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

When dialog box looks like this, click on the OK button:



Choose **Build Application...** from the **Project** menu. You'll see a dialog box like this:



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

Using a Resource File

The `MiniEdit Folder` you copied from disk `THINK C 2` contains a file called `MiniEdit.π.rsrc`. This file contains the resources that the MiniEdit project uses.

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 RMaker or ResEdit utilities (they're included in your THINK C package). `MiniEdit.π.rsrc` was created with ResEdit, so there's no RMaker source 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

Introduction

This chapter shows you how to use THINK C's source level debugger. You'll use an example program called Bullseye, which is included in your THINK C package. Bullseye is a simple application. It draws a series of concentric circles in a small window. Bullseye's **Width** menu lets you select how wide each of the rings is.

Before you begin

Make sure you're running MultiFinder on a Macintosh with at least 2Mb of memory.

Copy the `Bullseye Folder` from disk `THINK C 2` to your development folder. The `Bullseye Folder` contains all the files you'll need for this example.

Make sure that the file `THINK C Debugger` is in the same folder as THINK C (the `THINK C` 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.

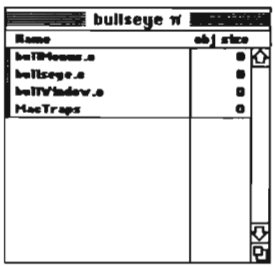
Topics covered in this chapter:

- Opening the Bullseye project
- Turning the debugger on
- Watching the program run
- Examining and setting variables
- Examining structs and arrays
- Expressions and contexts
- Quitting the debugger

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.

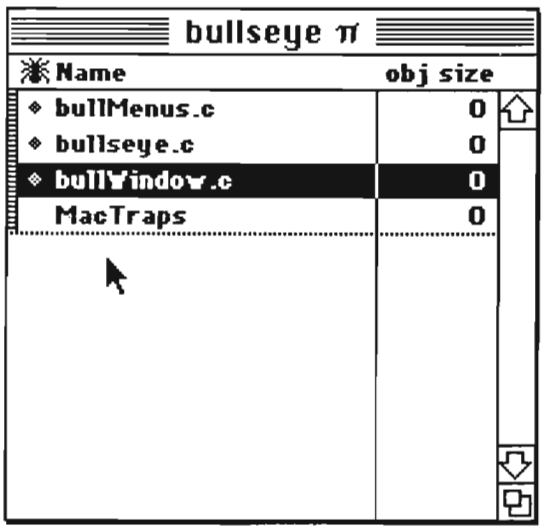


Note that none of the files have been compiled, and that the `MacTraps` library hasn't been loaded.

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.



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.

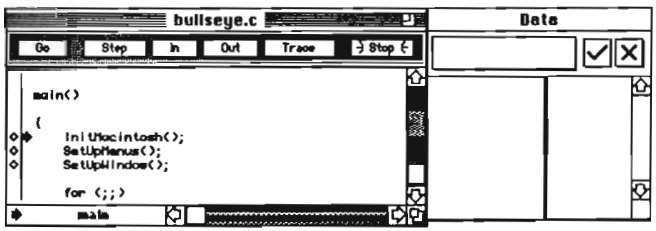
Watching the Program Run

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.

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.



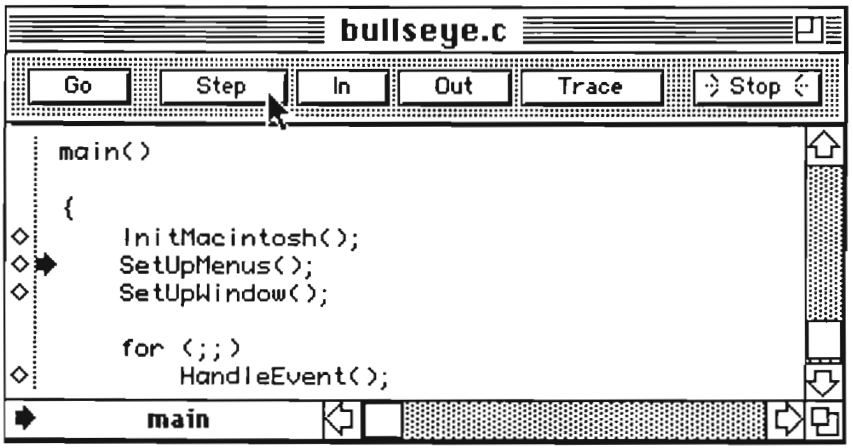
Along the top of the Source window is a row of six pushbuttons called the **status panel**. These buttons control the execution of your program. The status panel also shows you the state of your program. Right now, the Stop button is lit to show you that your program is stopped.

The arrow to the left of the first line of the program points to the statement that's about to be executed. This is called the **current statement**.

The hollow diamonds at the left of the Source window are **statement markers**. The debugger displays a statement marker for each statement in your program. (Loosely speaking, a statement is a line that generates code.) Later, you'll use the statement markers to set breakpoints.

Stepping through statements

Click on the Step button in the status panel.



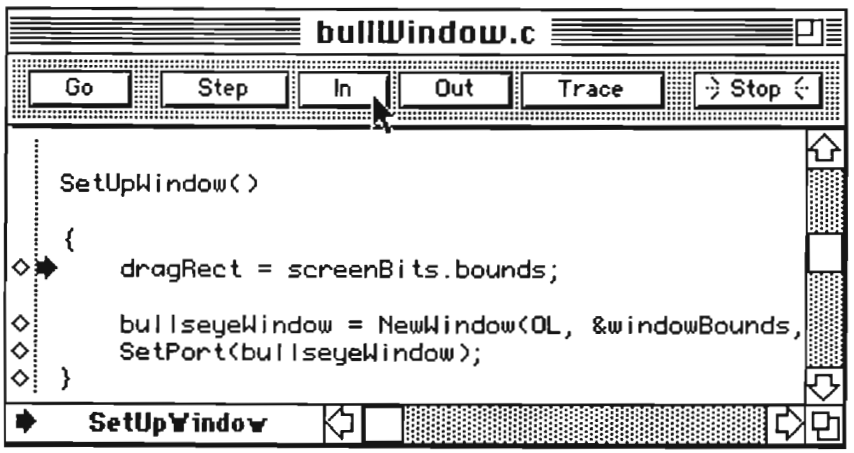
The Step button lights up for a moment, the current statement arrow moves to the second statement, and the program stops again.

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

To see how `SetUpWindow()` works, press the In button on the status panel (or type Command-I). (This command is called **Step In** in the **Debugger** menu.)



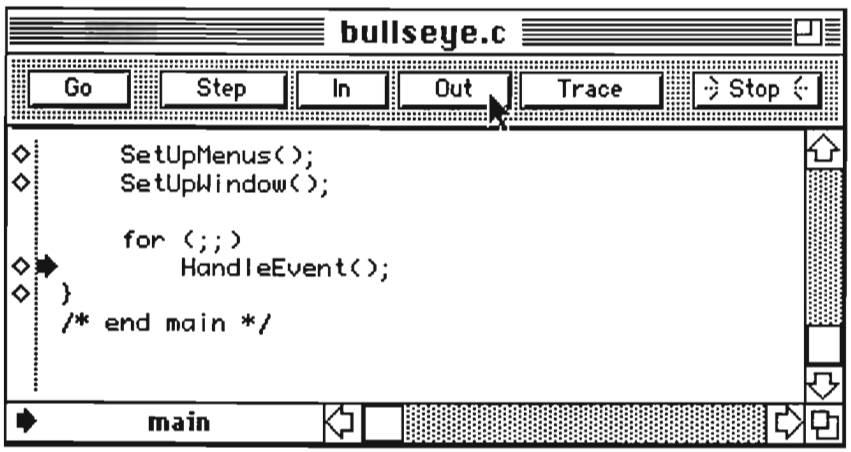
Now the current statement arrow points to the first line of the `SetUpWindow()` function. This function is in the file `bullWindow.c`, so the title of the window changes to let you know what file it's displaying.

> **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

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

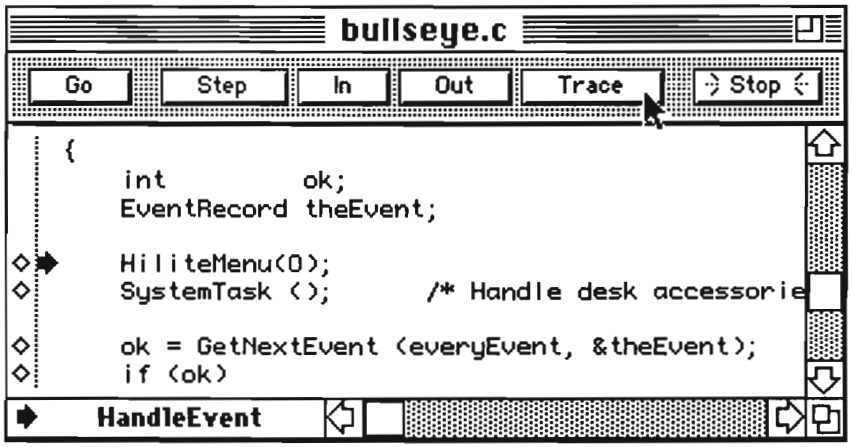
Click on the Out button to leave the `SetUpWindow()` function. The Source window now shows that the debugger is ready to execute the function `HandleEvent()` in the `bullseye.c` file. (Pressing the Out button is the same as choosing **Step Out** from the **Debug** menu.)



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

Tracing every statement

Now click on the Trace button (or type Command-T). The current statement arrow now points to the first statement of the `HandleEvent()` function.



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.

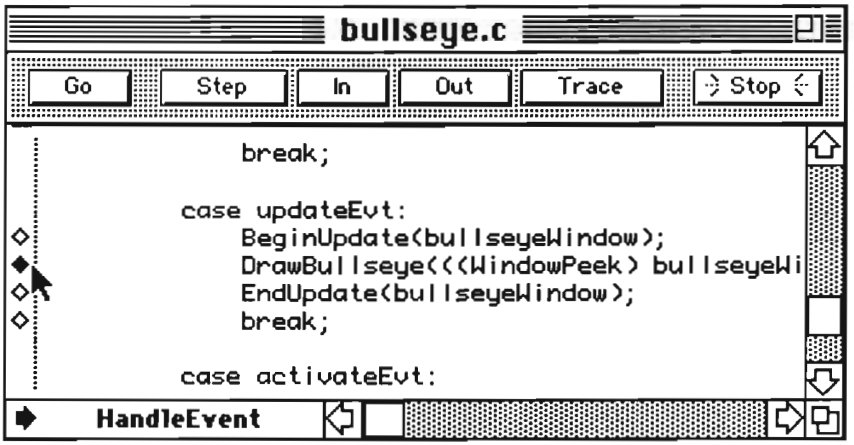
> **Note:** The In button actually does a Trace until the current statement arrow leaves the current function.

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.

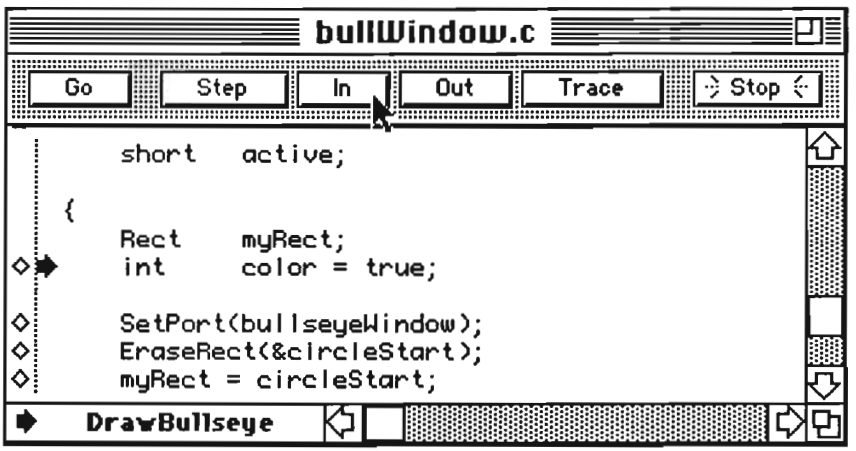
Scroll down in the source window until you get to the code that handles update events. Click on the statement marker to the left of the `DrawBullseye()` function.



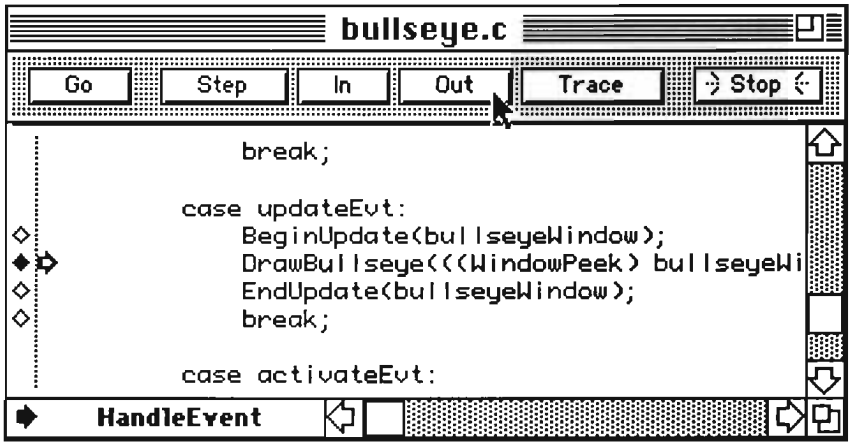
The hollow diamond turns black to indicate that you've set a breakpoint. You can set as many breakpoints as you like this way. When your program is about to execute a statement that has a breakpoint, it will stop. To remove a breakpoint, just click in the filled diamond.

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.)



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()`.



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 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

Sometimes you'll want the Source window to display another file in your project. For example, you might want to set a breakpoint in a file other than the one the current statement arrow is in.

To see a different file in the Source window, first bring THINK C to the foreground. You can do this several ways:type Command-0, choose you project name from the **Windows** menu, click on the project window, or choose THINK C from the list of applications at the bottom of the Apple menu.

Next, click on the file name in the project window. Then choose **Debug** from the **Source** menu. The source file will appear in the Source window, and you can set breakpoints in it.

To display the file that contains the current statement again, just click on the current function name at the bottom left of the Source window.

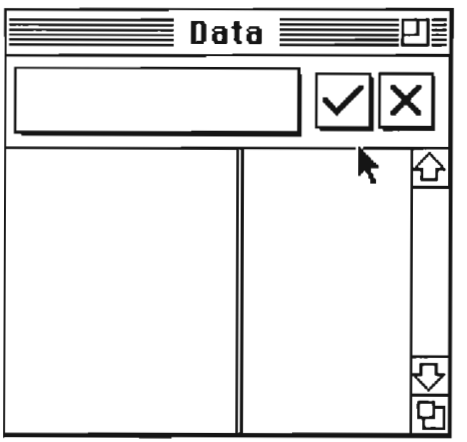
Examining and setting variables

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.



At the top of the Data window is the **entry field** and two pushbuttons. The button marked with a check mark is the **enter button**, and the button marked with an X is the **deselect button**.

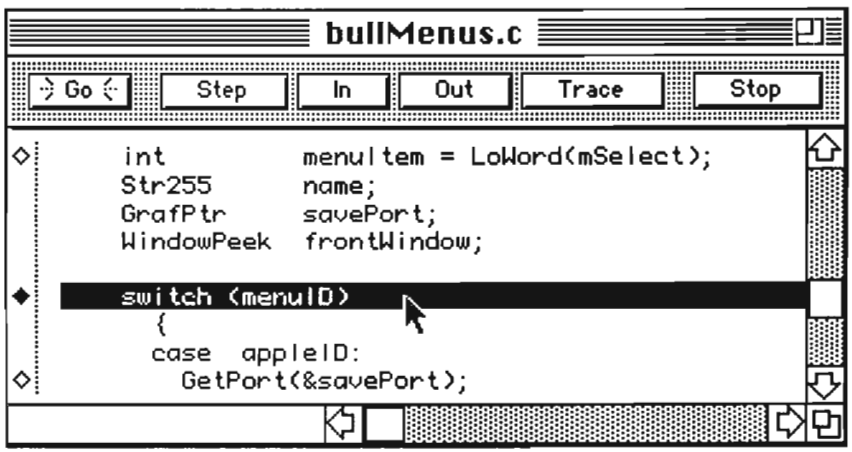
Below the entry field and the buttons are two columns. The column on the left is the **expression column**, and the one on the right is the **value column**. You enter expressions (usually variable names) in the left column, and their values appear in the right column.

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.

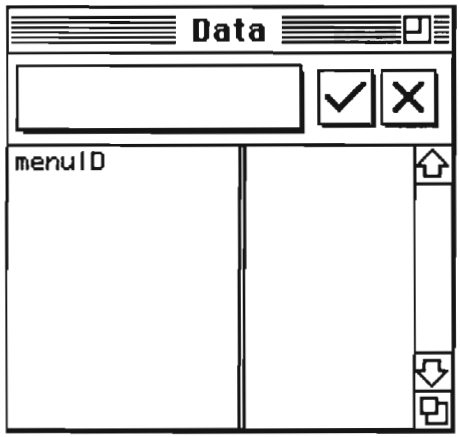


You select a line to give the debugger a **context** for evaluating `menuID`. In this case, you're saying that you want to know the value of `menuID` right before the `switch` statement.

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.

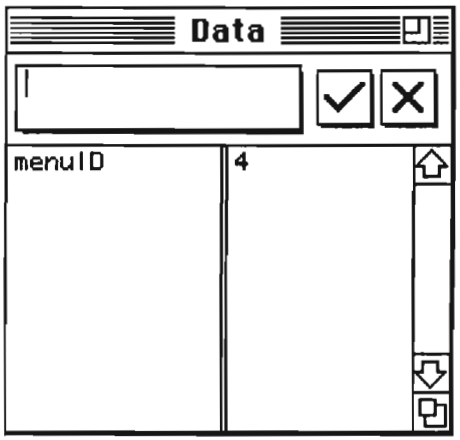
Click in the Data window. You'll see the insertion point blinking in the entry field. Type `menuID` in the entry field and press the Return key.

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.



Now, go back to the Bullseye program. Click on the Bullseye window or type Command-G to bring Bullseye to the foreground.

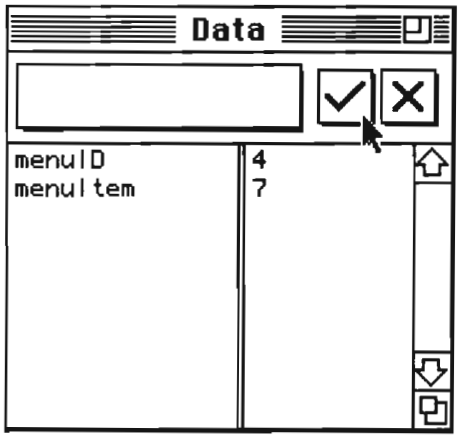
Next, select **7** from the **Width** menu. Your program stops at the breakpoint when you release the mouse button, and the value of `menuID` appears in the value column.



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.

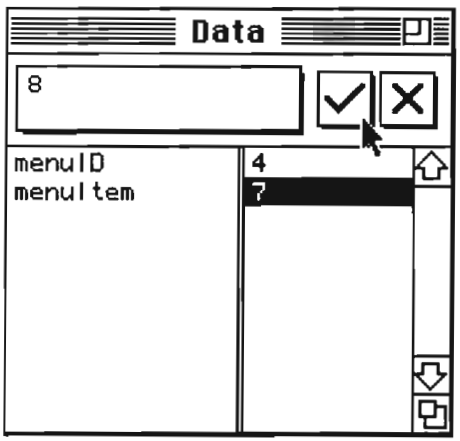
Changing the value of a variable

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.



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.



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" below.

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.

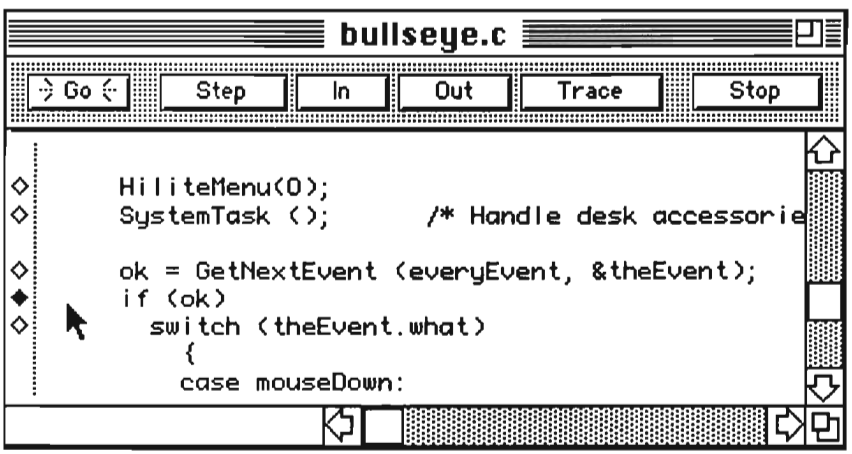
Examining structs and arrays

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.)

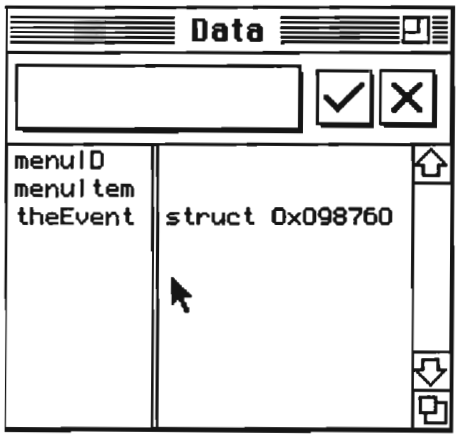
> **Note:** Anything you read here about structs applies to unions as well.

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

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()`.



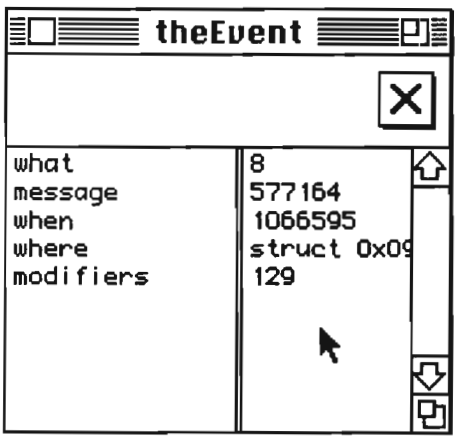
Now click in the Bullseye window. The program will stop at the breakpoint. When it does, type `theEvent` in the entry field of the Data window, and press the Return key.



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.



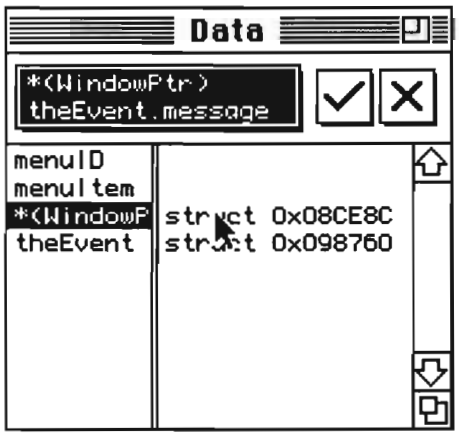
You can edit the values of the fields, but you can't edit the names.

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.

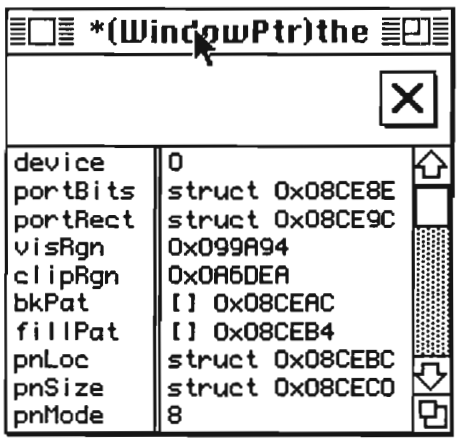
Double-click on the `message` field. The debugger enters a new expression in the main Data window: `theEvent.message`.

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

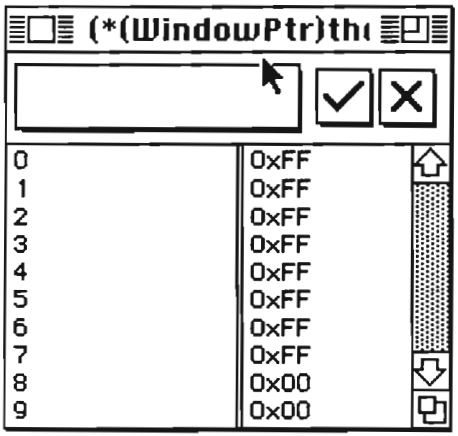
Edit the expression so it reads `*(WindowPtr)theEvent.message` so you can look at the `WindowPtr`.



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



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



Because C compilers don't enforce array bounds, array windows have "infinite" scrollbars. 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 type any C expression in the entry field of the Data window as long as it doesn't have any potential side effects. This means that you can't type in a function call, an assignment statement, or any expression that uses the autoincrement (++) or autodecrement (--) operators.

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. 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`, `\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`.

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's LightspeedC

PART THREE

Using THINK C

Overview 6

Introduction

This chapter describes the THINK's LightspeedC 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.

Topics covered in this chapter:

- The THINK C environment
- The Project
- Writing a program in THINK C
- Using THINK C

The THINK C Environment

THINK's LightspeedC 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 was 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 work in concert as parts of the same application. This way, THINK C 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 recompilation. 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 final 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, the first thing you do is create a project document. Usually, the project document is in a folder that you'll use for all the files related to your program.

Next you'll create your source files and add your libraries. THINK C source files are standard text files, so you'll be able to 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` library.

Your THINK C package also includes several other libraries you can use in your programs. The `stdio` library contains the standard I/O functions found on most C systems. 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.

Step	Action
Create a folder for your project	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.
Start THINK C	Double click on the THINK C icon from the Finder.
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.

Step	Action
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.
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.
Add source files to the project	Use the **Compile** command in the **Source** menu to compile the active source file and add it to the project window automatically. If you don't want to compile a source file, but you still want to add it to the project window, the **Add** command in the **Source** menu will add the active source file. Use the **Add...** command to add source files you haven't opened with the THINK C editor.
Add libraries	Use the **Add...** command in the **Source** menu to add libraries to the project window. A standard file dialog will appear. Move to the folder that contains the library you want to add, and click on the Add button. The dialog box will reappear to let you add more libraries. When you're done, click on the Cancel button.
Run project	Use the **Run** command in the **Project** menu to run your project. If there are uncompiled 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.

Step	Action
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

Introduction

You can write applications, desk accessories, device drivers, and any kind of code resource in THINK C. This chapter tells you how to build different 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: applications, desk accessories, device drivers, and code resources.

What you should know

You should know how THINK C works. If you haven't done so, run through the MiniEdit example in Chapter 4. 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.

If you want to write Macintosh applications, you should know about the resources that make up an application: menus, window templates, dialogs, control templates, etc. You should know how to build these kinds of resources with a resource editor, like ResEdit, or a resource compiler, like RMaker. If you don't know about these resources, look in *Inside Macintosh I*, and read the chapters that talk about the resources you want to build. *Inside Macintosh I*, Chapter 5, "The Resource Manager" talks about the resource manager in general.

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

If you want to build other kinds of code resources (`INIT`s, `WDEF`s, `cdev`s, etc.) see the section "Building Code Resources" later in this chapter.

If you don't know how to call the Macintosh Toolbox routines, read the section "Calling the Macintosh Toolbox Routines" in Chapter 10.

Topics covered in this chapter:

- Anatomy of a project

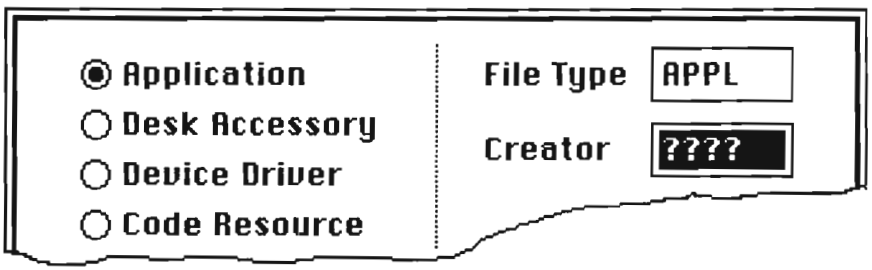
- Segmentation
- Building applications
- Building desk accessories and device drivers
- Building code resources

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 more 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 #include file. And if you're using the source level debugger, the project keeps the tables that the debugger needs.

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.



The File Type and Creator of a file let the Finder know what icon to display. Some types, like `RDEV`, `INIT`, and `cdev`, are treated specially by the Macintosh System software. To learn about File Types and Creators (also called signatures), see *Inside Macintosh III*, Chapter 1, "The Finder Interface."

There is a section in this chapter for each project type. Each section describes the type-specific settings.

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'll have to recompile everything if you do so.

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

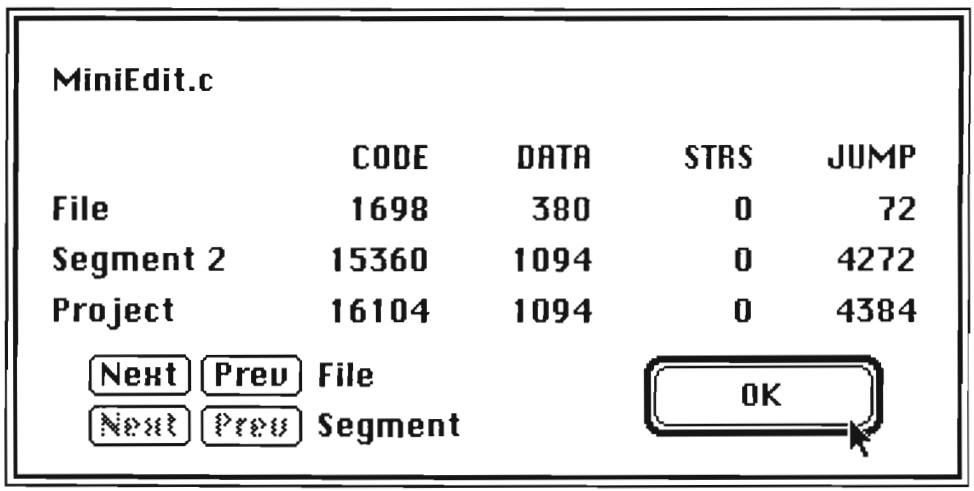
Changing the type of a project changes the way a project is built, not the way a project behaves. You can't turn an application into a desk accessory merely by changing the project type. Desk accessories are structurally different from applications.

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 appropriate CODE resources for applications, a DRVR resource for desk accessories and drivers, or whatever type you specify for your code resource. THINK C also creates resources that 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: CODE, DATA, STRS, and JUMP. The CODE component contains the object code generated for the project. The DATA component contains the global and static variables. In applications, string literals and floating-point constants can be stored separately in the STRS component. Finally, the JUMP component contains the jump table. Not all project types use all four components.

To examine the sizes (in bytes) of each of the components, select a source file in the project window and choose the **Get Info...** command in the **Source** menu. The display also shows the segment and project totals.



For some components there is a small amount of per segment or per project overhead.

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: 32K per project JUMP: 32K per project STRS: unlimited (if you use them)
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
Code resource	CODE + DATA: 32K per project JUMP is not used for code resources

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` as well as for each use of a function in a non-call context (i.e. whose address is taken). The jump table built by the **Build Application...** command may be smaller; the entry for each function that is only referenced within a single segment is removed.

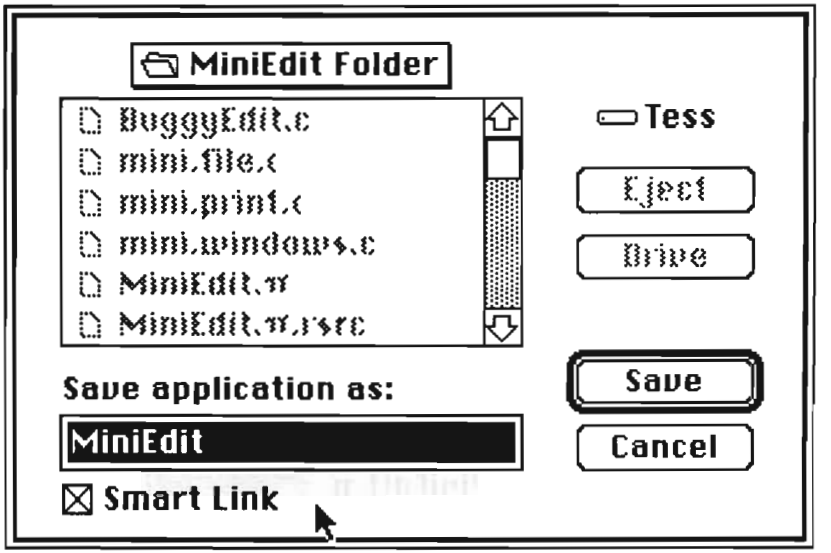
How THINK C puts projects together

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.

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 will be 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 will be larger, but they'll come out faster.

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



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.

Use ResEdit or RMaker 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 name `SuperWizzy.project`, THINK C will look for the project's resources in `SuperWizzy.project.rsrc`.

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.

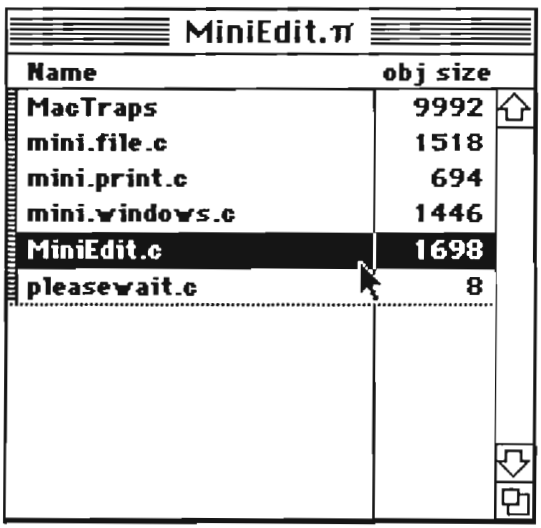
When you choose one of the **Build...** commands from the **Project** menu (**Build Application...**, **Build Desk Accessory...**, etc.) THINK C copies the resource file into the finished application.

Segmentation

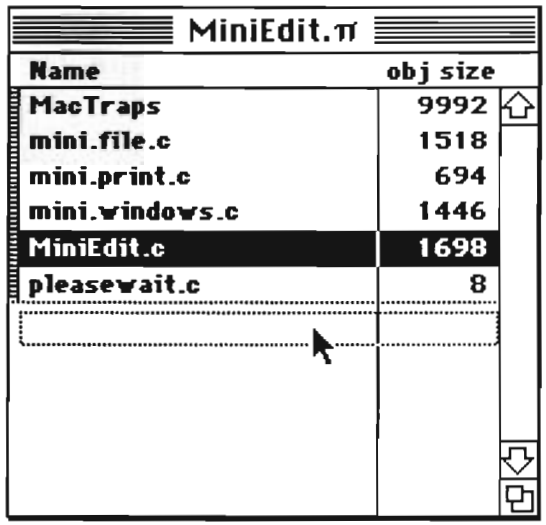
Most Macintosh programs are made up of several **segments**. Segments are units of object code that can be swapped in and out of memory as needed. The Macintosh Operating System limits segments to 32K, so if you're writing a large program, you will have to segment your code. To learn more about segments, see *Inside Macintosh II*, Chapter 2, "The Segment Loader."

THINK C lets you segment not only applications, but desk accessories and device drivers as well. Applications can have up to 254 segments. Desk accessories and device drivers can have up to 31 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.



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



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.

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.

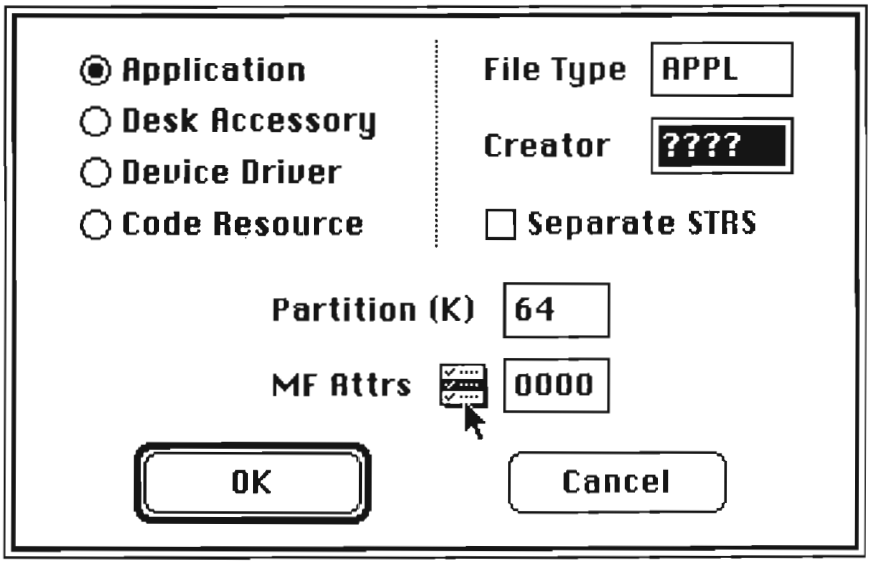
If you move all the files out of a segment, THINK C deletes the empty segment. When you use the **Add...** command, new files are placed in the current segment.

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

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'll see this dialog:



Setting the application file type and creator

When you're building an application, the default file type is APPL. (In order 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 separate STRS

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

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

The only time you might want to use this option is when your DATA component is coming close to the 32K per project limit. Separating the STRS component might open up some room in the DATA component. Remember, there's no limit to the size of the STRS component.

If you build a library (or use a project as a library) with the Separate STRS option on, you can use it only in projects that have the option on as well. To ensure compatibility for both cases, leave the option off when you build a library.

Setting the partition size and MultiFinder attributes

THINK C uses the values you set in the Partition and MF Attrs fields to build the SIZE resource your application needs to run under MultiFinder.

The **partition size** determines how much memory your application gets under MultiFinder. 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.

The MF Attrs sets the flag that tells MultiFinder how compatible your application is. To learn how to make your application MultiFinder compatible, you can order the MultiFinder Development Package from APDA. (See Chapter 1 for more information about APDA.)

You can type in the value of the flag (in hex) in the field, or you can set the bits from the pop up menu next to the field. The pop up menu looks like this:

If the MultiFinder-Aware bit is set, MultiFinder expects you to conform to the MultiFinder guidelines for shifting from the foreground to the background layers. Your application will get suspend/resume events as your application shifts from foreground to background but not activate/deactivate events.

> **Note:** 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 will get these events as it shifts from the foreground to the background layers in addition to the activate/deactivate events you normally receive.

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, your application runs in its own partition.

Under MultiFinder, you can watch your application run and examine your source code at the same time. You can switch back and forth between your application and THINK C to edit

your source files. When you quit your application, you'll be back in THINK C, and the automake facility will be 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.

> **Note:** When you're running under MultiFinder, you have to be a little more careful about stray pointers. 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 will be 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. You do have a backup, don't you?

Building Desk Accessories and Device Drivers

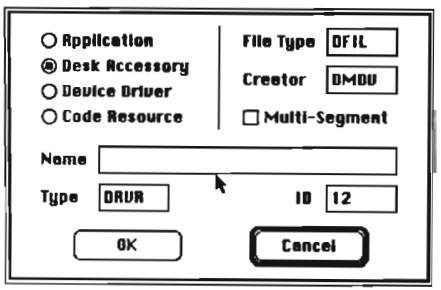
Desk accessories and drivers are structurally identical; they're both **drivers**. According to *Inside Macintosh*, drivers don't behave much like applications and have a different internal structure. In this section, the word *driver* by itself means either a device driver or a desk accessory.

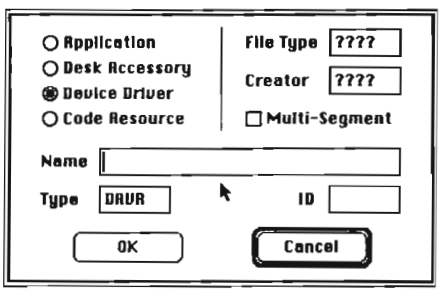
This section won't teach you how to write a desk accessory or a device driver from scratch. To learn how to write these, read *Inside Macintosh I*, Chapter 14, "The Desk Manager," *Inside Macintosh II*, Chapter 6, "The Device Manager," and *Inside Macintosh V*, Chapter 23, "The Device Manager."

Setting the project type

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

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





The Desk Accessory dialog presets the File Type and Creator so the resulting file will be a Font/DA Mover file. The ID is set to 12. The Font/DA Mover 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.

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 preset fields 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 checkbox. You can learn more about multi-segment drivers below, but read how a driver works first.

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.

The THINK C driver glue also does two things designed to make writing a driver easier. First, it sets up a data area so that your driver can have its own global variables. Second, it figures out the proper way to return control to the Device Manager automatically. 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 an `int` and is *not* declared `pascal`. A typical driver skeleton looks like this:

```
main (paramBlock, devCtlEnt, n)

    cntrlParam *paramBlock;
    DCtlPtr devCtlEnt;
    int n;

{
    switch (n) {
        case 0 : /*   Open    */
        case 1 : /*   Prime   */
        case 2 : /*   Control */
        case 3 : /*   Status  */
        case 4 : /*   Close   */
    }
}
```

`ParamBlock` 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` 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 and floating point constants 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 will be 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 will 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 #include 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` will be 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(dp, eventp, itemp)
DialogPtr dp;
EventRecord *eventp;
int *itemp;
{
    Boolean result;

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

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

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` library doesn't reference any globals, so you can use it 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 can't use them in drivers without modifying 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 driver global */
        return;
    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 will not be opened.

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

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

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

The jIODone problem

THINK 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 "goodbye kiss" (`csCode=-1`) issued to desk accessories that have requested to be notified when the current application exits out from under them. Other calls may be queued or immediate.

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

It is incorrect to violate these rules; in particular, it is incorrect to jump to `jIODone` to return from an immediate call. `jIODone` will attempt to examine the driver's I/O queue, and since the queue is usually empty it will end up examining low-memory locations beginning at `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.

> **Note:** Do not use `UnloadSeg()` instead of `UnloadA4Seg()` by mistake!

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 startup. 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.)

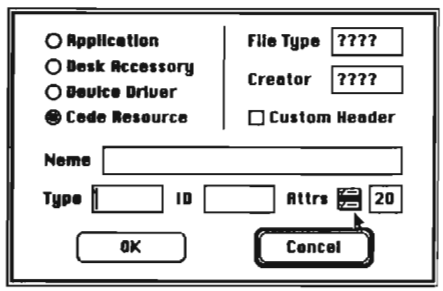
This section tells you how to build code resources in THINK C. The specific formats and calling sequences for code resources are given in the various volumes and chapters of *Inside Macintosh*. This list will help you get started.

To learn how to build a...	Read *Inside Macintosh*...
ADBS resource	Volume V, Chapter 20, "The Apple Desktop Bus"
CDEF resource	Volume I, Chapter 10, "The Control Manager"
cdev resource	Volume V, Chapter 18, "The Control Panel"
FKEY resource	Technical Note 3 (also see below)
INIT resource	Volume IV, Chapter 29, "The System Resource File" Volume V, Chapter 19, "The Start Manager"
LDEF resource	Volume IV, Chapter 30, "The List Manager Package"
MBDF resource	Volume V, Chapter 13, "The Menu Manager"
MDEF resource	Volume I, Chapter 11, "The Menu Manager" Volume V, Chapter 13, "The Menu Manager"
WDEF resource	Volume I, Chapter 9, "The Window Manager"

Setting the project type

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



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) popup 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."

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, 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(varCode, theWindow, message, param)
int varCode, message;
WindowPtr theWindow;
long param;
{
    ...
}
```

Global data in code resources

Code resources, like applications and drivers, can have global and static variables. When you build a code resource, the DATA component is appended to the CODE component, so CODE and DATA together 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:** THINK C treats string literals and floating points constants the same way as globals in code resources. If you're using literals and constants and not globals, you still need to set up A4. Earlier versions of THINK C did not handle string literals and floating point constants this way.

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 #include 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();
}
```

> **Note:** This technique works only when you use the default code resource header. If you use a custom header, you'll have to 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 library does not access any globals, so you can use it in your drivers. 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 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.

The other libraries supplied with THINK C reference their globals from register A5, so you can't use them in code resources without modifying 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 globals */
    SetUpA4();

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

    ...

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

> **Note:** This technique works only when you use the standard code resource header. If you use a custom header, you'll have to 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. `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.

> **Note:** `SetUpA4.h` generates code, so if you #include it at the top of your file, the internal function defined in `SetUpA4.h` will be your header. This is not what you want.

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

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

header()  /* First function in file */
{
    asm {
    DC.L    0   /* header information */
    ...
    /* end of header */
    LEA header, a0 /* put address of code resource in a0 */
    JMP main
    }
}

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

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

    ...

    RestoreA4();
}
```

The Editor
8

Introduction

This chapter describes the features of the built-in THINK's LightspeedC 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 #include files easier.

Topics covered in this chapter:

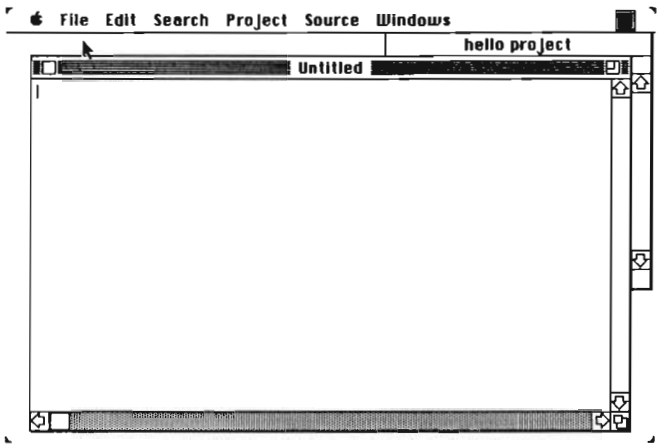
- Creating and opening files
- Editing text
- Printing files
- Closing and saving files
- Searching and replacing text

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 will allow, and each file appears in its own edit window. Although you usually create and open source or header files, you can also use the THINK C editor to open any text file.

Creating a new file

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



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, just 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.

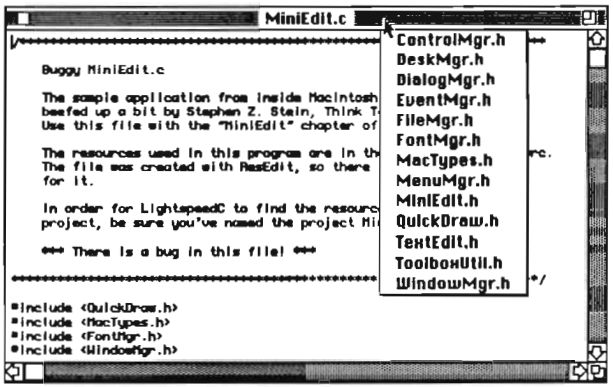
You can open a file by typing the first portion of its name and then pressing Return or Enter when the project window is the active window. Since the files in the project window are in alphabetical order, the selected file is the first file in the project which matches the characters typed so far. If 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 #include files

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

Hold down the Option or Command key as you click in the title bar of a source file's edit window to get a pop up menu of all the files included in the source file. Choose the #include file you want to look at and it will appear in its own edit window. If the file is already open, its window will be brought forward.



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

THINK C builds the pop up menus only for compiled source files. Any #include files that are part of a precompiled header don't appear in the pop up menus. (To learn more about precompiled headers see Chapter 10.)

If you want to open an #include file that you've added to the source file since the last compilation, or if you want to open an #include 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 an `#include` 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 will automatically be 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.)

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 will select the entire word. **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 will read Undo Cut. If you select Undo, the command will now say Redo Cut. You can undo the most recent replace (see Searching and Replacing in the next section), 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

THINK C includes a feature that allows you to examine other areas of the text, then instantly jump back to where you were. Pressing the Enter key while you are anywhere in the text will reposition the text to show the current insertion point or the start of the current selection. Since scrolling in any direction does not affect the insertion point or the current selection, this will take 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 is particularly useful after using the **Balance** command.

Using the arrow keys

The arrow keys on the Macintosh Plus, Macintosh SE, and Macintosh II keyboards move the insertion point up, down, left, and right. Holding down the Option key with an arrow key moves the insertion point to the beginning of file, end of file, beginning of line, and end of line, respectively. Shift-arrows and Shift-Option-arrows extend the current selection.

> **Note:** Because of the Macintosh keyboard hardware design, it's not possible to distinguish between Shift-arrows and the +, *, /, and = keys on the numeric keypad of the Macintosh Plus keyboard. For example, when you type a + on the keypad, it will be treated as a Shift-left arrow. The shifted versions

of these keys are the same as Shift-arrow combinations. On the Mac SE and Mac II, the +, *, /, and = keys work correctly.

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

If you indent a line with leading tabs or spaces, the editor will indent the following lines by the same amount of space.

To keep a line from auto-indenting, hold down the option key when you press Return. The editor uses tabs and spaces to indent the line, so you can just backspace over them if you want to change the indentation.

Shifting blocks right and left

To change the indentation level for a range of lines use the **Shift Left** and **Shift Right** commands from 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.

> **Note:** Do not type anything while doing **Shift Left** or **Shift Right** on a selected region. This will, of course, replace the selected text with what you typed.

Balancing parentheses, brackets, and braces

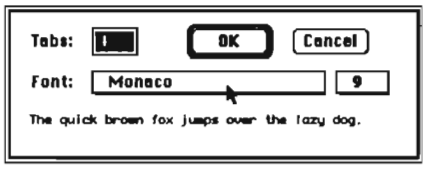
The **Balance** command in the **Edit** menu extends the current selection in both directions until it encloses the smallest surrounding balanced text enclosed in parentheses `()`, brackets `[]`, or braces `{}`. Successive invocations select larger sequences of text.

> **Note:** The **Balance** command just scans for matching characters without taking into consideration whether the match is inside a string or a comment.

Try this: 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. This is a quick way to check whether all your function definitions are properly balanced.

Changing font and tab settings

When you open a new edit window with the **New** command, the font is preset to Monaco-9, and the tab stops are set to every 4 spaces. You can change these settings with the **Set Tabs & Font...** command in the **Edit** menu. When you choose this command, you'll see this 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.

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 either the ImageWriter or LaserWriter.

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 usually come at the most inopportune times, and strange machine crashes do occur.

Closing a File

To close a file, click on its edit window's close box. If you've made changes, and you haven't saved the file, the editor will ask 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 will change 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 files the contents of the current edit window under a new name, but lets you continue editing the original file.

Saving and closing all open files

To save all the open files, use the **Save All** command in the **Windows** menu. 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 will ask you if you want to save it.

Saving Files Automatically

THINK C will save files for you automatically when you close them if you uncheck the Confirm Saves checkbox in the Preferences section of the **Options...** command. See Chapter 14 for more information about the **Options...** command.

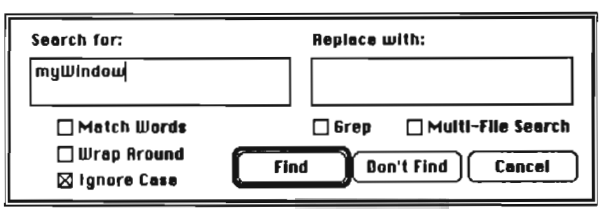
Searching and Replacing

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 **Edit** menu when you want to find a string. You'll see this dialog box:



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, it will be highlighted. 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 checkboxes on the left side of the Find dialog let you specify how the editor looks for your string. You can set the defaults for these options with the **Options...** command in the **Edit** menu.

If you check the Match Words option, the editor will match only whole words. This option is useful when you're looking for one-letter variable names, for instance.

The editor usually searches from the insertion point to the end of the file. If you check the Wrap Around option, it will search the entire file for your string. The search begins from the insertion point (or the end of the selection). If your string hasn't been found by the time the editor gets to the end of the file, it searches from the beginning to the insertion point.

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

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. Then, when the editor finds the first occurrence, you can use the **Find Again** command to go on to the next instance, **Replace**, to replace it with the replacement string, or **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 will delete every instance of the search string (that is, it will replace 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.

You might want to 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 the **Find Again** command to find your string.

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

Finding Non-printing Characters

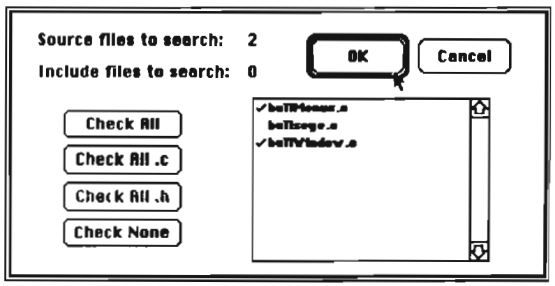
To look for tab and return characters, hold down the Command key as you type them into the Search for and Replace with fields. To insert other non-printing characters, use the **Copy** command to copy them into the Clipboard, then **Paste** them into the Search for and Replace with fields.

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. 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 checkbox in the **Find...** dialog box. When you check this box, another dialog box displays all the text files associated with the project.



Scroll through the list and click on individual files to select them. A small check mark appears next to the file name.You can use the buttons in the dialog box to Check All, Check None, Check All .c, or Check All .h files. (If a file is already selected, clicking on its name will remove the checkmark.)

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.

THINK C looks for the search string through each of the checked files, starting with the first one checked. When it finds a file that contains the search string, THINK C 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 just finds the first instance of the search string in each file. To find subsequent instances of the search string in the file, use the **Find Again** command. Once you issue a **Find in Next File** command, THINK C will look 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 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 will appear. 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 checkbox.

Finding the definition of a symbol

To find the definition of a symbol (a function name or a variable), hold down the Option (or Command) key as you double-click on it. THINK C opens the file in which the selected symbol is defined and looks for its first occurrence in that file. Usually, the first occurrence is the definition of the symbol. You can use the **Find Again** command if the first instance wasn't the definition.

If the editor can't determine where the symbol is defined, you'll hear a beep.

This feature relies on information the compiler keeps in the project file, so the file that defines the symbol must be compiled. This feature only works for global (non-static) functions and variables.

If the symbol is multiply defined, the editor arbitrarily opens one of the files that defines it.

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.

Note: The editor will look for patters 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.

- Any character, with the exceptions noted below, is a pattern that matches itself.
  Example: The pattern 2 matches a character 2. If you've checked Ignore Case in the **Find...** dialog box, any letter will match both its upper- and lower-case equivalent. So, either `a` or `A` will match both a and A.

- The character `.` is a pattern that will match any character.

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

- 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 a string of three characters in the form `a-b` appears in s, this represents all of the characters from `a` to `b` inclusive. All other characters in `s` are taken literally. The only way to include the character `]` in `s` is to make it the first character. Likewise, the only way to include the character `-` in `s` is if it appears either at the beginning or at the end of `s`.
  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 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.

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

- 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.
  Example: The pattern `P.` matches any string beginning with P and any other character.

- Of course, 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 `.*` will match the empty string between the (and the).

Will this pattern match the string `(())`? Since the sub-pattern `.*` will match 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 will match `(())`.

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.

- A pattern surrounded by `\(` and `\)` matches whatever the sub-pattern matches.
  Example: `\(a[b-y]z\)` matches the same thing as `a[b-y]z`.

- 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.
  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.

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

- 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 \>.
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`

- 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.

- 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 occurs 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.

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

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

- 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 will not match equ if it is part of another word, for example equal. `\<equ\>` is not surrounded by `\(` and `\)` because it will be 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` will consist of everything after the equ to the end of the line and `\3` will be 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

Introduction

This chapter tells you why your files and folders are organized the way they are, how THINK C looks for #include files, and what happens when you move your files from one folder to another or to another machine.

Before you begin

Make sure that you followed the installation instructions in Chapter 2. 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.

Topics covered in this chapter:

- Organizing your folders
- How THINK C names files
- How THINK C looks for #include files
- Moving files within a project
- Using the THINK C and project trees

Organizing Your Folders

This section tells you how THINK C knows where to find your source files, libraries, and #include files. THINK C expects to find your source files, libraries, and #include files relative to the folder THINK C is in or relative to the folder your project is in. The diagram at the end of this chapter shows you how your disk should be set up.

The THINK C and Project trees

THINK C treats all the files in all the folders within the `THINK C Folder` as if they were in the same flat folder. From now on, we'll call the `THINK C Folder` and all the subfolders in it the **THINK C Tree**.

THINK C treats all the files in your project folder as if they were all in the same folder. From now on, we'll call the folder and subfolders your project is in the **project tree**.

This organization lets you put your files into folders as you like, without worrying about pathnames. Since you use a standard file dialog to add files to your project, THINK C knows

which tree they're in. If you move them around later on, THINK C looks in the appropriate tree to find where you put them.

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:

`<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 #include Files

These are the rules THINK C uses to find #include files:

`<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 an #include 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.

Another example: You might use `#include <QuickDraw.h>` in your program to get the definitions for QuickDraw. If you look at `QuickDraw.h`, you'll see that it contains `#include "MacTypes.h"`. The `Mac #includes` folder is `QuickDraw.h`'s referencing folder, which is in the THINK C Tree, so that's where the preprocessor looks first to find `MacTypes.h`.

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 that's already in one of the trees to the other tree, it's best to use the **Save As...** command in the file menu. This command will take 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:

- Open the file you want to move. Double clicking on its name in the project window is the fastest way.
- 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.
- 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.

> **Note:** If you have Symantec's HFS Navigator™, you can delete the original file before you save it at the new location. When you get the standard file dialog box, hold down the Command key as you click on the folder pop up menu, and choose Get Info. Click on the delete button to delete the original file. (Don't worry, the file is in the edit window.) Then go on to save the file in the new folder.

Moving a library

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

- Move the library to a folder in the other tree. Use the Finder or a file-moving desk accessory.
- Choose **Remove** from the **Project** menu to remove references to it from the project.
- Use the **Add...** command in the **Project** 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 a new machine, 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.

Using the Trees

The way THINK C keeps track of your files gives you the flexibility to organize your files just the way you like without having to specify full path names. There are a couple of points you should remember, though, 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 (see below).

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.

Note: 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 Moving Files Within a Project above.

Shielding folders from the trees

To shield a folder from the 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. Since THINK C doesn't see these files, it's OK if they have the same name as another file in the tree. Project specific folders are the only exception to this rule.

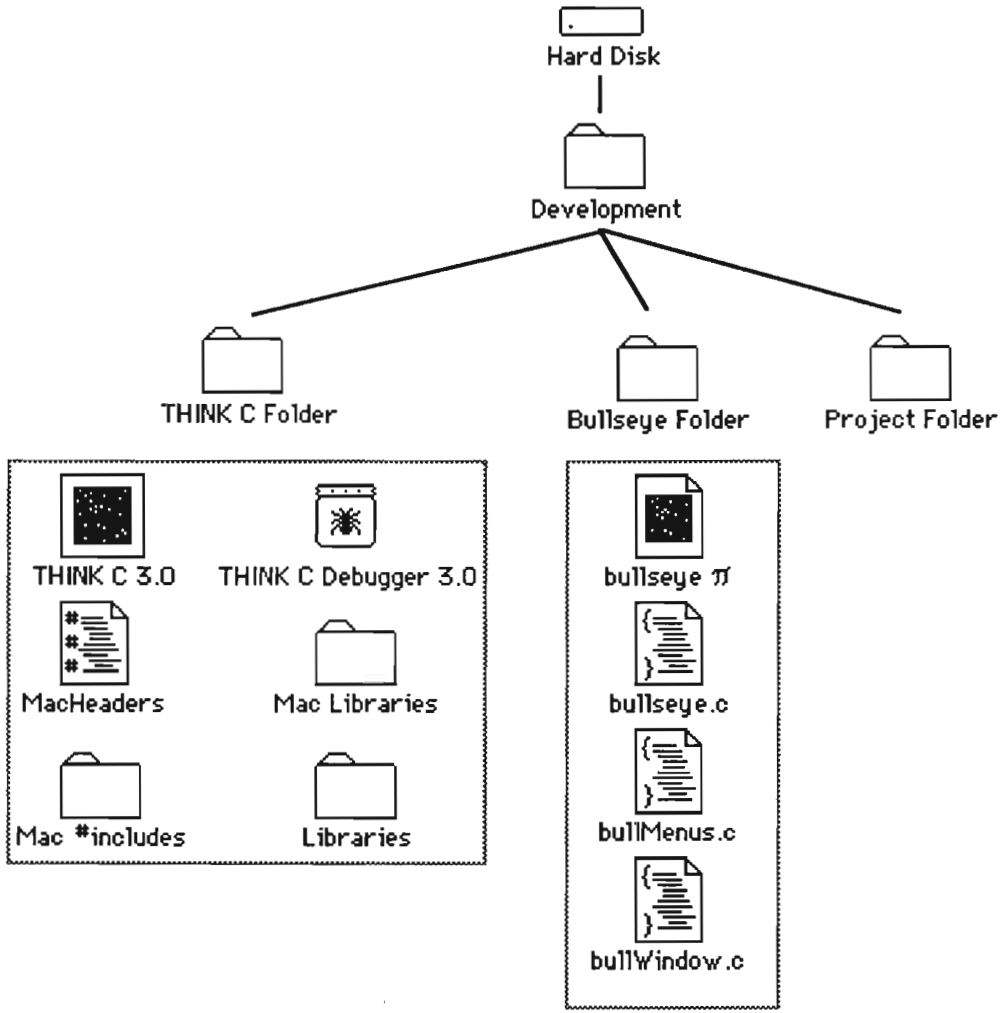
Project specific folders

There is one exception to the shielding rule above. 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.

This feature is useful is you're working on two projects that share files. For instance, suppose you're working on two projects, `INITProject` and `DAProject`, that share some source files. You create two folders, `(INITProject)` and `(DAProject)`, that both contain versions of the #include file `config.h` tailored to control conditional compilation of the common source files.

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.



The Compiler
10

Introduction

This chapter describes features unique to the THINK C compiler. It tells you how to compile source files, how to use precompiled headers, and how to call the Macintosh Toolbox routines. This chapter also tells you how to set options that affect the way THINK C compiles your source files. If you're porting code from other compilers or writing code that will run on other machines, read the Portability section at the end of this chapter.

Topics covered in this chapter:

- Compiling source files
- Precompiled headers
- Calling Macintosh Toolbox routines
- Code generation options
- Compiler options
- Function prototypes
- Portability

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.

Click anywhere in the alert box or press the Return or Enter key to get rid of it.

The source file that contains the error will be in an editor window with the insertion point at the beginning of the line that contains the error.

Identifier length and capitalization

In THINK C every character in an identifier and its case is significant, even when the identifiers aren't in the same file. For example, suppose you had two files like this:

```
/*  file 1  */
int a_very_long_external_name;
---------------
/*  file 2  */
extern int a_very_long_external_nme;  /* misspelled */
```

THINK C would report the second misspelled identifier as an undefined symbol at link time.

THINK C is stricter than other compilers in this matter to help you catch spelling and capitalization errors.

Using register variables

You can declare up to eight register variables per function. Five variables can hold data in registers `D3-D7`, and three variables can hold addresses in registers `A2-A4`.

If the 68881 option in the Code Generation section of the **Options...** dialog is on, THINK C uses floating point registers for floating point variables declared `register`. You can declare up to five floating point register variables. See "Code Generation Options" later 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.

Floating point arithmetic

The THINK C type `double` actually corresponds to the SANE type Extended. This type takes up 10 bytes of storage, but is the *fastest* floating point type. Conversely, the THINK C type `float` actually corresponds to the SANE type Single. This type takes up 4 bytes of storage, and is the second fastest floating point type. The THINK C type `short double` actually corresponds to the SANE type Double. This type takes up 8 bytes of storage, but strangely enough is the slowest floating point type. If you want speed, stick with `double`. If you want to save space and are willing to sacrifice some accuracy and speed, then use `float`.

Precompiled Headers

THINK C lets you "precompile" headers (#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. When the MacHeaders option is on, THINK C automatically loads `MacHeaders`, so you never have to explicitly #include common header files like `QuickDraw.h`. (It doesn't hurt if you do, but it slows down compilation.) See "Code Generation Options" below to learn how to turn the MacHeaders option on and off.

The MacHeaders file contains these files:

```
ControlMgr.h
DeskMgr.h
DeviceMgr.h
DialogMgr.h
EventMgr.h
FileMgr.h
FontMgr.h
HFS.h
IntlPkg.h
ListMgr.h
MacTypes.h
MemoryMgr.h
MenuMgr.h
OSUtil.h
PackageMgr.h
QuickDraw.h
ResourceMgr.h
ScrapMgr.h
SegmentLdr.h
StdFilePkg.h
TextEdit.h
ToolboxUtil.h
WindowMgr.h
asm.h
pascal.h
```

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

```
Appletalk.h
nAppletalk.h
Color.h
ColorToolbox.h
DeskBus.h
DiskDvr.h
PrintMgr.h
ScriptMgr.h
SCSIMgr.h
SetUpA4.h
SerialDvr.h
SlotMgr.h
SoundDvr.h
SoundMgr.h
StartMgr.h
TimeMgr.h
VRetraceMgr.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.

Open the file Mac #includes.c, and edit it to include any other files or declarations you need. Then, select **Precompile...**from the **Source** menu. After THINK C precompiles the file, save it as MacHeaders. (Precompiled files don't go into the project.)

The auto-make facility marks the files in the current project for recompilation if you change MacHeaders.

To let other projects know that `MacHeaders` has changes, use the **Make...**command in the **Source** menu to mark all the `.c` files, then click on the Make button to recompile all the files.

Creating your own precompiled headers

If you want to use your own precompiled headers, first disable THINK C's automatic loading of `MacHeaders`. Use the Code Generation radio button in the **Options...** dialog from the **Edit** menu to turn this feature off (see "Code Generation Options" below).

Create a file containing the desired series of #include statements, and choose the **Precompile...** command from the **Source** menu. When THINK C is through precompiling, save the file.

> **Note:** You can use `#include <MacHeaders>` as the first line of your precompiled header.

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 the MacHeaders option is enabled, you can't explicitly include any other precompiled header.)

When the MacHeaders option is off, you can use serval 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.

Calling the Macintosh Toolbox Routines

THINK C knows about all the routines in *Inside Macintosh I-V* 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.

THINK C knows how many arguments every Toolbox routine takes and the sizes of the arguments. If you supply a wrong size integer (for instance, an `int` for a `long`), THINK C adjusts the argument automatically. If you pass the wrong size non-integer argument (an `EventRecord` instead of a pointer to an `EventRecord`, for example), THINK C displays an error dialog.

THINK C knows the sizes of the return values but not their types, so it assumes the values are integers. (Of course, for functions that do not return a value the return type is `void`.) The #include files supplied with THINK C contain declarations that specify the correct return type. For example,

```
extern pascal Handle GetResource();
```

ensures that the result from `GetResource` will be considered a `Handle`. Without this definition the result would be considered a `long`.

Most Macintosh calls are implemented by traps that use the Pascal calling conventions directly. THINK C generates these traps inline instead of generating a subroutine call. For register based traps or routines marked `[Not In ROM]`, THINK C generates calls to library functions in `MacTraps`. Calls to these routines also follow Pascal calling conventions.

Passing arguments to Toolbox routines

Since the argument declarations in Inside Macintosh are given in Pascal, you have to know how to convert them to C. You don't need to know the details of C and Pascal calling conventions to call the Toolbox routines, but it helps; see Chapter 12. This table shows you the general rule for converting argument declarations in 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 parameter declarations and their C counterparts:

Pascal Type	**C Type**
`INTEGER`	`int`
`LONGINT`	`long`
`CHAR`	`int`
`BOOLEAN`	`char`
`Byte`	`Byte` or `unsigned char` in `struct` declarations, `int` when passed as an argument.
`VAR Byte`	`int *`
`OSType, ResType`	`long`
`PACKED ARRAY [1.4] OF CHAR`	`long`
`String255`	`String255` or `char *`
`VAR String255`	`String255` or `char *`
`StringPtr`	`StringPtr` or `char *`
`VAR StringPtr`	`StringPtr *` or `char **`
`Point`	`Point`

```
VAR Point                    Point *
```

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 your 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 largest Pascal string is 255 bytes.

> **Note:** Pascal strings are *not* null terminated.

Use the routines CtoPstr() and PtoCstr() convert back and forth from Pascal strings to C strings. These routines convert the strings in place, and return the converted string. These are their function prototypes:

```
char *CtoPstr(char *s);
char *PtoCstr(char *s);
```

> **Note:** If you turn the MacHeaders option off, be sure to #include pascal.h when you use these functions.

Some Macintosh Toolbox routines use the Pascal type PACKED ARRAY[1.4] OF CHAR to specify resource types and file types. Although this is an array type in Pascal, don't treat it as an array in C. Since it is only 4 bytes long it is passed by copying it onto the stack, and should be declared in C as a long. THINK C lets you specify multi-byte character constants like 'STR#' or 'TEXT' for this purpose.

The QuickDraw data type Point is only 4 bytes long, so it's passed by copying it onto the stack. Don't pass its address unless it is a VAR parameter. Most RECORD (struct) types, however, are larger than 4 bytes, so you would have to pass their address.

Special cases of Toolbox routines

Because the Macintosh Toolbox is written in Pascal, some functions work differently when you call them from THINK C. Fortunately, the number of cases is very small.

The Fix2X() and Frac2X() functions (see *Inside Macintosh IV*, Chapter 12, "Toolbox Utilities") are defined as returning Extended. In THINK C, these functions return double. These two functions are the only Toolbox functions that use C calling conventions. They are not declared pascal. See ToolboxUtil.h for their declarations.

The List Manager function `LLastClick()` returns `long` instead of `Cell` as documented in *Inside Macintosh IV*, Chapter 30, "The List Manager."

The "old" File Manager data structuresdescribed in *Inside Macintosh II*, Chapter 4, "The File Manager" are defined in `FileMgr.h`. The "new" data structures described in *Inside Macintosh IV*, Chapter 19, "The File Manager" are defined in `HFS.h`. The "new" definitions have an `H` prepended to the data structure name. For instance, the "new" `VCB` structure is called `HVCB`.

The routines `SetUpA5()` and `RestoreA5()` described in *Inside Macintosh II*, Chapter 13, "The Operating System Utilities" are provided as macros instead of as functions in `MacTraps`. They are defined in `OSUtil.h`.

Using the RAM serial driver routines

In the 64K ROMs, your application needs a `SERD` resource to use the RAM serial driver described in *Inside Macintosh II*, Chapter 9, "The Serial Drivers." This resource was incorporated into later ROMs. If your application uses the serial driver, and you want it to run on 64K ROM machines, you'll find the `SERD` resource in the file `SERD` in the Mac Libraries folder. Use ResEdit to copy the resource into your application's resource file.

Calling AppleTalk routines

If your application uses AppleTalk, you should be aware that there are now 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` and the `#include` file `AppleTalk.h`.

Interfaces to the preferred set are in the `MacTraps` library. The `#include` file `nAppletalk.h` defines data structures necessary when using these calls.

To use this interface...	**Use this library / header**
preferred	`MacTraps / nAppleTalk.h`
alternate	`AppleTalk / AppleTalk.h`

Remember, if you are using only the preferred calls, there's no need to load the `AppleTalk` library into your project.

Writing 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 conven-

tions. (To learn about the difference between Pascal calling conventions and C calling conventions, see Chapter 13.)

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 you're writing a function that behaves 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 supposed to be passed by value, not by reference. If the size of a non-VAR parameter is greater than 4 bytes, you'll need to pass its address, but you may not modify the parameter.

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 (theDialog, theEvent, itemHit)
    DialogPtr theDialog;
    EventRecord *theEvent;
    int *itemHit;
```

The call to `ModalDialog()`, then, looks like this:

```
extern pascal Boolean MyFilter();
int 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 function, you can call the function by

using one of the library functions `CallPascal()`, `CallPascalB()`, `CallPascalW()`, or `CallPascalL()`.

For example, suppose that `pC` is a pointer to a C function and `pP` is a pointer to a Pascal function; both functions accept two `int` arguments and return `void`. You can declare both pointers the same way:

```
void (*pC)(), (*pP)();
```

To call the function pointed to by `pC` you would write:

```
(*pC)(5, 7);
```

But to call the the function pointed to by `pP` you write:

```
CallPascal(5, 7, pP);
```

Note that the pointer to the function must be the last argument to `CallPascal()`. When you use these routines, be aware of the different return types.

> **Note:** If you turn the MacHeaders option off, be sure to #include pascal.h when you use these functions.

To call a Pascal...	**Use...**
`PROCEDURE`	`CallPascal(arg1, arg2, ..., fp)`
`FUNCTION returning BOOLEAN`	`CallPascalB(arg1, arg2, ..., fp)`
`FUNCTION returning INTEGER, CHAR`	`CallPascalW(arg1, arg2, ..., fp)`
`FUNCTION returning LONGINT, Ptr, Handle`	`CallPascalL(arg1, arg2, ..., fp)`

Accessing low memory globals

The #include files (or `MacHeaders`) define most of the low memory globals referenced in *Inside Macintosh*. THINK C provides a way to define additional low memory globals:

```
extern int MemErr : 0x220;
extern char FinderName[] : 0x2E0;
```

The address you provide cannot be greater than 0xFFFF.

Tips

`SetRect`, `SetPt`, etc., take more time than setting up the coordinates yourself, because of the overhead of the Macintosh trap dispatcher. However, you'll notice significant improvements only if you are executing lots of them.

If you are dereferencing Handles, make sure that the memory the Handle points to won't move while using it. Otherwise, HLock it (and HUnlock it later). Example:

```
HLock(aTEH);
/*  Without the previous HLock, */
/* the next two calls may   not always work*/

/*(1) printf calls Memory Mgr: */
printf("first character in text handle is %c\n", **(**aTEH).hText );

/*(2) Handle dereference on the left is done BEFORE call:*/
(**aTEH).hText = (Handle)NewHandle(somesize);

HUnlock(aTEH);
```

To increase your zone (the memory available to your program) to the maximum, call the procedure MaxApplZone().

Don't forget to make all necessary Macintosh initialization calls. In THINK C, you must make all the necessary calls yourself.

You don't need to call the initialization routine when you use the stdio library. Initializations are automatically done the first time a stdio procedure (that needs it) is called. You can turn this off if you're doing your own initializations — for example, if you are using the standard output window for debugging. The automatic initialization is for users of THINK C who want to have the more traditional, non-Macintosh C environment.

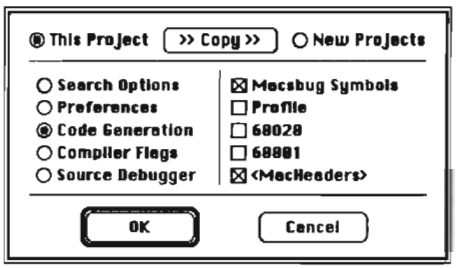
The Byte and Char types in Pascal actually correspond to the int type in C. The three places that this causes problems are the functions ATPOpenSocket(), GetItemMark(), and GetItemIcon(). In these functions, a Byte argument is passed by reference. The proper way to do this in C is to pass a pointer to an int.

Code Generation Options

THINK C lets you control some aspects of code generation. The simplest way you can affect code generation is by using register variables. You can also use the **Options...** command in the **Edit** menu to instruct THINK C to generate symbols for assembly language debuggers, to generate special code for the code Profiler (see Appendix A), and to generate code for the 68020 CPU or the 68881 coprocessor. You can also use the same dialog to tell THINK C whether to use the precompiled MacHeaders (described above).

Setting the options

To set the code generation options, choose **Options...** from the **Edit** menu. Click on the Code Generation radio button, and you'll see this dialog box:



Chapter 14 talks about the **Options...** command in detail.

Using Macsbug symbols

When the Macsbug Symbols options is set, THINK C generates symbols for assembly language level debuggers such as Macsbug or TMON . THINK C generates symbols only for functions that have stack frames, so functions without arguments and local variables don't get symbols. Be aware that while Macsbug symbols are useful for debugging, they add 8 bytes to every procedure. This option is on by default.

Using the Profile options

When the Profile option is set, THINK C generates calls to code profiler routines. The code profiler collects timing statistics about your functions. See Appendix A to learn more about the code profiler. This option is off by default.

Generating 68020 code

If the 68020 option is checked, THINK C uses the 68020 instructions for bitfield operations and long word multiplication, division, and modulo operations.

> **Note:** When you use the 68020 option, it's up to you to make sure your application is running on a Macintosh with a 68020 processor.

Generating 68881 code

If the 68881 option is checked, THINK C generates inline code for the floating point coprocessor. Up to five local variables of type `double` may be declared `register` and will be placed into 68881 registers.

When this option is on, the size of double variables is 96 bits. 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 68881.

If you have this option set, you'll need to use the 68881 version of the math library.

Note: When you use the 68881 option, it's up to you to make sure your application is running on a Macintosh with a 68881 processor.

Using the MacHeaders option

When the <MacHeaders> option is on, THINK C will automatically include the `MacHeaders` file for every file in your project. The `MacHeaders` file contains the declarations for the most common Macintosh Toolbox types, functions, and low memory globals. Since these declarations are in binary form, compilation is faster than if you included the header files manually. It doesn't hurt to include header files already included in `MacHeaders`, like `QuickDraw.h`, but compilation will be a bit slower.

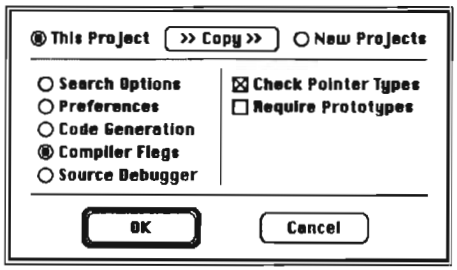
If you want to use your own precompiled headers, make sure this option is turned off. THINK C allows only one precompiled header per source file.

To learn more about precompiled headers, see the Precompiled Headers section in this chapter.

Compiler Options

THINK C lets you specify how strictly it should enforce type checking. You can set the compiler options to make sure that THINK C checks that pointer types are assignment compatible. You can also enforce some strict type checking by requiring that every function have a function prototype. (To learn about function prototypes, read the next section.)

To set the compiler options, choose **Options...** from the **Edit** menu, and click on the Compiler Flags radio button. You'll see this dialog box:



Chapter 14 talks about the **Options...** command in detail.

Check Pointer Types

When the Check Pointer Typesoption 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.

Require Prototypes

When the Require Prototypes option is on, THINK C forces very strict type checking: You can't use or define a function unless it has a prototype. (Macintosh Toolbox routines don't need prototypes even if this option is on.) Read the next section to learn about function prototypes.

Function Prototypes

A function prototype is a function declaration with additional information that describes the arguments the function takes. For example:

```
extern int strcpy(char *dest, char *source);
extern int printf(char *, ...);
```

Argument identifiers are optional and are ignored if supplied. You can specify functions with a variable number of arguments with ellipsis (...) as in the `printf` example above. The ellipsis must appear at the end of the argument list. No type information is provided about the additional arguments.

A function declaration with no argument information, e.g.

```
extern long foo();
```

is *not* a prototype and supplies no information about the arguments. It only declares the return type. To write a prototype specifying that the function takes no arguments, use

```
extern long foo(void);
```

(This is a special case and does *not* mean that the function takes a `void` argument.)

Prototypes are optional unless you have checked the Require Prototypes option. If you use prototypes, they must appear before the first definition or use of the function. The best place for prototypes is in #include files, so other files in your project will be aware of them.

If a prototype is in effect when a function is called, the actual arguments are checked against the prototype. Unless the prototype ends in ellipsis (...), the number of arguments must match exactly. The argument types must be assignment-compatible (the state of the Check Pointer Types option is honored). Appropriate conversions are applied to arithmetic (integral and floating-point) values. Additional arguments allowed by the ellipsis are not checked or converted.

If a prototype is in effect when a function is defined, the definition is checked against the prototype. Unless the prototype ends in ellipsis (...), the number of arguments must match

exactly. The types of the arguments must be identical. Additional arguments allowed by the ellipsis are not checked.

If no prototype is in effect when a function is called or defined, the "null" prototype (...) is assumed. However, if the Require Prototypes option is checked, THINK C displays an error message. (THINK C never requires prototypes for the Macintosh Toolbox routines, but you can supply them if you prefer.)

You can supply prototype information for a Macintosh Toolbox routines. THINK C checks the prototypes against the built-in size and count information, and will subsequently use the prototype in preference to the built-in information. For example:

```
extern pascal Handle GetResource(OSType, int);
```

Full prototype information is not built in for the Macintosh calls.

A function definition is not itself a prototype, so the following example won't work:

```
f(x)
long x;
{
    ...
}
g()
{
    ...
    f(0);
    ...
}
```

The `0` argument will be passed to `f()` as an `int`, not a `long`. To get automatic type coercion, you must supply this prototype before you call the function `f()`:

```
int f(long);
```

Portability

It is sometimes said — not entirely tongue-in-cheek — that the best feature of the C programming language is its portability while its worst feature is its lack of portability!

The American National Standards Institute (ANSI) is finalizing the standard definition of the C language. The standard is expected to be approved in late 1988. To learn more about the ANSI definition of C, see the second edition of Kernighan and Ritchie 's *The C Programming Language* (Prentice Hall).

THINK C follows the "stable" parts of the ANSI standard, and you can expect that it will follow the standard soon after it's released.

A section is devoted to Unix compatibility issues; comments made there may also be applicable to compilers designed for other non-Macintosh environments such as the IBM PC.

Predefined preprocessor symbol

THINK C predefines a preprocessor symbol, `THINK_C`, so you can test for THINK C when doing conditional compilation.

Sizes of numbers

The most common way C compilers differ is in the sizes of the three integer types, `short`, `int`, and `long`. There seems to be general — though by no means universal — consensus that:

1. short should be the shortest integer type bigger than char supported by the hardware
2. `long` should be the longest integer type supported by the hardware
3. `int` should be the "most natural" integer type supported by the hardware

On the 68000 this means that `short` should be 2 bytes and `long` should be 4 bytes, and most compilers do it this way. The correct size for plain `int` is more problematic. The 68000 has 32-bit registers, so some compilers (including MPW) implement 4-byte `int`s. But the 68000 has a 16-bit data bus and a 16-bit ALU, so 16-bit operations are considerably more efficient than 32-bit operations. Furthermore, Macintosh applications are often pressed for memory, and 2-byte `int`s use a lot less space than 4-byte `int`s, so some compilers implement 2-byte `int`s. THINK C implements 2-byte `int`s.

If you are porting from a compiler which has different integer sizes, and you have code which relies on those sizes, you'll have to do some conversion. The most common case is code that assumes that `int` and `long` are the same size. Here is an example:

```
foo()
{
    long a, b, result;
    result = baz(a, b);
}
baz(i, j)
{
    return(i + j);
}
```

This code works fine when `int` and `long` are the same size, but it is not portable. It won't work in THINK C (or in many other compilers). Either `i` and `j` should be declared `long`, or `a` and `b` should be cast to `int` before being passed to `baz`.

It is less common for a program to make specific assumptions about the sizes of floating-point numbers, but you should be aware that these also tend to differ between compilers. For maximum accuracy and efficiency, THINK C uses the SANE or 68881 extended-precision type to implement the C type `double`. Other compilers for the Macintosh either don't provide the extended-precision type at all, or add a new largest type.

Passing a Point as an argument

Because the QuickDraw `Point` is a structure, some compilers require that you pass the address of a `Point` instead of the `Point` itself. They require, for example,

```
result = PtInRgn(&aPoint, aRgnHandle);
```

even though `PtInRgn` expects the actual `Point` as its first argument.

In THINK C, the above statement would cause `aPoint`'s address to be passed to `PtInRgn`; the correct call is:

```
result = PtInRgn(aPoint, aRgnHandle);
```

Some Toolbox functions expect a `Point` to be passed as a `VAR` parameter; in such cases an address must be passed. For example,

```
SetPt(&aPoint, horizontal, vertical);
```

would be correct in THINK C as well as in other compilers.

Converting from Unix

The primary issue involved in porting code from a Unix or Unix-like environment to THINK C is library support for standard Unix functions. THINK C comes with a robust set of Unix compatibility libraries that provide many of the functions found on the most popular versions of Unix. Some standard Unix functions that would have no meaning on the Macintosh have been omitted, and Unix systems themselves vary in the libraries they support, so you may encounter occasional problems. We've tried to keep them to a minimum.

The standard libraries are described in the *Standard Libraries Reference*.

A useful Unix feature not available in the standard Macintosh environment, the command line, is supported by THINK C's libraries. To use this feature, name your main function `_main` instead of `main`, and add the source file `Unix main.c` to your project. This file contains an implementation of `main` which prompts for a command line when your program begins execution and calls `_main` with Unix-style `argc` and `argv` parameters. Redirection

of the standard input, standard output, and standard error channels is supported using the following conventions:

```
<    redirect stdin to the file name that follows
>    redirect stdout to the file name that follows
>>   redirect stderr to the file name that follows
```

Some Unix implementations use >> to cause the standard output to be appended to the file name that follows. You can easily modify `unix main.c` so that it behaves this way instead. Just change:

```
...
else if (*cp == '>') {
    mode = "w";
    filename = true;
    if (*++cp == '>') {
        file = stderr;
        cp++;
    }
    else
        file = stdout;
}
...
```

to:

```
...
else if (*cp == '>') {
    mode = "w";
    filename = true;
    if (*++cp == '>') {
        mode = "w+";
        cp++;
    }
    file = stdout;
}
```

The Debugger
11

Introduction

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 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 file `THINK C Debugger` is in the same folder as THINK C. If you followed the installation instructions , it should be in the folder named `THINK C Folder`.

To use the source debugger you need at least 2Mb of memory, and you must be running THINK C under MultiFinder. The debugger works only with application projects. It won't work with code resources or device drivers. To use the source debugger to debug desk accessories, use the file `DA main.c` in the disk THINK C 1 and follow the instructions there.

Topics covered in this chapter:

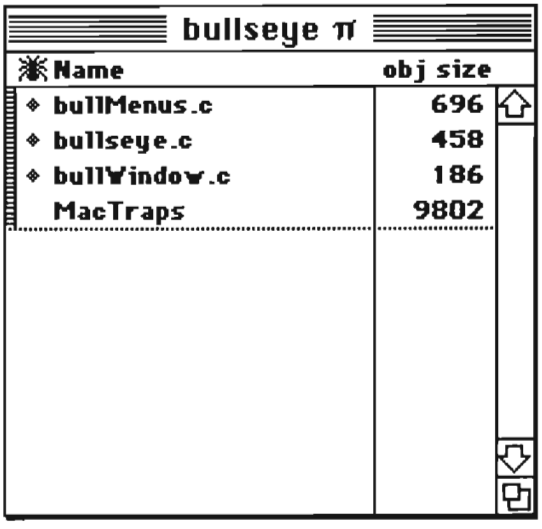
- Running with the debugger
- The debugger windows
- Working with the Source window
- Setting breakpoints
- Controlling execution
- Working with the Data window
- Using low level debuggers
- Quitting the debugger
- Memory considerations

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

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.



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 tables for libraries or projects used as libraries.

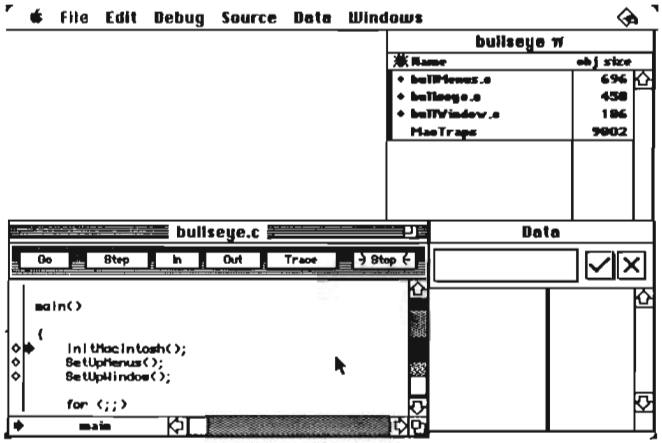
Clicking in the "bug" column next to a file name turns the gray diamonds on and off. If you hold down the Option key as you click on a gray diamond, THINK C removes the gray diamonds from every file. If you hold down the Option key as you click in the "bug" column where there isn't a gray diamond, THINK C turns the diamonds on for every file in the project.

> **Note:** Another way to turn the debugger on is to check the Use Debugger checkbox in the Source Debugger section of the **Options...** dialog.

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 II with two screens, and you have the Use 2nd Screen option on, the two debugger windows appear in the second screen.



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.

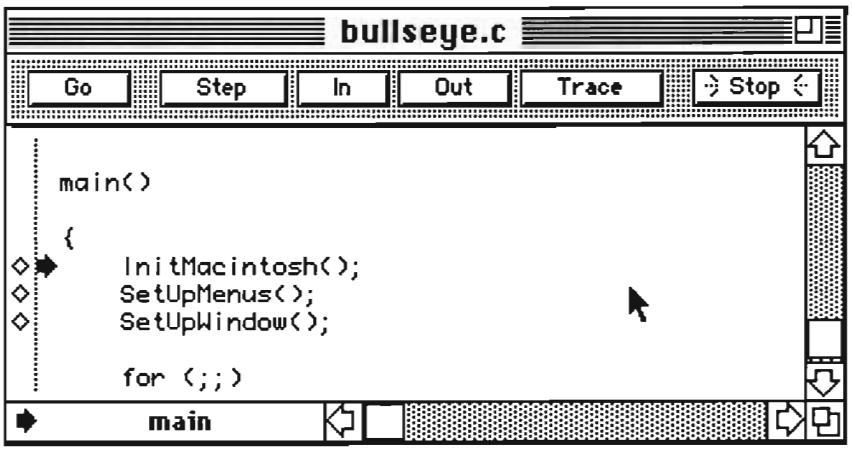
> **Note:** Unlike other Macintosh applications, the debugger doesn't use the **File** menu, so it's always dimmed. When your screen is cluttered with a lot of debugger and application windows, the dimmed **File** menu tells you instantly that the debugger's windows are active.

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'll need to be aware whether its your application or the source debugger is 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.

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.



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 "Working With the Source Window" to learn how to see other files in your project in the source window.

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 "Controlling Execution" below to learn how to use the status panel and the **Debug** menu commands to step through your code.

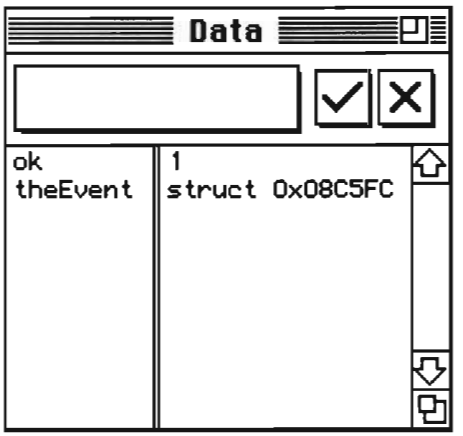
The column of diamonds running along the left side of the source text are **statement markers**. Every line of your program that generates code gets a statement marker. You can set breakpoints only at statement markers. See "Setting Breakpoints" to learn how to set and clear breakpoints.

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.

The source debugger uses the 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.



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.

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 is read only. You can select text in 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`.

If the debugger can't find a source file as it was when it was last compiled, the debugger displays the message `Source text not available`. (See "Editing while debugging" below.) This will happen, for instance, when the debugger can't find a source file or when a source file has been edited.

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.

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 is called the **selected statement**.

> **Note:** All commands that work on the selected statement operate on the current statement if you didn't select any other 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 from register `A6`, so if a function doesn't generate a stack frame, you won't see it in the call chain.

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:

- Click on the THINK C project window (Or choose your project name from the **Windows** menu)

- Click on the name of the file you want to look at
- Select **Debug** (Command-G) from THINK C's **Source** menu

The source debugger's Source window will display the text of the file you chose. 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.

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 will cache 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.

- Choose **Edit 'filename.c'** from the **Source** menu (or press Command-E) to open an edit window for the file in the Source window.
- Use the THINK C **Find...** command to find what you're looking for.
- Choose **Debug** from the THINK C **Source** menu (or press Command-G) to get back the source debugger. The line containing the selection in the editor window is displayed in Source window.

When the Source window displays what you're looking for, you'll usually want to use the **Go Until Here** command. See "Controlling Execution" later on to learn about this command.

Setting Breakpoints

The THINK C debugger lets you set breakpoints at any line that has a 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.

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.

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 all the breakpoints.

> **Note:** You can also use the **Set Breakpoint** and **Clear Breakpoint** commands in the **Source** menu to set and clear breakpoints. Select a line in the source window by clicking once on the line. Then choose **Set Breakpoint** or **Clear Breakpoint** from the **Source** menu.

Setting temporary breakpoints

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.

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.

The **Go Until Here** command (see "Controlling Execution" below) sets a temporary breakpoint at the selected line.

Setting conditional breakpoints

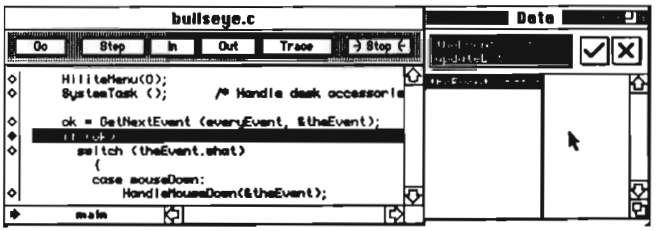
The THINK C debugger lets you set conditional breakpoints. A conditional breakpoint is a breakpoint has a condition associated with it. The debugger stops execution at conditional breakpoints only when the condition evaluates to non-zero.

To set a conditional breakpoint:

- Set a breakpoint by clicking on the statement marker diamond
- Click on the line to select it

- Click on an expression in the Data window
- Choose **Attach Condition** from the **Source** menu.

When you set a conditional breakpoint, the statement marker turns to a gray diamond



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 will be 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:

- Click on the line to select it
- 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 "Examining Variables" below.

Controlling Execution

The debugger has six commands that control execution. To make it easier to debug your programs, there 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.

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.

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 `JSR`s.)

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. (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 Breakpoints" above) at the selected line. Use this command when you want to move through a block of code quickly.

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. See "Searching in the Source window" above.

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.

> **Note:** 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 be 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 when the program stops. To trace through every line of your program automatically, for example, hold down the Option key as you click on the Trace button.

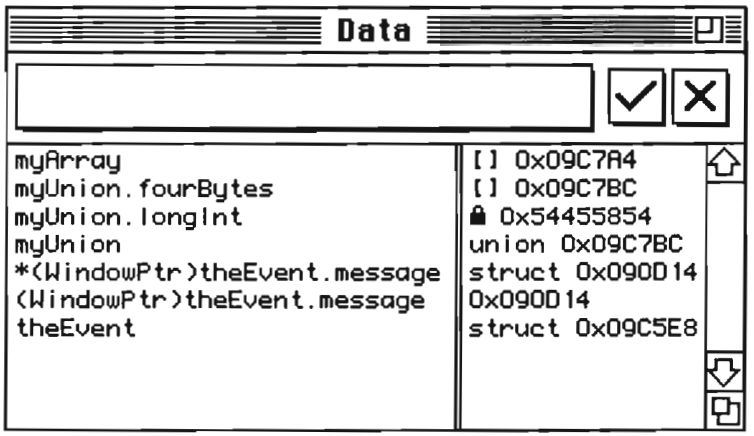
One useful technique is to set breakpoints at spots where you'd like to look at some variables and then do an Auto-Go. When your program hits the breakpoint, the debugger will update the Data window and start the program up again.

To cancel auto mode, type Command-Shift-Period or click on the Stop button in the status panel. The debugger will finish the command and then stop. Note that if you were doing an

Auto-Go, 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 left column, and its value is displayed in the right column. You can display values in several formats to make your debugging easier. You can also display structs and arrays in their own windows.



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, within the context of the current statement. If the expression you enter is not defined within the context, the debugger will display an error message as the value of the expression.

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.

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`
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`.

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.

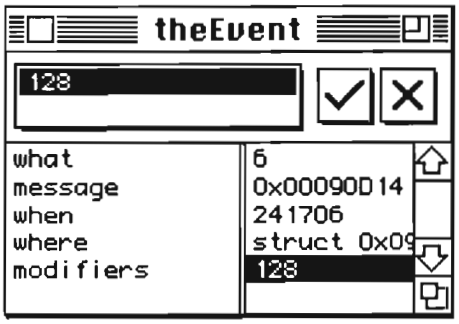
Double-clicking on a value formatted as Pointer enters a new, dereferenced, expression in the Data window. If you hold down the Shift key, the new expression will be dereferenced twice.

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.



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.

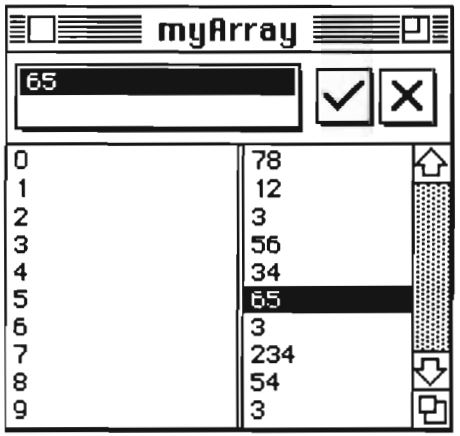
> **Note:** All struct and array windows "belong" to the main Data window. For every struct or array window there is an entry in the main Data window.

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 a 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" scrollbars.



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.

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 correct values. The PC (program counter), however will not be pointing to the next instruction. Instead, it will be pointing somewhere in the debugger.

> **Note:** If you don't have a low level debugger installed, don't use the **Monitor** command.

Using the Monitor command with TMON

If you're using TMON, 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 Macsbug

If you're using Macsbug, the correct value of the PC is right before the current PC. 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 Macsbug.

Use Macsbug's G command to return to the THINK C debugger.

Leaving the low level debugger

If you entered the low level debugger through the Monitor command, use your debugger's standard exit function: Exit in TMON, G in Macsbug.

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 enough to ensure that you can run all three with enough space left over for the Finder and the System on 2Mb of memory.

The default partition sizes are:

Application	Partition Size
THINK C	700K
debugger	200K
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.

When you're running under MultiFinder, your application doesn't need to reserve memory for the system heap, so you can set the project partition size to be quite small. For moderate size projects you might want to try values like 128K.

THINK C uses the most memory when its 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 500K.

If you have the Update Windows debugger option on (Source Debugger options under the **Options...** command in the **Edit** menu), the debugger partition might need to be bigger than 200K. 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 150K.

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.

Assembly Language
12

Introduction

This chapter tells you how to use assembly language in your THINK's LightspeedC 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.

Topics covered in this chapter:

- Using the inline assembler
- C calling conventions
- Pascal calling conventions
- Tips

Using the Inline Assembler

THINK C lets you use assembly language within your source files. The THINK C **inline assembler** works within the compiler to produce object code. You can refer to C variables and functions within assembly language routines. Your C routines can `goto` labels in the assembly routines and vice versa.

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 even if they appear within assembly language.You can use a C constant expression wherever a constant would be legal.

The inline assembler supports all the standard 68000instructions. (68020 and 68881 instructions are not supported.) 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 assembler is not case sensitive with respect to instruction mnemonics, register names, and size specifications (`.B`, `.W`, `.S`, `.L`).

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) 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.

> **Note:** C identifiers which conflict with register names (`D0-D7`, `A0-A7`, `SP`, `USP`, `SR`, and `CCR`) cannot be referenced by name in inline assembly.

You can reference fields of a `struct` directly. Use the `OFFSET()` macro in the #include file `asm.h` to get offsets of fields of a structure. For example, if you have a variable of type `WindowRecord`, you can get the `refCon` field like this:

```
long myRefcon() /*  refCon of myWindow  */
{
    extern WindowRecord myWindow;
    asm {
        move.l  myWindow.refCon,d0
    }   /*  same as "return(myWindow.refCon)"  */
}
```

If you only had a pointer to a `WindowRecord`, you'd use the `OFFSET()` macro to get the `refCon` field:

```
long refcon(wp) /*  same as "GetWRefCon"  */
WindowPtr wp;
{
    asm {
        move.l  wp,a0
        move.l  OFFSET(WindowRecord,refCon)(a0),d0
    }
}
```

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 digits. Colons are optional following assembler labels, but must appear after C labels.

```
foo ()
{
    asm {
    @123…   /* A legal asm label */
    @73:…   /* 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:

```
    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`.)

Do not use the `RTS` instruction unless you're absolutely sure you know what you're doing. Use `return`, or simply fall through to the end of the function. This way, the C stack frame will be cleaned up properly.

If you use the `return` statement, don't specify a return value. Put the return value in the proper place (usually D0) instead. See the following sections for information about C and Pascal calling conventions for more information.

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();

OtherFun()
{
    ...
    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. See the sections below for C and Pascal calling conventions.

> **Note:** A short branch (`BRA.S`) to the immediately following instruction is an error which is *not* detected by the inline assembler. (This instruction generates an 8-bit zero displacement, which results in the next instruction word being used as a 16-bit displacement for a long branch rather than being executed as an instruction. See the 68000 manual for more details.)

Using the Macintosh Toolbox in assembly language

You can use any of the Macintosh Toolbox functions (except those marked `[Not in ROM]`) in your assembly language routines. The inline assembler is case sensitive with respect to trap names. If you like, you can precede trap names with an underscore.

The inline assembler generates one instruction for ROM traps even though some might require glue when you call them from C. If you want to use the glue from assembly language, you'll need to `JSR` to the routine. For example:

```
DemoFun()
{
    asm {
    /* Direct, register based call */
        MOVE.L  #256,D0
        _NewHandle
        /* result in A0, error code in D0 */
        ...
    /* Stack based call through glue */
        CLR.L   -(SP)       /* save space for result */
        MOVE.L  #256,-(SP)  /* push number of bytes */
        JSR NewHandle
        /* Result at (SP), error in MemErr */
    }
}
```

Note: With the 64K ROMs, Memory Manager traps do not set the low-memory global `MemErr`, though the glue does. This is a time when you'd want to use the glue instead of using the trap directly.

You can provide an optional argument, a 2-bit value to be placed in bits 9.10 of the trap to set trap modifier bits, such as `AUTOPOP`, `SYS`, `CLEAR`, and `ASYNC`. The various trap modifier bits are defined in `asm.h`. For example:

```
Handle NewClearSysHandle(size)
{
    asm {
        move.l   size,d0
        NewHandle    CLEAR+SYS
        move.l   a0,d0
    }
}
```

Register usage

You may modify registers `D0`, `D1`, `D2`, `A0`, and `A1`, as well as registers holding register variables. All other registers should be saved and restored as you need them. If you want to use a register other than for scratch purposes, declare a register variable. You'll be able to refer to it by name, and you won't have to bother to save and restore its value.

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 the stack is cleaned up before returning from the function.

See the next sections for more information about C and Pascal calling conventions.

Differences from other assemblers

Omitting a zero displacement in the Address Register Indirect with Index addressing mode — `(A1,D2.W)` instead of `0(A1,D2.W)` — is not supported.

The `DC.B` directive always generates an even number of bytes. A zero pad byte is generated if an odd number of values are specified. For example:

```
DC.B    'a','b','c','d' ; Assembles as four packed bytes
DC.B    'a'             ; These assemble as four words with
DC.B    'b'             ; zero pad in the low byte
DC.B    'c'
DC.B    'd'
```

The syntax `$NNNN` is not available to designate a hex constant. Use the C syntax `0xNNNN` instead.

The difference of two addresses is not a constant expression, so instructions like

```
move.w  #@2-@1,d0
```

are not possible. Similarly, the inline assembler does not support the syntax

```
dc.w    @1-*
```

to assemble the PC-relative offset of the label `@1`. Use

```
dc.w    @1
```

instead. 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
    ...                 ;  ...
```

The alternative syntax for PC-relative addressing — `@1(PC)` instead of `@1, or @1(PC,D0)` instead of `@1(D0)` — is not supported.

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

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
```

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 in order.

The stack looks like this on entry (just after the call):

```
        arg-N
        ...
        arg-1
SP ->   return address
```

The first argument can be found at `4(SP)`. If the function begins with a `LINK A6,#...` instruction, the first argument can also be referred to as `8(A6)`.

All arguments occupy an even number of bytes on the stack. The caller converts a byte argument into a word. To address this argument as a word use `d(SP)`, where `d` is the appropriate offset. To access the argument as a byte, use `d+1(SP)`.

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 this on exit (just after the return):

```
        arg-N
        ...
SP ->   arg-1
```

It is the caller's responsibility to remove the arguments from the stack.

Functions that return struct, union, or double

An alternate method is used 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.)

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 obtain this address and store the result at the location pointed to. The address is considered a hidden argument to the function, and it is 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.

Functions that accept a variable number of arguments

The C calling conventions are designed to make it easy to write functions that take a variable number of arguments. The first argument(s) 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.

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.

```
int minimum(count, x)
int count, x;
{
    int *xp = &x;     /*  pointer to arg list  */
    while (--count) {
        if (*++xp < x)
            x = *xp;
    }
    return(x);
}
```

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 is written in Pascal and 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

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     -(SP)          ;  reserve space for result
MOVE    ...,-(SP)      ;  first argument
...
MOVE    ...,-(SP)      ;  last argument
JSR     function
MOVE    (SP)+,...      ;  result
```

The function's code looks something like this:

```
LINK    A6,#...        ;  (optional)
...
UNLK    A6             ;  (optional)
MOVE    (SP)+,A0       ;  return address
ADD     #...,SP        ;  total size of arguments
MOVE    ...,(SP)       ;  store return result
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 this on entry (just after the call):

```
        space for return value (if any)
        arg-1
        ...
        arg-N
SP ->   return address
```

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 this on exit (just after the return):

```
SP -> result (if any)
```

It is the function's responsibility to remove the arguments from the stack.

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. The result will be *very* different than what you intended.

```
extern int MemErr : 0x220; /* declare MemErr low memory global */
    asm{
        MOVE.W  0x220, D0    ;moves contents of location 0x220
                             ;(MemErr) into D0
        MOVE.W  MemErr, D0   ;same way of writing the above
                             ;symbolically
        MOVE.W  #0x220, D0   ;moves the value 0x220 into D0
        MOVE.W  5, D0        ;WRONG: this will cause an odd
                             ;address error!
        MOVE.W  #5, D0       ;Right
    }
```

Local storage

Use C variables to declare local storage. If you use the directive DC instead to declare storage space, 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 when referring to variables. Example:

```
function()
{
int anInt;
int GetsTrashed;
asm{
    move.L  #3, anInt    ;WRONG: will overwrite
                         ;variable GetsTrashed
    move.W  #3, anInt    ;RIGHT: Word size matches int.
    }
}
```

Libraries
13

Introduction

This chapter shows you how to use and build libraries with THINK's LightspeedC. A library is a collection of compiled code you can use in many programs. Libraries usually contain utility functions or interface functions to the operating system. The `MacTraps` library, for instance, contains the interfaces to the Macintosh Toolbox.

Topics covered in this chapter:

- Using libraries
- Using projects as libraries
- Creating libraries
- Converting object files into libraries

Using libraries

To add a library to a project, use the **Add...** command in the **Source** menu. The library name will appear in the project window. When you add a library to a project, its object size is zero. The object code isn't loaded automatically when you add a library.

You can use the **Load Library** command in the **Project** menu to load a library's object code, or you can rely on THINK C's auto-make facility to load the library for you the first time.

The auto-make facility can tell when a library hasn't been loaded, but it cannot tell whether a library you've added to a project has changed. Libraries are modified outside the project, so if you know a library has changed, reload it with the **Load Library** command.

If you're not sure whether a library has changed, use the **Make...** command to find out. Select **Make...** from the **Source** menu, and click on the Use Disk button.

When you press the Use Disk button, THINK C checks the dates of the libraries on disk against the libraries loaded in your project. If the creation date and time of a library on disk is different from when it was loaded, THINK C marks it as needing reloading. Click on the Make button to bring the project up to date. (To learn more about the **Make...** command, see its description in Chapter 14.)

Creating Libraries

THINK C gives you to ways to create libraries. You can either use any THINK C project as a library, or you can create a separate library document. 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.

Projects as libraries

You can use any THINK C project document as a library. 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. (See Chapter 7 to learn about the components of a THINK C project.)

Most of the time, you'll use projects as libraries.

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.

When you use a library in your project, the linker adds all of the object code in the library to your project.

By convention, libraries end in `.lib`.

Converting object files into libraries

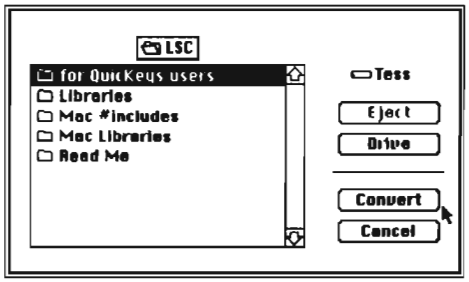
To use object code produced by other compilers and assemblers in THINK C, convert the object files to libraries or projects. Your THINK C package includes two utilities that convert object files.

To convert `.Rel` files produced by Consulair compilers and assemblers into libraries use RelConv. To convert `.o` files produced by Apple's MPW compilers and assemblers into projects use oConv.

Converting .Rel files

The RelConv application converts object files created by Consulair compilers and assemblers into THINK C libraries.

When you double click on the RelConv application, you'll see this standard file dialog:



Double click on the `.Rel` file you want to convert (or click on the Convert button). RelConv generates files with `.lib` suffix. If the name of your object file was `foodle.Rel`, the resulting library will be `foodle.lib`.

When RelConv is finished converting, it will display the standard file dialog again so you can convert more object files. To quite RelConv, select **Quit** from the **File** menu. (You can use the menus even though the standard dialog is active.)

Object files produced by Consulair compilers and assemblers do not retain case information. RelConv assumes that all symbols should be lower case. If you want upper case characters in your symbols, you can supply a vocabulary file.

If RelConv sees a file with the same name as the file it is converting but with a `.voc` suffix, it is considered to be a vocabulary file containing a list of symbols, one per line, with the desired capitalization. Each symbol in the `.rel` file that matches the spelling of a symbol in the vocabulary file will appear with the specified capitalization in the resulting library. Symbols not found in the vocabulary file will appear in lower case.

To help you build a vocabulary file when converting a `.Rel` file for the first time, RelConv, if it finds no vocabulary file, will create one containing all the symbols in the `.rel` file in lower case. You can then edit this file to supply the desired capitalizations and run RelConv again.

RelConv accepts a script file containing a list of `.Rel` files, one per line. The script file must be a text file with an extension of `.rcv`. If relative pathnames appear in the script file, they are interpreted relative to the directory containing the script file.

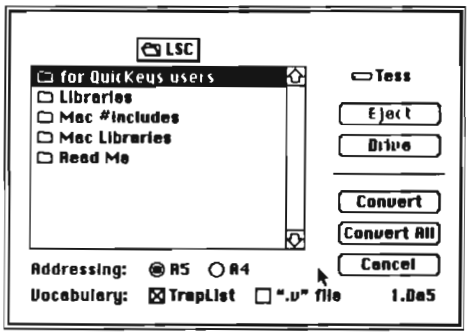
RelConv has an **Options** menu. There is only one option, requesting RelConv to delete each `.Rel` file after converting it.

RelConv will not convert files that were created with the `RESOURCE` directive. This directive is usually used to create resources containing code. THINK C has its own mechanisms for building code resources. See Chapter 7 (You can use the Apple utilities RMaker and ResEdit to create other kinds of resources.)

Converting .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 (see above) or, if you prefer, you can build a library from the resulting project.

When you double click on the oConv icon, you'll see this standard file dialog:



The dialog has some buttons that let you control how the converter builds the project.

The converter displays only files of type `'OBJ '`, with creator `'MPS '`, and that end in `.o`. When you double click on a `.o` file (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 exit the program, 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.

The addressing option only affects those relocatable references where the .o file specifies that PC-relative addressing be automatically substituted as appropriate. Other references to A5 or A4 appearing in the object code cannot be detected; it is the user's responsibility to ensure that they are correct for the kind of program being built. In particular, references to global data may work only with A5-relative addressing.

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 and OS glue routines are added to the vocabulary. This includes all the *Inside Macintosh* routines known to THINK C, except those for which THINK C generates inline traps.

When the ".v" file option is checked, the converter examines a user-supplied vocabulary file for each `.o` file converted. The vocabulary file is a text file that contains the proper capitalization for each symbol, 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.

THINK's LightspeedC

PART FOUR

Reference

THINK C Menus
14

Introduction

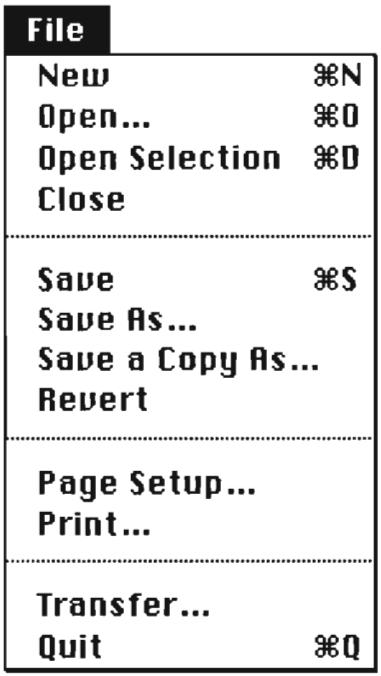
This chapter describes each of the THINK C menu commands. It is organized by menu, from left to right along the menu bar. Within each menu, commands are described in the order in which they appear in the menu.

The Menu

About THINK C...

This command tells you what version of THINK C you're using in an entertaining display. Click the mouse button 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 them 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 you Macintosh owner's manual for details.

Print...

This command lets you print the current file. The standard Print dialog box lets you set the page range among other options. When you press the OK button, your file 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 on an ImageWriter, and from bottom to top on a LaserWriter, to show you the printing progress. To cancel printing , press Command-Period.

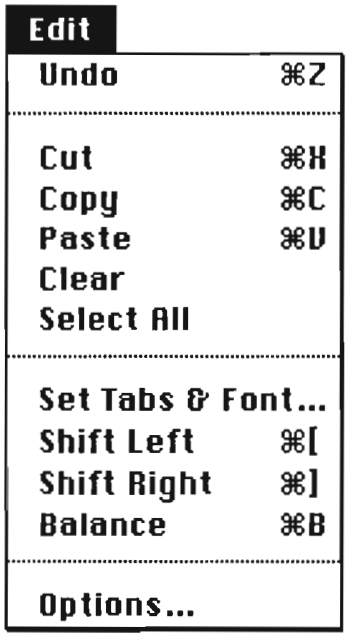
Transfer...

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

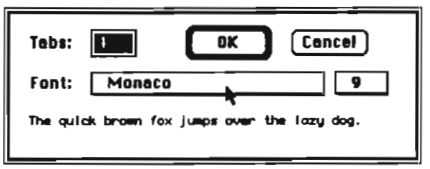
This command clears the selected text. The selection is not placed on the Clipboard. The Clear key on your keyboard has the same effect as the Clear command.

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 this one:



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.

Try this: 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. This is a quick way to check whether all your function definitions are properly balanced.

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...

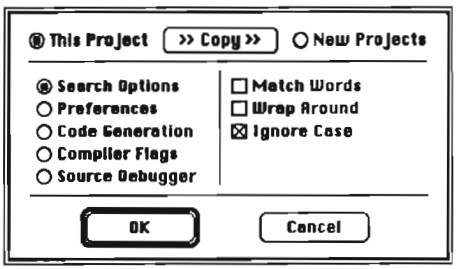
The **Options...** command opens a dialog box that lets you set five groups of options: search options in the editor, project preferences, code generation options, compiler options, and debugger options.

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 section of the options shows up in the dialog at one time, the Copy button copies *all* of the options, even the ones in the sections you can't see.

When you click on the OK button, the changes for all sections of the dialog are saved.

SEARCH OPTIONS

The Search Options section of the **Options...** dialog lets you set the defaults used in the **Find...** dialog. (When you set the same options in the **Find...** dialog, they apply only for the current session.



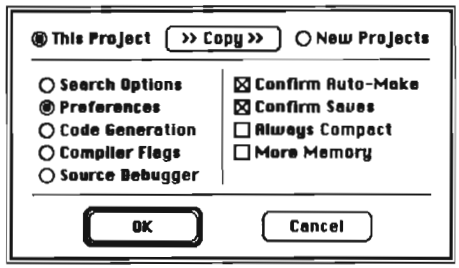
Match Words — When this option is set, the editor's search and replace functions will work on whole words only. For example, a search for "string" will not match the word "strings." This option is off by default.

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 on by default.

PREFERENCES

The Preferences section lets you specify how THINK C behaves when it rebuilds projects, closes files, closes projects, and how it deals with memory.



Confirm Auto-Make 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. This option is on by default.

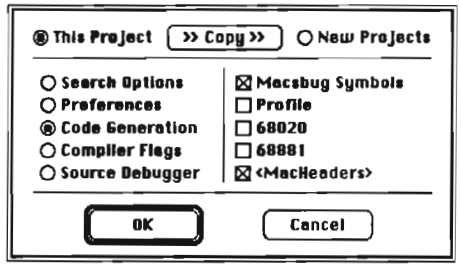
Confirm Saves When this option is off, THINK C automatically saves changes to a file that you have modified without asking if you are sure you want to do so. This is a dangerous option to turn off, since it protects you from inadvertently replacing previous versions of your files with newly modified versions. It is, however, very convenient when you want to do program development in quick, incremental steps. This option is on by default.

Always Compact In order 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. 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**). Compaction may be time-consuming and you will normally want this this option on 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. This option is off by default.

More Memory When this options is on, THINK C tries to find more memory during compilation by discarding certain data structures. This incurs some additional disk activity, since (1) the data structures will need to be read back in when compilation is complete, and (2) if they are "dirty", the data structures must be written out prior to compilation in case the compiler runs out of memory. Leave this option unchecked unless you are developing a large project on a small machine which keeps running out of memory.

CODE GENERATION

The Code Generation section lets you control how THINK C generates code.



Macsbug Symbols When the Macsbug Symbols options is set, THINK C generates symbols for assembly language level debuggers such as Macsbug or TMON. THINK C generates symbols only for functions that have stack frames, so functions without arguments and local variables don't get symbols. Be aware that while Macsbug symbols are useful for debugging, they add 8 bytes to every procedure. This option is on by default.

Profile When the Profile option is set, THINK C generates calls to code profiler routines. The code profiler collects timing statistics about your functions. See Appendix A to learn more about the code profiler. This option is off by default.

68020 If the 68020 option is checked, THINK C uses the 68020 instructions for bitfield operations and long word multiplication, division, and modulo operations.

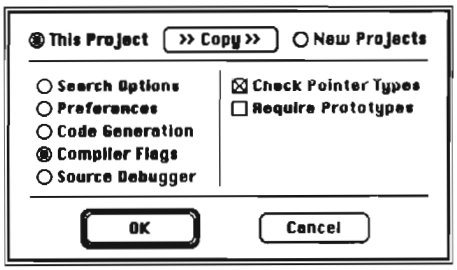
68881 If the 68881 option is checked, THINK C generates code for the floating point coprocessor. Up to five local variables of type `double` may be declared `register` and will be placed into 68881 registers. When this option is on, the size of double variables is 96 bits. Since SANE expects 80 bit doubles, 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 68881.

<MacHeaders> When the <MacHeaders> option is on, THINK C will automatically include the `MacHeaders` file for every file in your project. The `MacHeaders` file contains the declarations for the most common Macintosh Toolbox types, functions, and low memory globals. Since these declarations are in binary form, compilation is faster than if you included the header files manually. It doesn't hurt to include header files like `QuickDraw.h`, but compilation will be a bit slower. If you want to use your own precompiled headers, make sure this option is turned off. THINK C allows only one pre-

compiled header per source file. To learn more about precompiled headers, see the "Precompiled Headers" section in Chapter 10.

COMPILER FLAGS

The Compiler Flags section of the Options... section lets you specify how the THINK C compiler interprets your source files.

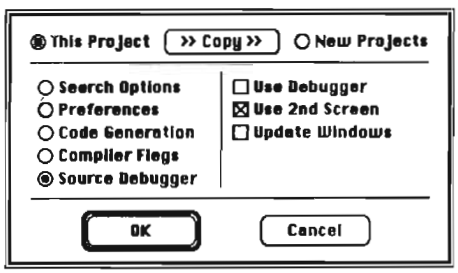


Check Pointer Types When the Check Pointer Types option is on, THINK C makes sure that pointer types match when you assign one pointer to another or when you do pointer arithmetic. If this option is off, THINK C treats all pointers as equivalent types, and won't display the "pointer types do not match" error message. When subtracting two pointers, however, the two types must be pointers to objects of the same size.

Require Prototypes When the Require Prototypes option is on, THINK C forces very strict type checking: You can't use or define a function unless it has a prototype. (Macintosh Toolbox routines don't need prototypes even if this option is on.) Read the "Function Prototypes" section in Chapter 10 to learn about function prototypes. By default, the option is not set.

SOURCE DEBUGGER

The Source Debugger section lets you specify whether to use the source level debugger, where the debugger windows will show up, and how your application's windows will be updated.

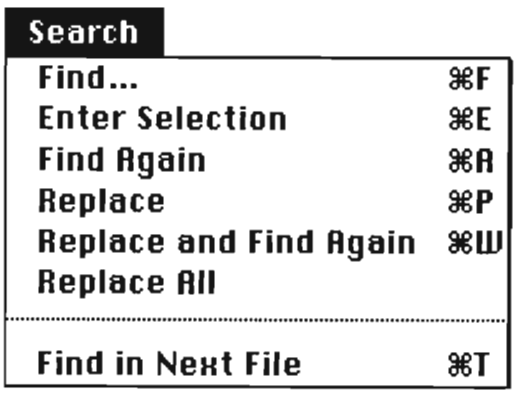


Use Debugger When this option is checked, THINK C launches the source level debugger when you run your project. Checking this option is the same choosing the **Use Debugger** command in the **Project** menu.

Use 2nd Screen When this options is on, and you're running on a Macintosh II with more than one screen, THINK C will display the source debugger windows in the second screen.

Update Windows When this option is on, the debugger tries to update your windows for you when your project is stopped. 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. See Chapter 11 for details.

The Search Menu

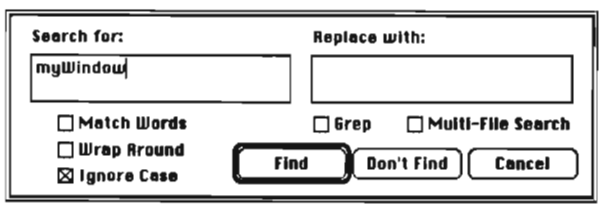


The commands in the **Search** menu let you find and replace strings in your source files. THINK C has extensive search and replace functions including multi-file searching and pattern matching searching. To learn the details of pattern searching, see Chapter 8.

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.



You can also set search options in this dialog box. Note that you can also set these options in the Search Options section 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.

Match Words — 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 seelction 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.

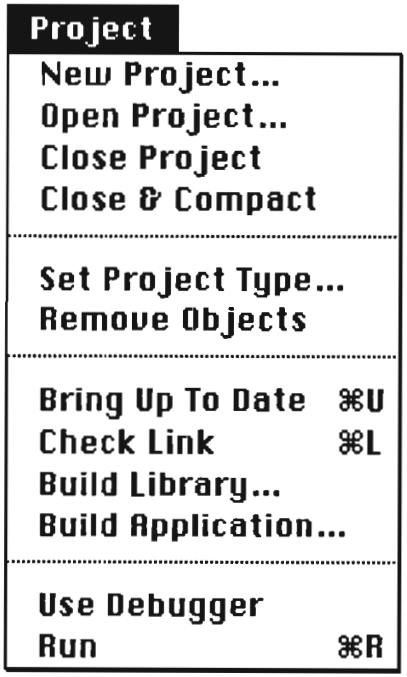
To use this command, you must check the Multi-File Search check box in the **Find...** dialog box. When you check this box, another dialog box displays all of the text files known to THINK C. You can scroll through the list and select individual files by clicking on them to place a checkmark by the name, 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 checkmark.)

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.)

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 whose latest versions have not been saved, a dialog box will ask you if you want to save your edits. To disable this dialog, turn the Confirm Saves option off in the Preferences section 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 (see Chapter 13) or when you plan to transmit the project through a modem (See also the **Remove Objects** command).

Set Project Type...

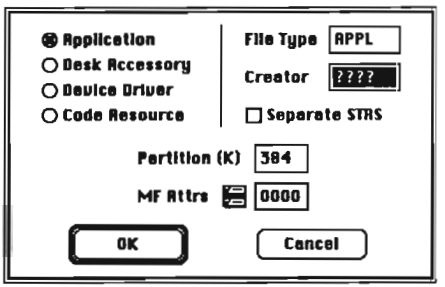
This command lets you set the project type. The default project type is Application, but you can change it to Desk Accessory, Device Driver, or Code Resource. All project types let you specify the File Type and Creator of the file created by one of the **Build...** commands. To learn the details of each project type, see Chapter 7.

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.



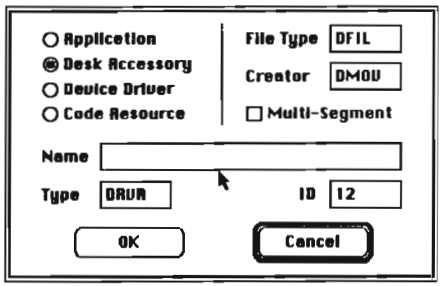
Separate STRS — When this options is set, string literals and floating point constants are placed in their own STRS component. Otherwise, strings and floating point constants are part of the DATA component.

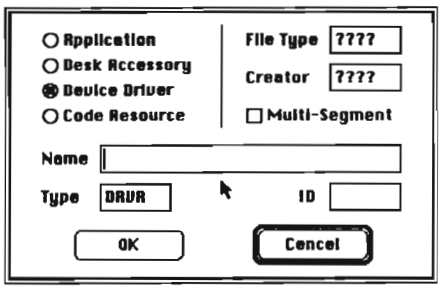
Partition — The value in this field is the amount of memory MultiFinder allocates for your application. The default is 384K, which is more than enough for most moderate size applications.

MF Attrs — The pop up menu lets you set the bits that tell MultiFinder how compatible your application is. You can use the pop up menu or type a hex value into the field.

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. See Chapter 7 for details.





Multi-Segment When this option is on, your desk accessory or device driver can have up to 31 segments.

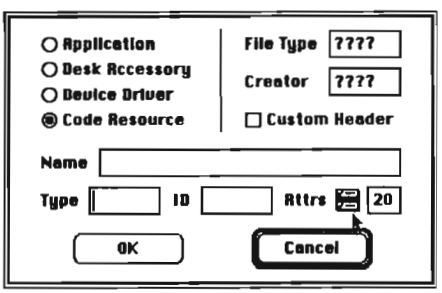
Name This field is the name of your desk accessory or device driver. By convention, desk accessory names begin with a null byte. THINK C takes care of this for you. Device drivers begin with a period. If you don't provide one, THINK C will add one for you.

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.



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.

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.

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.

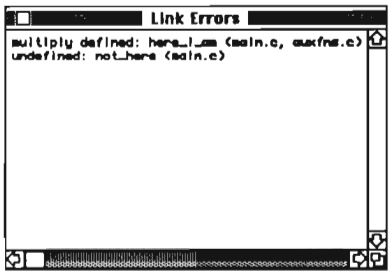
Bring Up To Date

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.



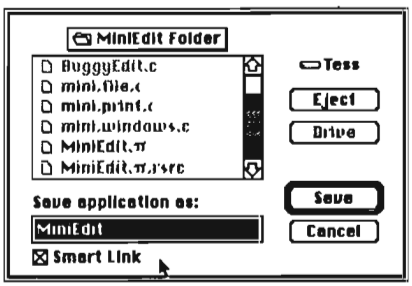
When the **Link Errors** window comes up as the result of an attempt to **Run** or **Build...**, this additional information is not displayed. Since some disk activity is required to compute the information, it is only displayed when you specifically request **Check Link**.

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 13 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:



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 ap-

plication or resource together, but the resulting file will be as small as possible. Uncheck this option if you're building frequently for testing.

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

Use Debugger

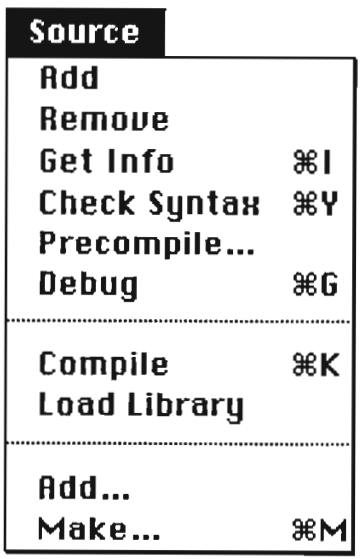
This command turns the source debugger on and off. When the source debugger is on, you'll see a "bug" column in the project window, and when you run the project the debuggers windows appear on the screen. Selecting this command is the same as clicking on the Use Debugger checkbox of the Source Debugger section of the **Options...** dialog box. See Chapter 11 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 pre-compiled headers, to compile source files, to load libraries, and to control the auto-make facility yourself.

Add

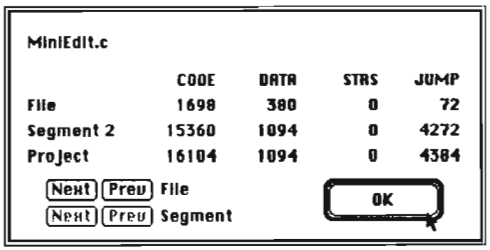
This command adds the file in the frontmost edit window to the project to the project window. The files must end in `.c`.

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.



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`.

Precompile...

This command creates a precompiled header from the contents of the frontmost edit window. Precompiled headers may not have any code or data definitions. 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 13 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.

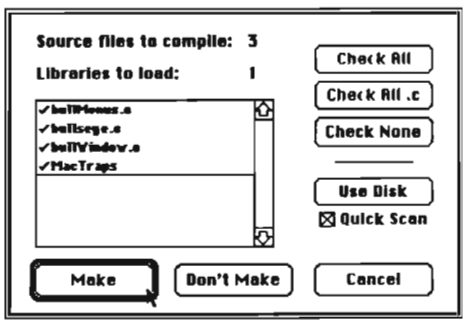
Add...

This command lets you add existing source files and libraries to a project. This command displays a standard file box and lets you select a source file, library, or project to be added to the project. It keeps asking for more until you press the Cancel button. Only files which can be added (i.e., source files that end in .c, libraries, or other projects) appear in the scroll box.

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 the Don't Make button, the project will be updated next time you use any of the commands that the update the project: **Bring Up To Date**, **Run**, **Check Link**, or **Make...**.



If you click the **Use Disk** button and the Quick Scan checkbox 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. Also, if a library file changes, you have to click Use Disk or explicitly check it to let THINK C know.

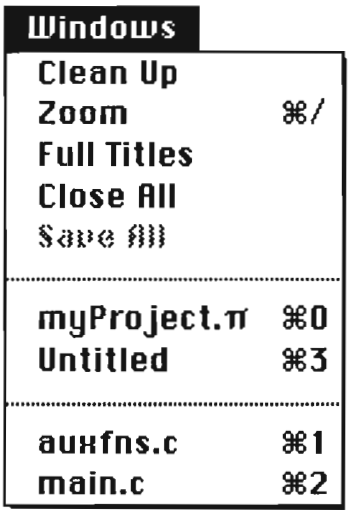
If the Quick Scan checkbox is not checked, THINK C does a more extensive check. If a file can't be found, THINK C searches the tree the file was originally in to find the file. (The Use Disk feature can't help you detect when you've moved files from one tree to another. See Chapter 9 for details.)

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 made (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 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".)

Titled and Untitled 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
15

Introduction

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.

The Menu

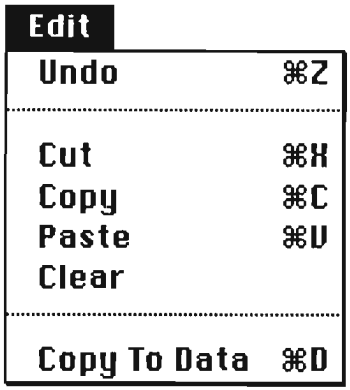
Shortcuts...

This command displays a series of dialog boxes that describe some shortcuts that make working with the debugger faster.

The File Menu

Unlike virtually every Macintosh application, the source level debugger does not have a File menu. This menu is always dimmed. The reason this non-menu still occupies a slot is to let you know at a glance that one of the source level debugger's windows is the active window.

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.

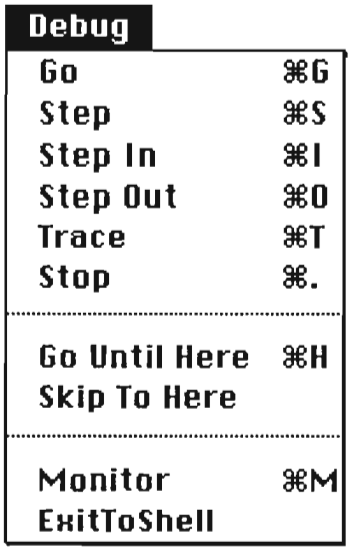
Clear

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 Breakpoints" in Chapter 11) 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.

> **Note:** 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:

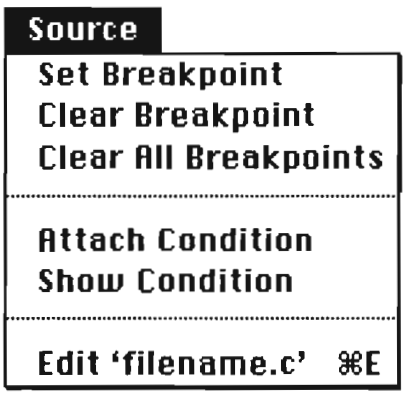
```
DM PC-4
IL @.
```

> **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:

- Set a breakpoint by clicking on the statement marker diamond
- Click on the line to select it
- Click on an expression in the Data window
- Choose **Attach Condition** from the **Source** menu.

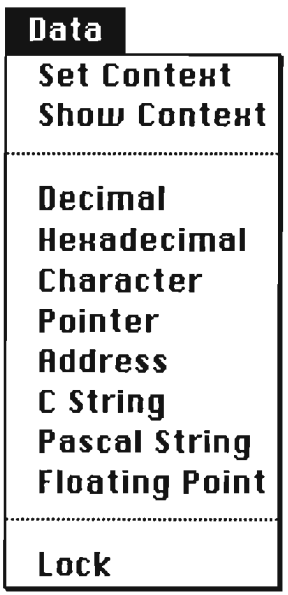
Show Condition

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.

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`
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.

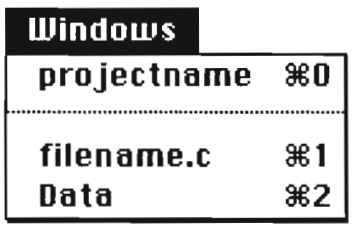
Of course, you can use type casting to use formats that aren't normally available. For example, if you really wanted to see an integer, `i`, as a C string, you would type this expression: `(char * ) i`.

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 16

Introduction

This section describes the C language implemented by THINK'S LightspeedC on the Macintosh. It is meant to be used in conjunction with Appendix A, "C Reference Manual", of the first edition of Kernighan and Ritchie's *The C Programming Language*. The sections are named and numbered exactly as in that work, and only those aspects of THINK C which differ from *K&R*, or are potentially ambiguous in *K&R*, are described here.

2.2 Identifiers (Names)

K&R specifies that no more than the first eight characters of identifiers are significant, although more can be used. In THINK C, there is no limit to the number of characters which are significant in an identifier.

In addition, *K&R* specifies that external identifiers, which are used by various assemblers and loaders, may have additional restrictions. In THINK C, no additional restrictions apply to external identifiers.

2.3 Keywords

The following identifiers are reserved in THINK C:

```
asm         extern      sizeof
auto        float       static
break       for         struct
case        goto        switch
char        if          typedef
continue    int         union
default     long        unsigned
do          pascal      void
double      register    while
else        return
enum        short
```

The `entry` keyword reserved by *K&R* for future use is not reserved in THINK C. In addition, *K&R* mentions that the `fortran` is reserved by some implementations of C; it is not reserved in THINK C.

The keywords `enum`, `pascal`, and `void` were not reserved in *K&R*. See §§ 4 (`void`), 8.1 (`pascal`) and 8.2 (`enum`) for additional details.

2.4.1 Integer constants

K&R allows the digits 8 and 9 to represent the values 10 and 11 in octal constants. THINK C does not allow the digits 8 and 9 in octal constants.

Decimal constants are of type `int`, or `long int` if necessary; hex and octal constants are of type `unsigned int`, or `unsigned long int` if necessary.

A leading '–' is a unary operator, not part of the constant.

2.4.3 Character constants

K&R specifies that certain non-printing characters can be specified using certain escape sequences. The following character escapes are implemented in THINK C:

```
'\a'0x07 (bell)
'\b'0x08 (backspace)
'\f'0x0C (form feed)
'\n'0x0A (line feed)
'\r'0x0D (carriage return)
'\t'0x09 (horizontal tab)
'\v'0x0B (vertical tab)
```

As described in *K&R*, an escape sequence of the form `'\ddd'` where `d` is an octal digit, is also supported.

A single-character character constant has type `int`, and its value is always positive; e.g. `'\377'` is 255, not –1.

Multi-character character constants are allowed. The type of such a constant is `int` if the value is in range, `long int` otherwise. The characters are assigned left-to-right and right-justified. More than 4 characters are not allowed.

2.4.4 Floating constants

Floating constants have type `double`. Data types `float` and `short double` are also supported. See §2.6 for details.

2.5 Strings

In C, each string is terminated with a null byte (`'\0'`) so that programs that scan the string can find its end. This convention is supported in THINK C. However, because Macintosh Toolbox and Operating System calls were designed to be called from Pascal, which uses a different string representation, a second type of string constant has been included.

A string beginning "`\p`" or "`\P`" is a Pascal string. It is not terminated with a null byte; instead the first byte is a length byte indicating the number of characters following. Be careful; the

length byte may appear negative if it exceeds 127, and it will not be meaningful at all if the string is longer than 255 characters.

Pascal strings are required when calling certain Macintosh routines; however, if you like, they can be used in other cases as well.

All string constants are aligned on word boundaries.

2.6 Hardware characteristics

Data types have the following hardware characteristics in THINK C:

type	size
`char`	8 bits
`short int (short)`	16 bits
`int`	16 bits
`long int (long)`	32 bits
`float`	32 bits
`short double`	64 bits
`double`	80 bits (96 bits if 68881 option is on)

Floating-point is IEEE standard, courtesy of Apple's Standard Apple Numeric Environment (SANE) numerics package. The range of `double` values is $\pm 10^{\pm 4932}$. `float` corresponds to SANE `SINGLE`, `short double` corresponds to SANE `DOUBLE`, and `double` corresponds to SANE `EXTENDED`.

4. What's in a name?

Integers of all sizes, including `char`, may be declared `unsigned`. "Plain" `char` is a signed quantity and suffers sign-extension.

Enumerated types are implemented; see §8.2. An enumerated type is not actually a new type, but a synonym for the appropriate size integral type.

A `void` type is available. (This is a recent addition to C, since *K&R*. See Harbison & Steele, section 5.10, for additional details.)

The type "function returning `void`" represents a procedure that does not return a value.

The type "pointer to `void`" (`void *`) is an anonymous pointer type which may be freely converted to any other pointer type without need of a cast.

Finally, an expression may be cast to `void` to indicate that it is being evaluated only for its side effects; its value is discarded.

The `void` type has no other uses; no `void` objects may be declared, and no `void` values may be used. The compiler prevents `void` functions from returning values.

6.1 Characters and Integers

K&R points out that a character may be used wherever an integer may be used, but that in all cases, the value is converted to an integer, and the conversion results in sign extension.

In THINK C, variables of type `char` are signed and do suffer sign-extension; however, single-character character constants are of type `int` and by contrast with *K&R* always have positive values. For example, the value of `'\377'` is 255, not –1.

6.2 Float and double

All floating-point arithmetic is carried out in extended 80-bit precision (90-bit if the 68881 option is on). Whenever a `float` or `short double` appears in an expression, it is extended to `double` which corresponds to SANE type `EXTENDED`. Conversion is performed according to Apple's SANE numerics package. `float` corresponds to the SANE type `SINGLE`. `short double` corresponds to the SANE type `DOUBLE`.

6.3 Floating to Integer Conversions

Floating-point values converted to arithmetic types are truncated to integral values.

6.5 Unsigned

In any conversion, sign-extension is performed according to the old type, not the new type. Thus, when an unsigned integer is converted to a longer signed integer, it is *not* sign-extended; when a signed integer is converted to a longer unsigned integer, it *is* sign-extended. (This feature is exactly as defined in *K&R*.)

6.6 Arithmetic Conversions

This section describes the "usual arithmetic conversions" that occur when various operators are used. These conversions may be applied to a single operand (as in a unary operator), or jointly to a pair of operands (as in a binary operator).

First, all operands of certain types are converted to a larger type as follows:

type	**converted to**
`char`	`int`
`unsigned char`	`unsigned int`
`float`	`double`
`short double`	`double`

Then, if both operands have the same type (or if there is only one operand), that is the type of the result. Otherwise, both operands are converted to a common type, and that is the type of the result. The common type is whichever of the two types appears *first* in the following list:

```
double
unsigned long int
long int
unsigned int
int
```

7.1 Primary expressions

The type of a constant may be `int`, `long int`, `unsigned int`, `unsigned long int`, or `double` depending on its form.

In a function call, actual arguments of integral type are extended to the size of an `int` if necessary, and floating-point values are converted to `double`. Arguments of `struct` or `union` type are allowed, and are passed by value.

If an undeclared identifier is used as the name of the function to be called, it is contextually declared to be a function returning an `int`, exactly as if the declaration `extern int` *identifier*`();` had appeared. **Exception**: if the identifier names a Macintosh Toolbox or OS call, the definition of that call is entered, somewhat as if the declaration `extern pascal` *type identifier*`();` had appeared, where *type* may be `void`, `char`, `int`, or `long`. However, more information is actually available than given by such a declaration.

If an identifier which was declared `pascal` is used as the name of the function to be called, Pascal calling conventions are used instead of C calling conventions. Integral arguments are not extended to the size of an `int`, and no argument may exceed 4 bytes in size. (See Chapter 10 for additional details.)

Macintosh Toolbox and OS calls are handled similarly to `pascal` functions. In addition, each actual argument is extended or truncated depending on the size of argument expected by the call. If an argument of non-integral type must be resized, or if the wrong number of actual arguments is supplied, the compiler signals an error. (Note that the compiler does not know the expected types of the arguments, only their sizes, so be careful.)

The rules given in *K&R* for `struct` and `union` references are strictly enforced by THINK C. The identifier in a `.` or `->` expression must be that of a member of the appropriate `struct` or `union`.

7.2 Unary operators

The `&` operator may be applied to an array, producing a value of type "pointer to array of ...". Without the `&` operator, the array would have been converted to type "pointer to ..." which is quite different, so be careful.

The `&` operator may be applied to a function, but with no effect since the function would have been converted to type "pointer to function returning ..." anyway.

7.3 Multiplicative operators

In a division or remainder operation, an unsigned division is performed if the result is to be an `unsigned` integer of any size. In a signed division, a non-zero remainder has the sign of the dividend.

7.4 Additive operators

The operands must not have type "pointer to `void`". For example:

```
void *generic_pointer;    /*This is legal*/
generic_pointer ++;       /*This is illegal*/
```

The difference of two pointers has type `long int`.

7.5 Shift operators

The usual arithmetic conversions are performed on the left operand alone, as though a unary operator were being applied. An arithmetic right shift is performed if the left operand is signed.

7.6 Relational operators

A pointer may only be compared to a pointer of the same type or a pointer to `void`.

7.7 Equality operators

A pointer may only be compared to a pointer of the same type, a pointer to `void`, or a constant zero.

Whenever a value is tested to see if it is non-zero, as in the conditional operator and in the `if`, `while`, `do`, and `for` statements, the test is equivalent to `(expression) != 0`. All conversions and type restrictions apply accordingly.

7.13 Conditional operator

In addition to the possibilities given in *K&R*, one of the second and third operands may be a pointer and the other a pointer to `void`; the type of the result is the non-anonymous pointer type.

7.14 Assignment operators

Values of `struct` and `union` type may be assigned; the left and right side must be of the same type. This applies only to simple assignment.

A pointer may be assigned the value of a pointer of the same type, a pointer to `void`, or a constant zero. A pointer to `void` may be assigned the value of a pointer of any type, or a constant zero. Anything else is a compiler error. Use casts to bypass these typing rules.

8.1 Storage class specifiers

THINK C implements the additional storage class `pascal`. An identifier declared `pascal` must have type "function returning ...". A `pascal` function is called using Pascal calling conventions; see §7.1. If the `pascal` keyword appears in a function definition, the function will expect to be called using Pascal calling conventions. The `pascal` keyword may appear in conjunction with `extern` and `static`.

A variable of any integral type may be declared `register`; it will be placed in a data register (if possible). A variable of any pointer type may be declared `register`; it will be placed in an address register (if possible). As many as five data registers (`D3-D7`) and three address registers (`A2-A4`) may be active at once; registers allocated to `register` variables become available again at the end of the block in which they are assigned. Excess register declarations are ignored. (When the project type is desk accessory, device driver, or code resource, register `A4` is not available for register variables.)

If the 68881 option is on, five floating point variables may be delcared `register` (`FP3-FP7`).

8.2 Type specifiers

Additional type specifiers are enumeration specifiers and `void`. The `unsigned` modifier may be applied to `char` and `int` (even in the presence of `short` or `long`). Finally, `short double` refers to the SANE double-precision floating-point type (`long float` is accepted as equivalent to `double`).

Enumeration specifiers have the following syntax:

enumeration-specifier:
    `enum` *identifier*
    `enum` $identifier_{opt}$ `{` *enum-list* `}`
enum-list:
    enum-item
    enum-item , *enum-list*
enum-item:
    identifier
    identifier = *constant-expression*

The tag identifier works just as with structs and unions to identify the enumeration type.

Each identifier appearing in the *enum-list* is defined as an enumeration constant whose type is the enumeration type. If not explicitly specified by "= *constant-expression*", the value of the constant is one greater than that of the constant preceding it in the *enum-list*, or 0 if it is the first one. Constant expressions appearing in `enum` declarations must have integral type.

An enumeration type is not really a distinct type, but equivalent to `int`, or to `char` if all of its declared constants have values in the range –128 to 127.

8.4 Meaning of declarators

According to *K&R*, "functions may not return arrays, structures, unions or functions, although they can return pointers to such things." In THINK C, functions may return structures and unions.

Caution: If a function returns a `struct` or `union`, make sure that declaration of the function is present whenever the function is used, *even if* the return value is ignored. Otherwise, the consequences may be dire. (See Chapter 12 for details.)

8.5 Structure and union declarations

Members of all types other than `char`, `unsigned char`, and singly and multiply dimensioned arrays of `char` or `unsigned char` are aligned on even addressing boundaries. Similarly, `structs` and `unions` are padded to an even size.

It is legal to specify an array without size as the last member in a `struct`. The array does not contribute to the size of the structure. For example:

```
struct {
        unsigned int    count;
        char    data[];
} CountData; /*  size of CountData is 2  */
```

Bitfields (or fields, as they are called in *K&R*) may be declared to be of any integral type. The size of the declared type determines the word size for that bitfield; thus a word may be 8, 16, or 32 bits in length. Keep this definition of "word" in mind throughout the following discussion.

A sequence of bitfields with the same word size are packed into a word, but a bitfield is placed in the next word if it would otherwise straddle a word boundary. No bitfield may be wider than a word. 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 bitfield with a different word size from the preceding bitfield causes this to happen automatically (just as a non-bitfield member does).

The high-order bit of a bitfield is *not* treated as a sign bit, even if the declared type of the bitfield is signed. Think of it this way: the "real" type of the bitfield is "`unsigned` *n* bits", which gets converted to the declared type whenever the bitfield appears in an expression.

Names of members need only be distinct from each other within a single `struct` or `union` declaration. Each such declaration introduces a unique name space for its members; see §11.1.

8.6 Initialization

Unions may be initialized; the initialization applies to the first member of the union. Structures containing bitfields may be initialized.

9.7 Switch statement

The `switch` expression and `case` constants may be of any integral type, after the usual arithmetic conversions are applied.

9.10 Return statement

A function declared "function returning `void`" may not return a value.

10.1 External function definitions

The `pascal` specifier is allowed in addition to `extern` or `static`; see §8.1.

Formal parameters declared `float` or `short double` have their declaration adjusted to read `double`. Formal parameters may have `struct` or `union` type.

11.1 Lexical scope

THINK C follows *K&R* in giving file scope to identifiers declared `extern`, whether explicitly or implicitly. (Many C implementations treat `extern` variables as local to a procedure.)

There are several distinct name spaces. Names in the same space must not conflict with each other, but may be the same as names in other spaces. For example, the same identifier may be used without conflict for a statement label and an enumeration constant. However, macro substitution is performed by the preprocessor without regard for name spaces.

Statement labels form a name space.

`struct`, `union`, and `enum` tags form a name space.

Each `struct` or `union` defines a unique name space for its members.

Variables, functions, `typedef` names, and enumeration constants form a name space.

11.2 Scope of externals

A function which is declared `static` when its definition is given is *not* exported to other files, even if it was previously declared `extern` (explicitly or implicitly). This allows forward references to private functions.

12. Compiler control lines

Preprocessor lines may begin with any number of spaces or tabs (but *not* comments). Lines containing only a # (preceded by any number of spaces or tabs) are ignored.

12.3 Conditional compilation

An identifier may optionally appear following `#else` or `#endif`. This identifier must match the corresponding `#ifdef` or `#ifndef`.

The command `#elif` is allowed; it is like `#else` followed by `#if`, but no additional matching `#endif` is required.

Identifiers appearing in an `#if` (or `#elif`) expression evaluate to 0 if they are not macro names.

12.4 Line control

The `#line` command is accepted but ignored in the THINK C environment.

13. Implicit declarations

An undeclared identifier appearing in a function-call context is implicitly declared to be of type "function returning `int`", *unless* it is recognized as the name of a Macintosh Toolbox or OS call; see §7.1.

14.1 Structures and unions

Structures and unions can be assigned, passed as parameters, and returned from functions.

The identifier in a `.` or `->` expression must be that of a member of the appropriate `struct` or `union`.

14.4 Explicit pointer conversions

A `long int` (or `unsigned long int`) is required to hold a pointer without loss of information. Pointers are byte addresses. `chars` (and `unsigned chars`) have no alignment requirements; everything else must have an even address.

15. Constant expressions

The `!` operator may be used in constant expressions; its omission is clearly just an oversight on the part of *K&R*.

16. Portability considerations

In addition to the portability considerations specified in *K&R*, the use of the `pascal` type, of Pascal string conventions, or of the type `short double` will result in code that is not portable to most other C environments.

17. Anachronisms

Obsolete constructions are not supported.

18. Syntax Summary

The constructions described in the next two sections have been added to THINK C.

18.2 Declarations

`pascal` is an additional *sc-specifier*.

`void` and *enumeration-specifier* are additional *type-specifiers*.

enumeration-specifier:
`enum` *identifier*
`enum` *identifier*$_{opt}$ `{` *enum-list* `}`

enum-list:
enum-item
enum-item , *enum-list*

enum-item:
identifier
identifier = *constant-expression*

18.5 Preprocessor

`#else` *identifier*$_{opt}$

`#endif` *identifier*$_{opt}$

`#elif` *constant-expression*

THINK's LightspeedC

PART FIVE

Appendices

A The Profiler
B Troubleshooting
C Error Messages
D RMaker Reference

The Profiler
A

Introduction

This chapter shows you how to use the THINK C code profiler. The profiler is a tool that records how much time your functions take to execute. The profiler uses the `stdio` library. The profiler source code is part of your THINK C package, so you can modify to suit your task.

Topics covered in this chapter:

- Using the profiler
- Modifying the profiler

Using the Profiler

To use the profiler, check the Profile checkbox in the Code Generation section of the **Options...** dialog. When this option is on, THINK C inserts calls to profile routines at the beginning and end of your functions.

> **Note:** The Profiler always generates a stack frame, even for functions with no parameters and no local storage.

Your project must contain the `profile` library as well as the `stdio` library. The profiler logs the amount of time spent in each function, and prints the results to `stdout` on exit. The display reports on:

- minimum time spent in routine
- maximum time spent in routine
- 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 profile option on.)
- number of times the routine was called

The profiler uses the units of the VIA 1 timer. Each unit is 1.2766 µsec. You can change the units to ticks (60ths of a second) by editing the profiler code. See "Modifying the Profiler" below.

Turning the profiler on and off

The profiler prints its report to `stdout` when your program exits. (It uses the `onexit()` function.) To see the statistics any time before your program ends, just call the function `DumpProfile()`.

You can control whether the profiler is accumulating time statistics by setting the value of the `_profile` variable. If this variable is non-zero, the profiler stops recording time information.

The profiler can print an indented call-tree of your functions calls as your program runs. Set the `_trace` variable to non-zero to have the profiler print every time a function is entered.

Modifying the Profiler

You can modify the profiler code to change its behavior. The profiler sources are in the `Profiler` folder of the `Library Sources` folder in disk `THINK C 2`. The profiler consists of two files: `profilehooks.c` and `profile.c`.The `profilehooks.c` file contains the two assembly language routines that are called at the beginning and end of each function. These routines call the higher level profiler routines in profile.c which do the statistics collection.

The symbol `VIATIMER` in `profile.c` controls whether to use the VIA timer or the tick counter. This symbol is initially defined.

If the symbol `SINGLE` is defined the accumulated times of all routines called by a given routine are not incorporated into the time of the routine. This way you can get an idea of how much time is spent in individual routines. If you'd like to see accumulated times, undefine `SINGLE`.

> **Note:** When you rebuild the `profile` library, make sure that the Profile option is not selected. Otherwise you will be in an infinite loop at run time.

Summary

`DumpProfile()`	(`profiler.c`) Write the statistics collected so far to `stdout`. You can call this function any time to print the current statistics. It is called automatically on exit.
`_profile_()`	(`profilehooks.c`) Assembly language routine called at the beginning of each function when the Profile 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()`.
`__profile()`	(`profile.c`) C function that collects timing statistics.

`_profile_exit()`	(`profilehooks.c`) Assembly language routine called at the end of a function. Code to execute this routine is not generated, instead, `_profile()` munges the stack so the original routine "returns" to this one.
`__profile_exit()`	(`profile.c`) C function that computes timing statistics.
`_profile`	(`profile.c`) Variable, if non-zero, enables profiling.
`_trace`	(`profile.c`) Variable, if non-zero, enables call tree generation.

Troubleshooting B

Introduction

This chapter contains a list of frequently encountered problems, and a set of tips and suggestions about avoiding errors and improving code.

Topics covered in this chapter

- Getting help
- Some common problems

Getting Help

If you run into a problem, look in this chapter to see if it's covered here. If it's not, look in the section of the manual that deals with the part of the THINK C environment that's giving you trouble. If you can't find the answer to your questions in the manual, you can call THINK Customer Support at (617) 275-1710 from 9am to 5pm Eastern Time.

Some Common Problems

You might run into some common problems when you use THINK C. Most of these problems are easy to correct.

Undefined symbols

Undefined symbol errors show up in the Link Errors window when THINK C can't link your project during a **Run**, **Build...**, **Bring Up To Date**, or **Make...** command. Typically, you'll see a message like `undefined: printf`.

- You have not added a necessary library. If the missing symbol is a Macintosh Toolbox function, you probably forgot to add the `MacTraps` library to your project. The `MacTraps` library must be in nearly all projects, since it contains the QuickDraw globals and glue code that accesses the register-based Macintosh Toolbox functions, as well as the `[Not in ROM]` routines.

- If you're using any C library routines, you may have forgotten to add the library that contains that routine. Check the documentation for the routine in the *Standard C Libraries Reference*. The name of the library you need is given with the description of each routine. Use the **Add...** command to add the library to your project.

- You may have made a spelling or capitalization error. C is a *case-sensitive* language, which means that `printf` is not the same as `Printf`. Check to see that the undefined function name is spelled and capitalized correctly.

- You have redefined a Macintosh or C library function as "`extern`". For example:

```
extern void SetRect();
```

  Such re-definitions are unnecessary, since Toolbox function definitions are already built into THINK C. If one appears, the linker will try to resolve the reference with a user-defined function. This is fine if what you want to do is *replace* a standard function, but will cause a linker error if no replacement function is provided. (If you want to supply return type or prototype information for Toolbox calls, see Chapter 10.)

Code segment too large

If the Link Error window reports `Code segment too big`, one or more of your segments contains more than 32K of object code.

- Move files out of the too-big segment until it contains less than 32K of code. See the "Segmentation" section of Chapter 7 to learn how to move files between segments. Use the **Get Info...** command in the **Source** menu to see how large your segments are.

Data segment too big
Stack frame too big

The static and global data area of THINK C applications is limited to 32K *total* per project. The limits for desk accessories, device drivers, and code resources are different. See Chapter 7 for details.

- If you want to be able to access global or local memory larger than the 32K limit, allocate the memory dynamically as a pointer or handle. Because the C language blurs the distinction between arrays and pointers, you can then use the pointer with an array notation to access elements in dynamic memory. (Refer to a good C language manual, such as Kernighan and Ritchie's *The C Programming Language* for more details.) For example:

```
#define SIZE     100000
#define SIZE2D       100

int BigBadArray[SIZE];                  /* a too big array */
int *LooksLikeAnArray;
int BigBad2DimArray[SIZE2D][SIZE];  /*too big 2D array*/
int *LooksLikeA2DArray[SIZE2D];     /*array of pointers*/

proc()
{
 register long i;

 /* allocate "array"*/
 LooksLikeAnArray = (int *)NewPtr(sizeof(int)*SIZE);

 /* You can index it like an array */
 LooksLikeAnArray[60000] = 5;

 /*  allocate 2D "array"*/
 for (i = 0; i < SIZE2D; i++)
 LooksLikeA2DArray[i] = (int *)NewPtr(sizeof(int)*SIZE);

 /* You can index this like an array, too */
 LooksLikeA2DArray[10][20] = 7;
}
```

Can't find an #include file

When THINK C reports that it can't find an #include file, the file is probably not in the right tree. Your project's #include files should be in your project tree, and standard #include files should be in the THINK C Tree. To learn about the two trees, see Chapter 9.

Odd address error (System error ID=02)

An odd address error usually means that your program is accessing memory on the heap incorrectly. These errors can usually be traced to a few common causes.

- You may have referenced a null handle or pointer. If your program uses a resource file that is not found, a `GetResource()` (or `GetNewMenu()`, `GetNewWindow()`, etc.)

call will return a null handle. Make sure that if you use a resource file, it's in the same folder as your project file, and that it is named *projectname*`.rsrc`. (See Chapter 7.)

- You may not have enough memory for a `NewHandle` or `NewPtr` call, resulting in a null return value. Be sure you check the results of every memory allocation.
- You may have not initialized a pointer variable.
- You may have a Toolbox routine with bad or inappropriate data. For example, you are likely to crash if you call `DisposHandle()` with a handle to a resource. Use `ReleaseResource()` instead.

Printf and scanf not working

It might seem that the standard C functions `printf()` and `scanf()` don't work correctly.

- A common misconception about `printf()` is that `printf()` "knows" about its arguments. If you pass a long expression to `printf()`, you must specify in the format string that you want a long expression to be printed. Example:

```
int anInt;
long aLong;
printf("long is %ld, int is %d\n", aLong, anInt);
```

In `scanf`, the same goes for floats and doubles:

```
float aFloat;
double aDouble;
printf("Input a double and a float:");
scanf("%lf %f", &aDouble, &aFloat);
```

Link error: printf undefined

Many novice users (and not-so-novice users!) get confused about the difference between "header" files and "library" files. *Header files* (which conventionally have names which end in `.h`) contain source statements: definitions and declarations which allow the compiler to make sense of source code calls to a library function. *Libraries* contain the actual object code for the functions themselves, which the linker references in building the project. Including a header file is for the compiler only; it tells the linker nothing. `#include`-ing a library in a source file is an error.

Error Messages C

8-bit reference to *'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. Example:

```
asm {
    bra.s   @foo
    …               /* over 127 bytes */
foo:
```

'&' (address-of) operator illegal here

The address-of operator was used on an object (such as a register variable or a bitfield) that doesn't have an address. Example:

```
function()
{
  register int i;
  int *p = &i;     /* Illegal - i is a register */
}
```

'symbol' has not been declared

The identifier *symbol* has been used before it has been declared.

'symbol' is not a formal parameter

The *symbol* was declared as if it was a formal parameter in a function definition, but it is not in the parameter list. Example:

```
function()
int left_out_of_parameter_list;  /* Illegal */
{
}
```

'filename' is not a text file

A file name in an #include statement refers to an existing file that is not a text file. You'll also see the message when you Option double-click on a symbol to find its definition, and the symbol is defined in a library. Example:

```
#include <stdio>     /* stdio.h is the #include file */
                     /* stdio is the library */
```

"array of functions" is not allowed

An array of functions is not allowed. However an array of pointers to functions is allowed. Example:

```
int array_of_functions[]();                 /* Illegal */
int (*array_of_ptrs_to_functions[SIZE])(); /* Legal */
```

"array of void" is not allowed

Since `void` objects are not allowed, arrays of `void` objects are not allowed either. However, it is legal to have an array of pointers to `void`. This is a C idiom for an array of generic pointers. Example:

```
void abyss[SIZE];              /* Illegal */
void *generic_ptr_array[SIZE]; /* Legal */
```

bad operand

(Assembly) An assembly language operand has an instruction that is not legal in assembly language. Example:

```
asm {
    move    d0, x->field    /* use OFFSET() macro, instead */
}
```

break outside of loop or switch

The `break` statement must be inside a loop (`while`, `do-while`, `for`) or a `switch`. Example:

```
function()
{
    int i;
    break;                  /* Illegal */
    while(1)
        break;              /* Legal */
    do { break; }           /* Legal */
        while(1);
    switch(i){
    case 1:
        break;              /* Legal */
}
}
```

call of '*symbol*' does not match prototype

The prototype given for symbol and the call for function symbol don't match. Example:

```
extern function(int, char); /* the prototype */

another_function()
{
    function(23, "skidoo");     /* types don't match */
}
```

call of non-function

An expression in the function call position does not evaluate to a function. Example:

```
function(f_ptr)
int (*f_ptr)();    /* f_ptr is a pointer to a function */
{
  int i, j, k, result;
  /* probably left out an operator between i and (j+k) */
  result = i(j+k);          /* Illegal */
  result = f_ptr(i, j);     /* Illegal */
  result = (*f_ptr)(i,j); /* Legal */
}
```

cannot initialize auto struct/union/array

Only global or static structures/unions/arrays may be declared with initializers. Example:

```
int array[6] = { 0,1,2,3,4,5 };      /* Legal */
function()
{
  int array[6] = { 0,1,2,3,4,5 }; /* Illegal - auto array */
  static struct{
    double pi;
  } pi = { 3.14159 };        /* Legal - because of static */
}
```

can't do that with multi-segment project

What you're trying to do is not legal when the project contains more than one segment. You'll see this message when you use the **Build Library...** or **Build Code Resource...** command on a multi-segment project, when you use a multi-segment project as a library, or when you use the **Build Device Driver...** or **Build Desk Accessory...** command on a project when the Multi-Segment option is not on.

can't load STRS in this project

You can't load a library or project built with the Separate STRS option on into a project whose Separate STRS option is off. You can rebuild the library with the Separate STRS off, or you can turn the option on in the project that's using the library.

can't open #include'd file

The file to be `#include`'d cannot be opened. Its name may be misspelled, it may not exist, or it may be in the wrong tree (see Chapter 9).

can't precompile code or data

You tried to Precompile a file that contained a function or variable definition. Only declarations are allowed in precompiled headers.

case not in switch

The `case` keyword was found outside of a `switch` block.

code overflow

This message means that you have more than 32K of code or data in one file. Break your file up into smaller files.

constant required

A constant was required, but none was found. Example:

```
#define aConstant  12
function()
{
  int h;
  int x[2];  /* Legal - array dimension is a constant */
  int y[h];  /* Illegal - h is not a constant */
  enum{
     color = h, /* Illegal - enum values must be constant */
     shape = -2.4,   /* Illegal - enum values must be int */
     size = aConstant /* Legal */
  } attributes;
  struct{
      unsigned illegal_bitfield : h;  /* Illegal */
      unsigned legal_bitfield : size; /* Legal */
  }bits;
  int z[size];  /* Legal - array dimension is a constant */
  int w[3.14159];  /* Illegal - constant must be an int */
}
```

code segment too big

See **link failed**.

constant too large

The absolute value of decimal integer constants must be less than or equal to `2147483647` ($2^{31}-1$). Hexadecimal integer constants must range from `0x0` to `0xffffffff` inclusive. If you want to express an unsigned number larger than `2147483647`, you must express it as a hexadecimal number. Example:

```
unsigned long a = 2147483648;  /* Illegal */
unsigned long b = 0x80000000;  /* Legal - equals 2147483648 */
unsigned long c = 0x100000000; /* Illegal - larger than 32 bits */
```

continue outside of loop

The `continue` statement must be inside a loop (`while`, `do-while`, `for`). Example:

```
function()
{
  int i;
  continue;                /* Illegal */
  while(1)
      continue;            /* Legal */
  do { continue; }         /* Legal */
      while(1);
  switch(i){
  case 1:
      continue;            /* Illegal */
  }
}
```

data segment too big

See **link failed**.

debug table overflow

You've exceeded the size limits for debugging information tables. You can usually fix this by using a precompiled header. If you're not using MacHeaders, consider using it. If you don't want to use MacHeaders, make a precompiled header of the header files you are using. See Chapter 10 to learn how to make precompiled headers.

declarator too complex

The declaration statement is too complex. Example:

```
int **************************x[1][1][1][1][1][1][1][1][1][1][1][1]
```

default not in switch

The `default` keyword was found outside of a `switch` block.

duplicate case

A `case` constant expression evaluates to the same value as a previous `case` constant expression within the same `switch` block.

duplicate default

There may only be one `default` specified in a `switch` block.

expression too complex

An expression was too complex. Try breaking up the expression into subexpressions. Example:

```
typedef int ***************************** indirect_type;

function(meta_ptr)
indirect_type *****************************meta_ptr;
{
   int i;
   indirect_type intermediary;
   /* The following expression is too complex */
   i = ****************************
       **************************** meta_ptr;
   /* The equivalent broken up into two subexpressions */
   intermediary = ****************************meta_ptr;
   i = **************************** intermediary;
}
```

floating-point expression too complex

The compiler has used up all of its 68881 floating point registers. Break up the floating point expression or use fewer floating point register variables.

formal parameter 'symbol' appears more than once

The formal parameter *symbol* was used in the function parameter list more than once. Example:

```
function(again, again, again)  /* Illegal */
int again;
{
}
```

function definition does not match prototype

The data types in a prototype defined for a function don't match the types in the definition of the function. Example:

```
extern function(char);  /* the prototype */
...
function(c)
int c;              /* type doesn't match prototype */
{
}
```

"function returning array" is not allowed

A function cannot return an array. However a function can return a pointer that is the address of an array. Example:

```
char function_returning_array()[SIZE];          /* Illegal */
char *function_returning_ptr_to_array();        /* Legal */
```

"function returning function" is not allowed

A function cannot return a function. However a function can return a pointer to a function. Example:

```
int function_returning_function()();     /* Illegal */
typedef int (* function_ptr)();
function_ptr  f_returning_ptr_to_f();    /* Legal */
```

Identifier does not match #ifdef

The `#endif` and `#else` macro preprocessor statements take an optional argument which is the symbol that the matching `#ifdef` or `#ifndef` (but not `#if` or `#elif`) used. If the optional argument is present, it must match the corresponding `#ifdef` or `#ifndef`. Example:

```
#ifdef macro_name
#else  Other_name /* Illegal - identifier does not match */
#endif Other_name /* Illegal - identifier does not match */
#ifdef macro_name
#endif macro_name /* Legal */
```

Illegal #else

Every `#else` or `#elif` must have a matching `#if`, `#ifdef`, or `#ifndef`. Example:

```
#ifdef name
#else
#else             /* Illegal - has no matching #ifdef */
#endif name
#elif condition   /* Illegal - has no matching #if */
```

Illegal #endif

An `#endif` was encountered that didn't have a matching `#if`, `#ifdef`, or `#ifndef`.

Illegal array bounds

Examples:

```
int a[-7];       /* Illegal - bounds can't be negative */
int b[0];        /* Illegal - bounds can't be zero */
char c[32768];   /* Illegal - total array size > 32767 bytes */
int d[16384];    /* Illegal - total array size > 32767 bytes */
long e[8192];    /* Illegal - total array size > 32767 bytes */
```

Illegal cast

It is illegal to cast `structs`/`unions` to other types. It is legal to cast numerical values/pointers to other numerical values/pointers. Example:

```
typedef struct {int v, h;} Point;
function()
{
  Point p;
  long l;
  p = (Point) l;      /* Illegal */
  p = *(Point *) &l   /* Legal */
}
```

Illegal floating-point operation

You cannot use some of the C operators on floating point values. Example:

```
function()
{
  double d1;
  double d2;
  double d3;
  d3 = d1 % d2;    /* Illegal - can't do floating modulo */
  d3 = d1 << 3;    /* Illegal - can't do floating shifts */
  d3 = 3 << d1;    /* Illegal - can't do floating shifts */
}
```

Illegal function prototype

Only types and optional identifiers are allowed in prototypes. Example:

```
extern  char *badProto(int, char, cem[8]);  /* Illegal */
extern  int naughtyProto(int p, char fn()); /* Illegal */
```

Illegal operation on array

You cannot use many of the C operators on arrays. Example:

```
function()
{
  char a1[4];
  char a2[4];
  char a3[4];
  int i;
  if ( a1 == a2 )   /* Legal - compares arrays' addresses */
   a1++;            /* Illegal - can't increment arrays */
  a1 = 0;           /* Illegal - can't assign to an array */
  a1 = a2;          /* Illegal - can't assign arrays */
  i = a1 - a2;      /* Legal - subtracts arrays' addresses */
}
```

Illegal operation on function

You cannot use many of the C operators on functions. Example:

```
function()
{
  int f1();
  int f2();
  int f3();
  int i;

  if (f1 == f2)   /* Legal - compares functions' addresses */
    f1++;         /* Illegal - can't increment functions */
  f1 = 0;         /* Illegal - can't assign to a function */
  f1 = f2;        /* Illegal - can't assign functions */
}
```

Illegal operation on struct/union

You cannot use many of the C operators on `structs/unions`. Example:

```
int function()

  struct{
    int x;
  }s1, s2, s3;
  int i;

  if (s1 == s2) /* Illegal - can't compare structs */
    s1++;       /* Illegal - can't increment structs */
  s1 = 0;       /* Illegal - can't assign int to a struct */
  s1 = s2;      /* Legal - struct assignment is allowed */
  s3 = s1 + s2; /* Illegal - can't add structs */
  i = s1 - s2;  /* Illegal - can't add structs */
  i = (long)s1; /* Illegal - can't cast a struct */
  return(s1);   /* Illegal - return does an implicit cast */
}
```

Illegal operator

(Debugger only) You've entered an expression with an operator that has a potential side effect: a function call, ++, or --.

Illegal pointer/integer combination

Integers may not be assigned to pointers with the exception of the constant zero. In addition, pointers may not be compared with integers, again with the exception of the constant zero. In both cases a cast can be used to force the assignment or comparison where necessary. Example:

```
function()
{
  int *int_ptr;
  int i;
  int_ptr = 0x220;          /* Illegal */
  int_ptr = 0;              /* Legal */
  int_ptr = (int *) 0x220; /* Legal - address of MemErr */
  i = * 0x220;              /* Illegal */
  i = * (int *) 0x220;      /* Legal - contents of MemErr */
  return(int_ptr);          /* Illegal - implicit cast */
}
```

illegal pointer arithmetic

The arithmetic being performed on the pointer is illegal. Example:

```
function()
{
  char *p1, *p2;
  long result;
  result = p1 + p2;  /* Illegal - can't add pointers */
  result = p1 / p2;  /* Illegal - can't divide pointers */
}
```

illegal return type for pascal function

Pascal functions are only allowed to return void, integers, and pointers. Example:

```
typedef struct {int v, h;} Point;
pascal Point /* Illegal - can't return a Point */
pascal_function(x,y)
{
  Point p;
  p.v = x;
  p.h = y;
  return p;
}
pascal double illegal_pascal_fnc(); /* Illegal return type */
pascal void pascal_proc();          /* Legal */
```

illegal size for bitfield

Bitfield sizes must be less than or equal to the number of bits in the word type. (See Chapter 16, § 8.5.) Labelled bitfield sizes must be greater than zero. Unlabelled bitfield sizes equal to 0 force a word alignment. Example:

```
struct bitfields{
  char c : 9;      /* Illegal - chars have 8 or fewer bits */
  int i : 17;      /* Illegal - ints have 16 or fewer bits */
  int zero : 0;    /* Illegal - bitfield size must be > 0 */
  int good : 11;   /* Legal */
  int : 0;         /* Legal - this forces word alignment */
  long l : 33;     /* Illegal - longs have <= 32 bits */
  int z : -4;      /* Illegal - bitfield size must be > 0 */
};
```

Illegal size for this operation

(Assembly) The size specifier for an assembly language instruction is the wrong size for what you're trying to do. Example:

```
asm {
    move.b  a0,a1    /* can't move bytes to address registers */
}
```

Illegal token

Certain characters that are part of the ASCII character set and all of the characters in the extended Macintosh character set are illegal tokens in THINK C. The character # is also an illegal token when it is other than the first character in a line preceded by any number of whitespace characters. However, any character can occur within comments, string literals or character literals. Example:

```
int a$variable;          /* Illegal */
char c = '$';            /* Legal to have $ here */
#define N    "###"       /* Legal to have # here */
char d = #define M 4;    /* Illegal */
```

Illegal type for bitfield

The only allowed types for a bitfield are `char`, `short`, `int`, and `long`, and these types prefixed by the modifier `unsigned`. Any other type for a bitfield is illegal. Example:

```
struct bitfields{
  char c : 5;             /* Legal */
  unsigned int i : 5;     /* Legal */
  long l : 17;            /* Legal */
  double d : 10;          /* Illegal */
  void *p : 8;            /* Illegal */
};
```

Illegal use of inline Macintosh function

Even though the inline Toolbox Trap calls look like function calls, they are not. Hence, it is illegal to take the address of an inline Toolbox Trap call. Another possible source of error is forgetting to put in the parameter parentheses in a built-in call that takes no arguments. Example:

```
function()
{
  void *generic_pointer = FrameRect; /* Illegal */
  /* what a forgetful Pascal programmer might write */
  while (Button)       /* Illegal - () missing */
    ;
  while(Button()) /* Legal */
    ;
}
```

Illegal use of type name 'symbol'

A `typedef` name has appeared where it shouldn't. If a `typedef` name is used instead of a variable name, then an error occurs. Example:

```
typedef char Byte;
function()
{
  return Byte;      /* Illegal */
}
```

Illegal use of void

You cannot use most of the C operators on `void` values. Example:

```
void f()
{
  int i, j;
  i = (void)j + 5; /* Illegal - can't add a void */
  i = f() - 3;     /* Illegal - f() returns void */
}
```

Immediate operand out of range

(Assembly) Some assembly language instructions must be a certain size. Example:

```
asm {
    addq    #10, d0      /* immediate limit is 1-8 */
}
```

Incomplete macro call

A preprocessor command was found while reading macro parameters during macro expansion. Preprocessor commands are not allowed in this situation. Example:

```
#define macro(arg1, arg2) (arg1 > arg2)
macro(x,
#ifdef AndIfYouCallNow    /* Illegal */
1
#else
2
#endif
)
```

Initialized object too complex

Initialization of a deeply nested `struct` will cause this error. Example:

```
struct s1{
struct s2{
...
struct s21{
    int i;
}...}}s = {4};        /* Too deeply nested */
```

Initialization to an address is illegal in a non-application

If the THINK C project is a code resource, desk accessory, or device driver, it is illegal to initialize an address in global or static memory. Example:

```
static int i;
static int *p = &i;  /* Illegal in non-application */
```

Invalid declaration

Example:

```
/* Illegal - only a defining instance can be initialized */
extern int x = 1;
extern int y;
int y = 1;          /* Legal */
```

Invalid function definition

Example:

```
/* Illegal - function definition can't be declared extern */
extern f1()
{
}
f2() = 2;  /* Illegal */
```

Invalid redeclaration of 'symbol'

An identifier was declared twice and this redeclaration is incompatible with the first. Example:

```
extern int x;
long x;     /* Illegal - x is redeclared differently */
extern int y;
int y;      /* Legal - y is redeclared as the same type */
int z;
int z;      /* Illegal - can only have one defining instance */
typedef int IsARose;
IsARose IsARose;     /* Illegal */
```

Invalid register list

(Assembly) The register list contains duplicates, or the list is in the wrong order. Example:

```
asm {
    movem   a0/a0,-(sp)
    movem   d7-d1,-(sp)
}
```

Invalid storage class

Incompatible attributes have been explicitly or implicitly applied to a data item. Example:

```
register int aGlobal;  /* Illegal - register global */
extern auto int y;     /* Illegal - contradictory storage class */
main()
{
}
```

Invalid type

The specified combination of type keywords create an ambiguous or impossible data item. Example:

```
unsigned double d;          /* Illegal */
long struct {int i;} s;      /* Illegal */
```

jump table too big

See **link failed**.

label *'symbol'* already defined

The label *symbol:* has been defined twice within the function.

label *'symbol'* not defined

There was a `goto` to the label *symbol:* but none has been defined within the function.

link failed

The linker was unable to link the program. Usually the link 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 symbols, 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" in Chapter 7.
- data segment too big — you've declared more than 32K of global and static data in your project. Use memory allocation to create large data structures.
- jump table too big — you've exceeded the 32K jump table limit
- multiply defined: 'symbol' — 'symbol' was defined more than once
- resource too big — for drivers, DRVR + DATA resource is > 32K; for multisegment drivers, DATA + JUMP > 32K; for code resources, CODE + DATA + JUMP > 32K
- undefined: 'symbol' — there was a reference to 'symbol' that was never defined

lvalue required

An lvalue is an expression that refers to an object in memory that can be stored to, as well as examined. See Harbison & Steele or *K&R* for more details. Example:

```
int int_f();
int *pint_f();
function()
{
  int i;
  int *p = &i;
  /* operand of ++ must be an lvalue */
  7++;              /* Illegal */
  int_f()++;        /* Illegal */
  pint_f()++;       /* Illegal */
  /* left operand of an assignment must be an lvalue */
  int_f() = i;      /* Illegal */
  pint_f() = i;     /* Illegal */
  *pint_f() = i;    /* Legal - * produces an lvalue */
  (*p)++;           /* Legal */
  (*pint_f())++;    /* Legal */
}
```

macro name already #define'd

It is illegal to `#define` a macro name that has already been `#define`'d. However, it is always legal to `#undef` a macro name whether or not it has been `#define`'d before. If you use a `#undef` before a `#define`, you will be sure that a subsequent `#define` will always work. Example:

```
#define macro_name 1
#define macro_name 2    /* Illegal - already #define'd */
#undef macro_name
#define macro_name 3    /* Legal to #define AFTER an #undef */
```

macro parameter *'symbol'* appears more than once

The formal parameters in a macro definition must appear only once. Example:

```
#define macro(again,again)   (again+7)   /* Illegal */
```

memory critical - proceed at your own risk

Now would be a really good time to quit. Save your files!

missing #endif

Every `#if`, `#ifdef`, and `#ifndef` must have a matching `#endif`.

missing '('

There is a missing parenthesis. `if`, `while`, `do-while`, `switch`, and `for` statements require the expressions following them to be contained inside parentheses. Example:

```
function()
{
  int flag, value, i;
  if (flag) return 1;          /* Legal */
  if flag return 2;            /* Illegal */
  while flag {...}             /* Illegal */
  do {...} while flag;         /* Illegal */
  switch value {...}           /* Illegal */
  for i = 0; i < 10; i++ {...} /* Illegal */
}
```

missing ')'

There is a missing right parenthesis. Example:

```
function(i)
int i;
{
  if (i > 4    /* Illegal - missing close parenthesis */
    /* then clause */
}
```

missing ':'

There is a missing colon in a conditional (`?:`) expression or in a `case` or `default` label. Example:

```
function(flag)
int flag;
{
  int i;
  i = flag ? 3 4; /* Illegal - Missing colon */
  switch(i){
  case 3          /* Illegal - Missing colon */
    return i+5;
  default         /* Illegal - Missing colon */
    return i+6;
  }
}
```

missing ';'

A semicolon was expected, but one was not found. Sometimes it is not possible for THINK C to determine that a semicolon was missing which will result in the error message **syntax error**.

missing ']'

A closing bracket was expected to match an open bracket, but one was not found.

multiply defined: *'symbol'*

See **link failed**.

no files in project

You need at least one file in the project window for what you're trying to do.

no members defined

A `struct/union` was defined without members. Example:

```
struct memberless_struct{ };     /* Illegal */
union memberless_union { };      /* Illegal */
```

out of memory

THINK C ran out of memory. Close open windows to reclaim more memory. You might want to check the More Memory option in the Preferences section of the **Options...** command in the **Edit** menu.

If you get this message when you try to run the debugger, try making application's partition size smaller. See "Memory Condsiderations" in Chapter 11.

parameter list is inappropriate here

A declaration involving a function must not have a parameter list, except when a function is being defined. or in a function prototype Example:

```
int function(illegal_parameter);            /* Illegal */
int (*ptr_to_function)(illegal_parameter); /* Illegal */
```

pascal argument wrong size

The call to a built-in Toolbox or OS function has a non-integral argument of the wrong size. Example:

```
function()
{
  Rect r;
  Point p;
  ...
  FrameRect(r);      /* Illegal - Should pass ptr to Rect */
  FrameRect(&r);     /* Legal - Correct call to FrameRect */
  FrameRect(4);      /* Legal but wrong - 4 gets cast to ptr */
  SetPt(&p,10L,3);   /* Legal - 10L gets cast to an int */
  SetPt(p,4,5);      /* Legal but wrong - sizeof(p) == sizeof(&p)*/
  SetPt(&p, &p, 5); /* Illegal - ptr won't be cast to int */
  SetPt(&p, p, 5);   /* Illegal - struct won't be cast to int */
}
```

pointer required

A pointer was implied by an operator, but there is no pointer. Example:

```
function()
{
  int x, result;
  result = *x;            /* Illegal - x is not a pointer */
  result = x->a_member;   /* Illegal - x is not a pointer */
}
```

pointer types do not match

Incompatible pointers were used in an assignment, comparison, or subtraction. In THINK C, pointers are more strictly typed than some other C compilers. In the case of assignment and comparison, the two pointer types must match or be of type `void *`. In the case of subtracting two pointers, the types must match and not be `void *`. Of course, pointers may be cast to force compatiblity. Example:

```
function()
{
  char *char_ptr;
  int *int_ptr;
  void *void_ptr;
  int result;
  long difference;
  char_ptr = int_ptr;                        /* Illegal */
  char_ptr = (char *)int_ptr;                /* Legal */
  void_ptr = int_ptr;                        /* Legal */
  int_ptr = void_ptr;                        /* Legal */
  result = (char_ptr == int_ptr);            /* Illegal */
  result = ((int *)char_ptr == int_ptr);     /* Legal */
  result = (char_ptr == (void *)int_ptr);    /* Legal */
  result = (void_ptr == int_ptr);            /* Legal */
  difference = char_ptr - int_ptr;           /* Illegal */
  difference = char_ptr - (char *)int_ptr;   /* Illegal */
}
```

prototype required for '*symbol*'

When the Require Prototypes option in the Compiler Flags section of the Options... dialog is checked, you must provide a function prototype for each of your functions.

recursive #include or preprocessor overflow

The most likely cause of this error is that an `#include` file has `#included` itself directly or indirectly. Another possibility is deeply nested `#ifdefs` or macro invocations. Example:

```
#define name_1     0
#define name_2     name_1
...
#define name_54    name_53
int i = name_54;
/* Deeply nested macro overflows preprocessor */
#ifdef name_1
#ifdef name_2
...
#ifdef name_54
/* Deeply nested #ifdef overflows preprocessor */
```

redefinition of existing struct/union/enum

A `struct/union/enum` with the same tag was declared twice. Only one defining instance is allowed. `struct`, `union`, and `enum` tags share the same name space. Example:

```
struct Rumpelstiltskin{
  int member;
};
/* The following are illegal only because the tag   */
/* Rumpelstiltskin has been used previously         */
/* Illegal */
struct Rumpelstiltskin {
  int member;
};
/* Illegal - union name conflicts with previous struct name */
union Rumpelstiltskin {
  void *where_prohibited;
  long l;
};
/* Illegal - enum name conflicts with previous struct name */
enum Rumpelstiltskin {
  Jakob_Ludwig_Karl_Grimm,
  Wilhelm_Karl_Grimm
};
```

required array bounds missing

No size was specified for an array when one was needed to determine data storage space. The size can be explicit or implicit through the use of an initializer. Example:

```
extern char a[];    /* Legal - bounds not needed */
int i = sizeof(a); /* Illegal - need to know size */
function(b)
char b[];           /* Legal - bounds not needed */
{
  char c[];         /* Illegal - auto array needs bounds */
}
```

resource too big

See **link failed**.

stack frame too large

The local and temporary variable stack space required by a function exceeded 32768 bytes.

statements nested too deeply

THINK C allows nesting of at least 20 statements. `else ifs` do not introduce nested `if` statements and can be put together in arbitrarily long chains. Example:

```
/* maximum nesting is 20 */
if (condition_1) … if (condition_20) {/* then clause */}

/* arbitrary number of else-ifs can be chained */
if (condition_1)          {/* then clause 1 */}
else if (condition_2)     {/* then clause 2 */}
…
else if (condition_many) {/* then clause many */}
```

struct/union too large

The declared `struct/union` exceeded 32768 bytes of data storage.

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){…}      /* Illegal - switch of pointer */
switch(f){…}      /* Illegal - switch of float */
```

syntax error

There was a syntax error. Some common errors: too many `}`s, label without `:`, malformed expressions.

there are no void objects!

Declarations provide storage space for the variable declared. It is an error to have a variable that has no storage space. It is legal to have a pointer to `void`; this is a C idiom for a generic pointer. Example:

```
void nugatory;          /* Illegal */
void *generic_pointer; /* Legal */
```

too many formal parameters

THINK C allows you to have up to 25 formal parameters in a function definition. Example:

```
f(arg1, arg2…, arg25, arg26)  /* 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" }; /* Illegal */
struct { int a,b,c; } x[2] = { 1, 2, {3, 4 } };  /* Illegal */
```

too many macro parameters

Macros are allowed to have up to 25 arguments. Example:

```
#define macro(arg1, arg2, ..., arg26)  /* one too many args */
```

too many segments

Applications can have no more than 254 segments. Multi-Segment desk accesories and device drivers are limited to 31 segments.

undefined enumeration

An enumeration declaration refers to an enumeration tag that has not been defined. Example:

```
enum unknown colors;  /* Illegal */
```

undefined struct/union

A `struct/union` must be defined before an instance of it can be declared. However, a pointer to an undefined `struct/union` is legal since the size of the pointer is known and the size of the undefined `struct/union` is not needed. Example:

```
struct not_previously_defined s;          /* Illegal */
struct not_previously_defined *p;         /* Legal */
int i = sizeof(p);                        /* Legal */
int j = sizeof(*p);                       /* Illegal */
struct link_list_element{
  struct link_list_element *next;         /* Legal */
  struct link_list_element recursive;     /* Illegal */
};
```

undefined '*symbol*'

See **link failed**.

unexpected end-of-file

End of file was reached before a C language construct was completed. Example:

```
main(
/* EOF - end of file encountered before close parenthesis */
```

unions may not have bitfields

unions may not have bitfields. However, they may include structs that have bitfields. Example:

```
union {
  int i;
  unsigned int bits : 5;    /* Illegal */
}union_1;
typedef struct { unsigned int bits : 5; } bitfield_type;
union {
  int i;
  bitfield_type b;          /* Legal */
}union_2;
```

unknown instruction

(Assembly) The inline assembler doesn't recognize the mnemonic you've provided. This usually happens when you forget the period in an menonic. Example:

```
asm {
    movew    d0,d1
}
```

unknown struct/union member 'symbol'

In a . or -> expression either the left operand was not a struct/union, or the right operand was not the name of a member of the type of the left operand. Example:

```
function()
{
  int non_struct, *int_ptr;
  struct s1_struct{ int member_1; }s1, *p1;
  struct s2_struct{ int member_2; }s2, *p2;
  /* These are all illegal */
  s1.non_member;        /* non_member is not in s1_struct */
  p1 -> non_member;     /* non_member is not in s1_struct */
  non_struct.member_1; /* non_struct is not a struct */
  int_ptr -> member_1; /* int_ptr is not a struct pointer */
  s1.member_2;          /* member_2 is not in s1_struct */
}
```

unterminated comment

End of file was reached before end of comment was detected.

unterminated quote

Either a character constant or a string constant is missing its end quote. Example:

```
function()
{
  long file_type;
  /* Illegal - missing " after world */
  function("hello world);

  /* Illegal - missing ' after TEXT  */
  file_type = 'TEXT;
  /* Legal */
  file_type = 'TEXT';
}
```

use of struct/union/enum does not match declaration

A `struct/union/enum` tag was used in a declaration that conflicts with the orginal `struct/union/enum` tag declaration. `struct/union/enum` tags share the same name space. Example:

```
struct Rumpelstiltskin {
  int member;
};
union Rumpelstiltskin anIllegalUnion;   /* Illegal */
enum Rumpelstiltskin anIllegalEnum;     /* Illegal */
```

void function must not return a value

Example:

```
void in_partners_suit(condition)
int condition;
{
if (condition)
  return;     /* Legal */
else
  return(1); /* Illegal */
}
```

wrong number of arguments to 'symbol'

A call to the built-in Macintosh Toolbox or OS function *symbol* was made with the wrong number of parameters. Example:

```
SetPt(&aPoint, 3, 4, 8);  /* one too many arguments */
```

wrong number of arguments to macro 'symbol'

A macro was called with the wrong number of arguments. Example:

```
#define twice(x)  (x+x)
function()
{
  int i;
  i = twice(3,4); /* Illegal - macro called with 2 args */
}
```

wrong number of operands

(Assembly) You've probably forgotten something. Example:

```
asm {
    add.w   d0
}
```

wrong type(s) of operand(s)

(Assembly) Some instructions require operands of a specific type. Example:

```
asm {
    movea   d1,d0   /* movea can only move into address registers */
}
```

zero-sized object

An operator was used illegally on a zero-sized object. Example:

```
void vacuous()
{
  int i;
  void *nowhere, *erewhon;
  i = sizeof( vacuous() );     /* Illegal */
  i = nowhere - erewhon;       /* Illegal */
}
```

RMaker Reference
D

Introduction

Macintosh programs are designed around objects. Windows, menus, dialog boxes, and alerts are all objects that the Macintosh uses. You can build these objects on the fly in your programs, or you can load them in from the resource fork of your application. The advantage of storing these Macintosh objects as resources is that your program's function (the code) is separate from the user interface (the look).

RMaker is a resource compiler. It takes a textual specification of the objects to be used in a program and produces the resource data structures which are understood by the Macintosh Toolbox routines.

To learn how to use resources with THINK C projects, read the "Anatomy of a Project" section in Chapter 7. To learn about resources read *Inside Macintosh I*, Chapter 5, "The Resource Manager."

Topics covered in this appendix

- Using RMaker
- RMaker file format
- Predefined resource types

Using RMaker

To create a resource file, use the THINK C editor to create the RMaker source file. Then use the **Transfer...** command in the **File** menu to launch RMaker. RMaker displays a file selection dialog. Choose your RMaker source file, and RMaker will produce a resource file from it.

RMaker File Format

RMaker input is line-oriented and has a quite rigid syntax. The RMaker input file consists of an output specification followed by an arbitrary number of resource definitions separated by blank lines. Comments are also allowed, either as whole lines or as tags at the end of lines. Comment lines begin with an asterisk (`*`). Comments at the end of lines are preceded by two semicolons (`;;`).

All numbers in your file are decimal, except in certain resource type declarations (see below). To enter special characters, use a backslash followed by two hexadecimal digits. For example, the code for the Apple (🍎) symbol is `\14`. A ++ at the end of a line tells RMaker that the line is continued on the next line.

You can use the `/QUIT` directive to tell RMaker to quit after it builds your resource file. The `/NOSCROLL` directive tells RMaker not to scroll your file in its source window. For example:

```
/QUIT
/NOSCROLL
FileName
????SAMP

* resource declarations here...
```

Output file specification

RMaker files begin with the output specification. The name of the output file appears on the first line, followed by the file signature. The file signature is a four-character file type followed by a four-character creator. To learn more about file signatures, see *Inside Macintosh III*, Chapter 1, "The Finder Interface."

For example, if you want to name the output file `Sample` and give it the file type `????` and creator `SAMP`, the first two lines of the RMaker input file would be:

```
Sample      ;; the name of the output file
????SAMP    ;; the file type ???? and creator SAMP
```

The file signature line may be left blank, in which case the file type and file creator will be set to nulls (four bytes of 0 each). The output specification may be preceded by an arbitrary number of blank lines or comment lines.

If the file name begins with an exclamation point `(!)`, RMaker merges the resources into that file instead of creating a new file. In this case, if you don't supply a file signature, RMaker

You can use the `INCLUDE` *`filename`* statement to merge the resources from another resource file into the output file.

Resource declarations

The body of an RMaker source file consists of groups of resource declarations headed by a resource type clause.

(Note: in the following examples, the brackets `[]` enclose optional data. Words in *italics* describe user-supplied data.)

A resource type section begins with a TYPE declaration:

TYPE *resource type* [= *resource type*]

The = clause lets you define your own own resource types. If the optional = clause is not present, the first resource type must be a predefined type. If the clause is present, the resource type named there must be a predefined type.

The resource declarations come after the TYPE declaration. They look like this:

[*name*], ID [(*attributes*)]
type-specific resource data

Note that the comma separating the name from the ID must be included even if you don't name the resource. The attributes byte must be surrounded in parentheses. See the *Inside Macintosh I*, Chapter 5, "The Resource Manager" to learn more about resource attributes.

A blank line must follow the resource definition.

Predefined Resource Types

RMaker recognizes these 12 resource types.

```
'ALRT' - Alert
'BNDL' - Bundle
'CNTL' - Control
'DITL' - Dialog (or Alert) Item List
'DLOG' - Dialog
'FREF' - File Reference
'GNRL' - General
'MENU' - Menu
'PROC' - Procedure (contains code)
'STR ' - String
'STR#' - String List
'WIND' - Window
```

The following sections illustrate the different types of resource declarations:

'ALRT' - Alert Template

```
TYPE ALRT
    , 128          ;; the resource number
70 100 150 412     ;; rectangle for the alert (top left bottom right)
10                 ;; the resource ID for the item list
FFFF               ;; stages word (hex)

* always remember the blank line at the end of a resource
* definition - it is a required separator
```

'BNDL' - Bundle Template for Application

```
TYPE BNDL
    ,128     ;; resource number
SAMP 0       ;; creator for bundle
ICN#         ;; resource type (icon list)
0 128 1 129  ;; local ID 0 maps to ICN# resource 128, 1 to 129
FREF         ;; resource type (file reference)
0 128 1 129  ;; local to FREF mapping 0 to 128, 1 to 129
```

'CNTL' - Control Template

```
TYPE CNTL
    ,128     ;; resource number
MyControl    ;; title for control
10 10 20 20  ;; rectangle for control (top left bottom right)
Visible      ;; may also be Invisible
0            ;; CDEF proc ID
0            ;; reference constant (defines control type)
0 100 0      ;; minimum value, maximum value, initial value
```

'DITL' - Dialog (or Alert) Item List

```
TYPE DITL
    ,10                    ;; resource number
9                          ;; number of items in the item list

button                     ;; enabled button items (enabled by default)
20 20 40 100               ;; rectangle (window-relative coordinates)
Cancel                     ;; text in the button

radioButton                ;; radio button item
50 20 70 120               ;; rectangle (includes button and text)
Push Me                    ;; the text goes to the right of the button

radioButton disabled       ;; dimmed radio button item
50 20 70 120               ;; rectangle (includes button and text)
```

```
Can't Push Me              ;; you can't push a disabled button

checkBox                   ;; check box item
80 20 100 120              ;; rectangle (includes check box and text)
Check Me                   ;; the text goes to the right of the box

staticText                 ;; static text item
20 120 40 320              ;; rect of text
This text would get placed next to the Cancel button ;; the text

editText Disabled          ;; disabled editable text item
50 140 90 320              ;; rect of the box for editable test
initial string             ;; initial edit text

editText                   ;; editable text item (enabled by default)
100 140 120 320            ;; rectangle
you can edit this text     ;; the initial string for editable item

iconItem                   ;; for the display of an icon
100 100 132 132            ;; rectangle should be 32x32
3                          ;; resource ID for icon (Type ICON)

picItem                    ;; to display of a Quickdraw picture
30 100 20 200              ;; display rectangle (picture will be scaled)
57                         ;; resource ID for picture (Type PICT)

userItem                   ;; a user-defined item
30 40 80 90                ;; the rectangle
```

'DLOG' - Dialog Template

```
TYPE DLOG
    ,128                   ;; the resource number
My Dialog Box              ;; a message
70 100 150 412             ;; the rectangle (top left bottom right)
Visible NoGoAway           ;; or Invisible or GoAway
0                          ;; the dialog definition ID
0                          ;; the refCon, available to the user
10                         ;; the resource ID for the dialog item list
```

'FREF' - File Reference

```
TYPE FREF
    ,128                   ;; the resource number
APPL 0                     ;; the file type and local ID
```

'GNRL' - General

'GNRL' is used to define your own resource types and define their format. The resource's format is constructed from "elements." The elements available are:

- `.P` Pascal string
- `.S` String without a leading length byte
- `.I` Decimal integer
- `.L` Decimal 32-bit integer
- `.H` Hexadecimal integer
- `.R` Read the given resource from the given file.
  R takes the arguments `filename  resource TYPE resource ID`

```
TYPE ICN# = GNRL          ;; define the type ICN#
    ,128                  ;; the resource ID
.H                        ;; hexadecimal data follows
0001 8000 0002 4000       ;; ICN#'s need 2 icons (icon and
0003 C000 0004 2000       ;; mask) of 32x32 bits each, or 32
…                         ;; lines of two longwords apiece.
FFFF FFFF FFFF FFFF
```

'MENU' - Menu

```
TYPE MENU
    ,10                   ;; the resource number (Menu ID)
MyMenu                    ;; the menu title
First Item                ;; the first menu item
Second Item /S            ;; the second menu item with Command-S
(Third Item               ;; the third item ("(" disables items)
(-                        ;; a gray line (this is item #4)
Fifth Item                ;; the fifth item
```

'PROC' - Procedure (contains code)

```
TYPE PROC
    ,128                  ;; the resource number
Filename                  ;; the code from this file will get placed
                          ;; in the resource
```

'STR ' - String

```
TYPE STR                  ;; spelled 'STR ' - trailing space  required!
    ,128                  ;; the resource number
My Wild Irish Rose        ;; the string assigned to the resource 128
```

'STR#' - String List

```
TYPE STR#
    ,128                    ;; the resource number
2                           ;; the number of strings in the list
The First String
And the second string       ;; the two strings in the list
```

'WIND' - Window

```
TYPE WIND
    ,128                    ;; the resource ID
My Window                   ;; the window title
40 40 200 472               ;; the window rect (top left bottom right)
Visible GoAway              ;; or Invisible or NoGoAway
0                           ;; the window definition ID
0                           ;; refCon, a long word available to user
```

Index

License Agreement

License Agreement

This manual and the software described in it were developed and are copyrighted by Symantec Corp. (Symantec) and are licensed to you on a non-exclusive, non-transferable basis. Neither the manual nor the software may be copied in whole or in part except as follows:

1) You may make backup copies of the software for your use provided that they bear Symantec's copyright notice.

2) You have the right to include object code derived from the libraries in programs that you develop using the software and you also have the right to use, distribute and license such programs to third parties without payment of any further license fees, so long as a copyright notice sufficient to protect *your* copyright in the software in the United States or any other country is included in the graphic display of your software and on the labels affixed to the media on which you software is distributed.

You may not in any event distribute any of the source files provided or licensed as part of the software. You may use the software at any number of locations so long as there is no possibility of it being used at more than one location at a time.

Symantec's Plain Language License Statement

Symantec is concerned with how you copyright your software only in the case where you use object code of libraries which Symantec provides in source form (MacTraps does not fall into this category). These libraries may be included in your program so long as a copyright notice that will protect *your* copyright in the software is in the "About box" of your software and on the disk labels, as specified in the license agreement. You are not required to include a specific Symantec copyright notice except if your copyright does not satisfy the above requirement. This is only an explanation of the License Agreement. All terms and conditions of the License Agreement apply.

Limited Warranty on Media and Manuals

If you discover physical defects in the media on which this software is distributed or in the manuals distributed with the software, Symantec will replace the media or manuals at no cost to you provided that you return the defective materials along with a copy of your receipt to Symantec or to an authorized Symantec dealer during the 60-day period following your receipt of the software.

Limited Warranty on the Product

Symantec warrants that the software will perform substantially as described in the User's Manual. If within 60 days of receiving the software, you give written notification to Symantec

of a significant, reproducible error in the software which prevents operation, and provide a written description of the possible problem along with a machine readable example, if appropriate, Symantec will either provide you with corrective or workaround instructions, a corrected copy of the software, a correction to the User's Guide and Reference Manual, or Symantec will refund your purchase price upon return of all copies of the software and documentation together will a copy of your receipt. This warranty extends only to you and shall be void if the software has been tampered with, modified, or improperly used, or if the software is used on hardware other than the Apple Macintosh™ Computer.

EXCEPT FOR THE LIMITED WARRANTY DESCRIBED ABOVE, THERE ARE NO WARRANTIES TO YOU OR ANY OTHER PERSON OR ENTITY FOR THE PRODUCT EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OR MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE. ALL SUCH WARRANTIES ARE EXPRESSLY AND SPECIFICALLY DISCLAIMED. Some states do not allow the exclusion of implied warranties or limitations on how long they last, and you also may have other rights that vary from state to state. IN NO EVENT SHALL SYMANTEC BE RESPONSIBLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR SIMILAR DAMAGES OR LOST DATA OR PROFITS TO YOU OR ANY OTHER PERSON OR ENTITY REGARDLESS OF THE LEGAL THEORY, EVEN IF WE HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. Some states do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you. The warranty and remedies set forth are exclusive and in lieu of all others, oral or written, express or implied.

SYMANTEC MAKES NO WARRANTY OF THE PERFORMANCE OF THE LIBRARIES WHEN USED IN YOUR SOFTWARE. YOU AGREE TO INDEMNIFY SYMANTEC FROM ALL CLAIMS BY THIRD PARTIES ARISING IN CONNECTION WITH THE USE OF YOUR SOFTWARE.

General Terms This license states the entire agreement between the parties and supercedes all other communications between the parties relating to this License, which shall be governed and construed in accordance with the laws of the State of California. You agree to bring any proceeding to enforce or construe this License or involving the performance of the software only in a federal or state court residing in the State of California. The prevailing party in any such proceedings shall be entitled to recover its attorneys' fee and litigation expenses in addition to other appropriate relief. If any provision of this License by Symantec shall be held to be unenforceable such holding shall not affect the enforceability of any other provision hereof. Waiver of any breach of this License by Symantec shall not be considered a waiver of any other or subsequent breach. the licensed software is a unique and valuable asset of Symantec and Symantec has the right to seek whatever equitable and legal redress which may be available to it for your breach of the provisions of the License.

"Lightspeed" is a trademark of Lightspeed, Inc., and is used with its permission.