MACINTOSH
PROGRAMMER’S
WORKSHOP

Building and Managing Programs in MPW

Second Edition

For MPW version 3.4

030-9408-A
Developer Press

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Appendix B Disassembler Routines B-1

Appendix C The Rez Language C-1

Figures, Tables, and Listings

About This Book

This book, *Building and Managing Programs in MPW*, is part of the series of manuals that describe how to use the Macintosh Programmer's Workshop (MPW) Development Environment.

The MPW Development Environment includes the MPW Shell, the application that creates this environment, and the MPW Tool Suite, a collection of MPW tools, scripts, libraries, and interfaces that work with the MPW Shell. This manual describes how to use version 3.4 or later of this software to build, manage, optimize, and test your programs. You can use MPW to build programs in all three Macintosh runtime environments: PowerPC™, classic 68K, and CFM-68K.

You can expand your development environment even further by writing your own tools and scripts. You can also get other MPW-compatible software from APDA to make your program development easier.

This book assumes you have basic knowledge of the MPW environment, including how to use the MPW Shell, manipulate windows, edit text, and manipulate files and directories. If you are unfamiliar with MPW, read *Introduction to MPW* before starting this book.

This book also assumes basic knowledge of programming and the build process (compiling, linking , and so on). Most of the programming examples in this book are in C; in a few cases they are in Pascal. While reading this book you may want to consult the companion volumes *Macintosh Runtime Architectures* and the *MPW Command Reference*.

What's in This Book

This book describes all aspects of building and managing your software programs. It does not cover specific aspects of the Macintosh runtime architecture that affect how you design your programs. For that information, you should consult *Macintosh Runtime Architectures* or the various volumes of *Inside Macintosh*.

The first six chapters describe the build process for basic types of Macintosh programs:

- Chapter 1, "Building Macintosh Programs," gives an overview of the build process, the runtime architectures available, and the types of programs you can build.
- Chapter 2, "Building PowerPC Runtime Programs," describes how to build applications and shared libraries in the PowerPC runtime environment.
- Chapter 3, "Building Classic 68K Runtime Programs," describes how to build classic 680x0 runtime applications.
- Chapter 4, "Building CFM-68K Runtime Programs," describes how to build CFM-68K runtime applications and shared libraries.
- Chapter 5, "Building Fat Binary Files," describes how to build programs that combine code from different runtime architectures.
- Chapter 6, "Creating Noncode Resources and Manipulating Resources," describes the process of building noncode resources—the resources that a Macintosh program uses to define windows, menus, dialog boxes, controls, icons, and the cursor. This chapter also explains how to add or delete resources from the resource fork of a file.

You can learn more about specific aspects of the build process in these chapters:

- Chapter 7, "Source Code Porting Checklists," describes factors to keep in mind when you port source code between Macintosh runtime architectures or from an entirely different environment.
- Chapter 8, "More About Linking," includes information about import library version checking, segmentation, link maps, the Lib tool, and other useful linking details.
- Chapter 9, "Standard Libraries," gives information about the libraries included with MPW.
- Chapter 10, "Make and Makefiles," describes how to create makefiles and use them to automate the build process.

You can learn how to write and build different types of programs, including your own MPW tools, by reading these chapters:

- Chapter 11, "Writing Stand-Alone Code," describes stand-alone code, its uses and limitations, and build procedures for the PowerPC and classic 680x0 versions.
- Chapter 12, "Building PCI Device Drivers," describes how to build PCI drivers for PowerPC-based Macintosh computers.

- Chapter 13, "Writing and Building MPW Tools," provides guidelines for writing an integrated MPW tool and describes the utility routines used by the tools that run in the MPW Shell. Some of these routines may also be used by applications.
- Chapter 14, "Creating Commando Dialog Boxes for Tools and Scripts," describes how to build a Commando dialog box for a tool or script you have created.
- Chapter 15, "Building SIOW Applications," describes how to use the Simple Input/Output Window (SIOW) package, which allows programs not originally written for Macintosh computers to read from, and write to, a window.

You can learn to manage large programs as well as test the performance of your program by reading these chapters:

- Chapter 16, "Managing Projects With Projector," discusses special tools and scripts you can use to control your source files. Projector allows several users to work simultaneously on a large project by regulating files checked in and out of the Projector database.
- Chapter 17, "Measuring Performance," discusses the various performance tools available in the MPW environment. Using these tools, you can determine how often a routine is called and how much time is spent executing that routine.

Reference material is provided in the following appendixes:

- Appendix A, "The 'ckid' Resource Format," describes the format of the `'ckid'` resource, which is used to identify Projector files.
- Appendix B, "Disassembler Routines," describes routines you can use to disassemble PowerPC and MC68000 machine code.
- Appendix C, "The Rez Language," provides a complete detailed description of the Rez language, which you can use to write resource description files.
- Appendix D, "Apple Event Support," describes the support provided by the MPW Shell for Apple events that enable the remote execution of tools and scripts.
- Appendix E, "Troubleshooting," may be helpful if you encounter problems during your build.

A glossary and index are provided in the back of the book.

How to Use This Book

To use this book effectively, you must be familar with the MPW Shell. You should be able to work with windows in MPW, edit text, manipulate files, and write simple scripts. You should be familiar with the MPW `Startup` file, command substitution, and redirection, and you should know how to use the MPW Shell variables. If you need to learn this information, you should read *Introduction to MPW*.

For more information about the tools introduced in this book, you should see the *MPW Command Reference*. For specific information about programming for Macintosh computers, you should consult the various volumes of the *Inside Macintosh* series and also the book *Macintosh Runtime Architectures*.

Conventions Used in This Book

This book uses various conventions to present certain types of information. Some information, such as command line options, uses special formats so that you can scan it quickly.

Special Fonts

All code listings, reserved words, command options, resource types, and the names of actual libraries are shown in Letter Gothic (`this is Letter Gothic`).

Words that appear in **boldface** are key terms and are defined in the glossary.

Command Syntax

This book uses the following syntax conventions:

`literal`	Letter Gothic text indicates a word that must appear exactly as shown. Special MPW symbols (∂, •, &, ƒ, `, ∑, and so on) must also be entered exactly as shown.
italics	Italics indicate a parameter that you must replace with anything that matches the parameter's definition.
[ ]	Brackets indicate that the enclosed item is optional.
. . .	Ellipses (. . .) indicate that the preceding item can be repeated one or more times.
|	A vertical bar (|) indicates an either/or choice.

Types of Notes

This book uses three types of notes.

Note
A note like this contains information that is useful but that you do not have to read to understand the main text. ◆

IMPORTANT
A note like this contains information that is crucial to understanding the main text. ▲

▲ **WARNING**
Warnings like this indicate potential problems that you should keep in mind as you build your programs. Failure to heed these warnings could result in system crashes or other runtime errors. ▲

For More Information

The *Apple Developer Catalog* (ADC) is Apple Computer's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in developing applications on Apple computer platforms. Customers receive the *Apple Developer Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. ADC offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *Apple Developer Catalog*, contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	ORDER.ADC
Internet	order.adc@applelink.apple.com

Building Macintosh Programs

Contents

This chapter describes the various types of Macintosh programs you can build using MPW and gives an overview of the build process. The general build process is the same for all types of programs: you compile the source files and then link them to any required libraries. However, many variations are possible given the types of programs and runtime architectures to choose from. This chapter gives an overview of the build process and describes

- the types of programs you can build
- the three runtime architectures available
- static linking versus dynamic linking of libraries
- the approaches you can use for building programs
- tips for improving build times

If you are new to MPW, you should read the entire chapter. After you understand the basic concepts, you should proceed to the build chapter that focuses on the runtime architecture you intend to use. The runtime architectures available are

- **PowerPC™ runtime architecture,** designed to run on PowerPC-based Macintosh computers
- **classic 68K runtime architecture,** the original Macintosh runtime architecture, designed to run on 68K-based Macintosh computers
- **CFM-68K runtime architecture,** which allows 68K-based Macintosh computers to use some features of the PowerPC runtime (notably the Code Fragment Manager and shared libraries)

After reading the appropriate build chapter, you can go to other chapters that provide specific information about building different types of programs.

If you have worked with MPW before, you should read the section “Macintosh Runtime Architectures,” beginning on page 1-4, and then proceed to the build chapter devoted to the architecture you intend to use.

What Is a Macintosh Program?

A Macintosh **program** consists of code (CPU instructions) and data. The code and data can be stored in either the data fork or the resource fork of a program file (or possibly both).

The location of the program's executable code depends upon the type of program and the runtime architecture used. Programs built for the PowerPC runtime store all their code in the data fork. Classic 68K runtime programs store their code in the resource fork as **code resources:** for example, `'CODE'`, `'DRVR'`, or `'INIT'` resources. CFM-68K runtime programs are a hybrid, and, depending on the type of program, may store their code in either the data fork or the resource fork.

Data used by the program, as opposed to data on which the program operates, is stored in **noncode resources:** for example, `'WDEF'`, `'SIZE'`, `'DITL'`, or `'DLOG'` resources. You use the resource compiler Rez or a resource editor to create noncode resources. Chapter 6, "Creating Noncode Resources and Manipulating Resources," describes how to write and build noncode resources.

Note
The term *code resource* denotes any resource (which is found in the resource fork of a file) that contains executable code. A `'CODE'` resource, however, refers specifically to a resource of type `'CODE'`. ◆

Macintosh Runtime Architectures

With MPW you can build Macintosh programs for any of three different runtime architectures. Each architecture has unique characteristics that may influence your decision regarding which one to use. In addition, you can also build fat binary programs that contain code for multiple architectures.

PowerPC Runtime Architecture

The PowerPC runtime architecture was designed for Power Macintosh computers using the PowerPC family of microprocessors. The PowerPC runtime depends upon the **Code Fragment Manager,** which handles code and data (stored as **fragments**) differently from the Segment Loader on 68K-based Macintosh computers. The Code Fragment Manager also allows you to use shared libraries, also known as *dynamically linked libraries*. All executable code is stored in the data fork, while noncode resources are stored in the resource fork. PowerPC runtime programs can run only on PowerPC-based computers.

PowerPC is the platform for all future Macintosh development. You should choose the PowerPC runtime architecture if any of the following is the case:

- You want to get maximum performance from your program.
- You need to use shared libraries (for example, for OpenDoc development).
- You are creating code that will evolve over time and will require taking advantage of the latest technological developments.

For more information about the PowerPC runtime architecture, see *Inside Macintosh: PowerPC System Software*.

Classic 68K Runtime Architecture

The classic 68K runtime is the original Macintosh runtime architecture, designed for computers running a Motorola 68000-series microprocessor. Classic 68K runtime programs are handled as **segments,** and they are stored as `'CODE'` resources in the resource fork. Classic 68K runtime code cannot use shared libraries. However, classic 68K runtime code can run transparently under emulation on PowerPC-based computers.

You may choose the classic 68K runtime architecture if any of the following is the case:

- You wish to build for the widest range of Macintosh computers (since classic 68K runtime programs can run on both 68K-based and PowerPC-based computers).
- Your program does not rely on CPU-intensive tasks (such as filtering or signal processing) that could be better served with the PowerPC microprocessor.
- You are modifying an application that was originally written for the classic 68K runtime architecture.
- Your program does not need to use shared libraries.

For more information about the classic 68K runtime architecture, see the *Inside Macintosh* series, especially the *Overview*, *Processes*, and *Memory* books.

CFM-68K Runtime Architecture

The **CFM-68K runtime architecture** is a hybrid of the PowerPC runtime and the classic 68K runtime that uses both the Segment Loader and the Code Fragment Manager. Its major advantage is that it allows 68K-based Macintosh computers to use shared libraries. However, CFM-68K runtime programs cannot run under emulation on PowerPC-based computers. CFM-68K runtime applications are stored as `'CODE'` resources, as in the classic 68K runtime, but shared libraries are stored as fragments in the data fork.

You should choose the CFM-68K runtime architecture if any of the following is the case:

- You are developing an application to run on both PowerPC-based and 68K-based computers and would like to use shared libraries for both versions.
- You are a library developer and would like to develop shared libraries for both PowerPC-based and 68K-based computers.
- You are using or planning to use OpenDoc with your 68K runtime application. Because OpenDoc requires shared library support, only CFM-68K runtime applications can take advantage of OpenDoc on 68K-based Macintosh computers.

For details of the CFM-68K runtime architecture, see *Macintosh Runtime Architectures*.

Fat Binary Files

Because of the way data is stored in Macintosh files, it is possible to create a **fat binary program** that contains code for multiple architectures. For example, you can create a fat binary application that contains both PowerPC runtime and classic 68K runtime code. When run on a PowerPC-based computer, the PowerPC runtime code executes; on a 68K-based computer, the classic 68K runtime code executes. You can also combine PowerPC code with CFM-68K code (for use with OpenDoc, for example). Fat binary programs require a little more work to create, but they allow greater flexibility for the user. For example, a user can store a fat application on a portable hard drive and then move it between a 68K-based computer and a PowerPC-based computer.

Fat binary programs are larger than traditional single architecture programs since they must contain additional executable code. Noncode resources are shared by the executable code, however, and do not add to the size overhead.

For detailed information on how to build fat binary files, see Chapter 5, "Building Fat Binary Files."

Building a Macintosh Program

This section describes the types of Macintosh programs you can build and gives an overview of the build process. Table 1-1 summarizes the types of programs you can build for Macintosh computers.

Table 1-1 Types of Macintosh programs

Program type	**Where stored**			**Characteristics**
	PowerPC	Classic 68K	CFM-68K	
Application	Data fork	`'CODE'` resources	`'CODE'` resources and `'rseg'` resources	An event-driven program that initializes Macintosh Toolbox managers. Applications that can run in the foreground take control of the Macintosh computer, implement a user interface, and return to the Finder.
Shared library	Data fork	N.A.	Data fork	A code fragment that exports code and data for use by other fragments. A shared library is stored independently of client applications and can be shared by several at one time. The Code Fragment Manager links shared libraries to clients at runtime.

continued

Table 1-1 Types of Macintosh programs (continued)

Program type	Where stored: PowerPC	Where stored: Classic 68K	Where stored: CFM-68K	Characteristics
MPW tool	Data fork	`'CODE'` resources	N.A.	A program that runs in the MPW Shell and uses a line-oriented interface. Tools can be complex, like compilers and linkers, or simple, like a one-line program that prints a string. Tools must not initialize Toolbox managers (except for using `InitGraf`) but can declare global variables. A 68K-based tool can be divided into multiple segments and can use the jump table. For more information, see Chapter 13, "Writing and Building MPW Tools." MPW tools you write yourself can be given a user interface and run outside of MPW by converting them into SIOW applications.
Device driver	Data fork	`'DRVR'` resource	N.A.	A collection of routines (called by an application or by the operating system) used to communicate with peripherals. See Chapter 12, "Building PCI Device Drivers," for more information
Stand-alone code	Resource fork	Resource fork	N.A.	Code that implements or extends Toolbox, operating system, or application functions. Stand-alone code is called by an application or by the operating system. For more information, see Chapter 11, "Writing Stand-Alone Code."
SIOW application	Data fork	`'CODE'` resources	`'CODE'` resources and `'rseg'` resources	A program written using standard C, Pascal, or Fortran I/O routines or an MPW tool that runs in its own window and allows the use of the File, Edit, Font, and Size menus. For additional information, see Chapter 15, "Building SIOW Applications."

About Shared Libraries

Traditionally libraries are attached to the application code at link time. Such libraries, called **static libraries,** are physically included as part of the application. If you make any changes to a library, you must relink in order to effect the changes.

Shared libraries, on the other hand, exist as files separate from the actual application. During the link process, the linker adds references to the libraries required but does not attach any code. Shared libraries are sometimes called *dynamically linked libraries* (DLLs).

Shared libraries come in two forms:

- **Import libraries,** which are automatically loaded at application launch time. The Code Fragment Manager determines which libraries are needed and binds the references to the application, even if another application is using the same library.
- **Drop-in additions,** which must be specifically loaded by the client application.

IMPORTANT

Only runtime architectures that support the Code Fragment Manager can use shared libraries (that is, only PowerPC runtime and CFM-68K runtime). ▲

Shared libraries have several advantages over static libraries:

- Multiple applications can share a single shared library. This feature saves both hard disk space and RAM.
- You can modify a shared library without having to recompile the client application. Bug fixes or modifications can be easily implemented by updating the proper shared library.
- Using drop-in additions, you can store less commonly used routines in shared libraries (a spelling checker, for example) and load them into memory only when needed.

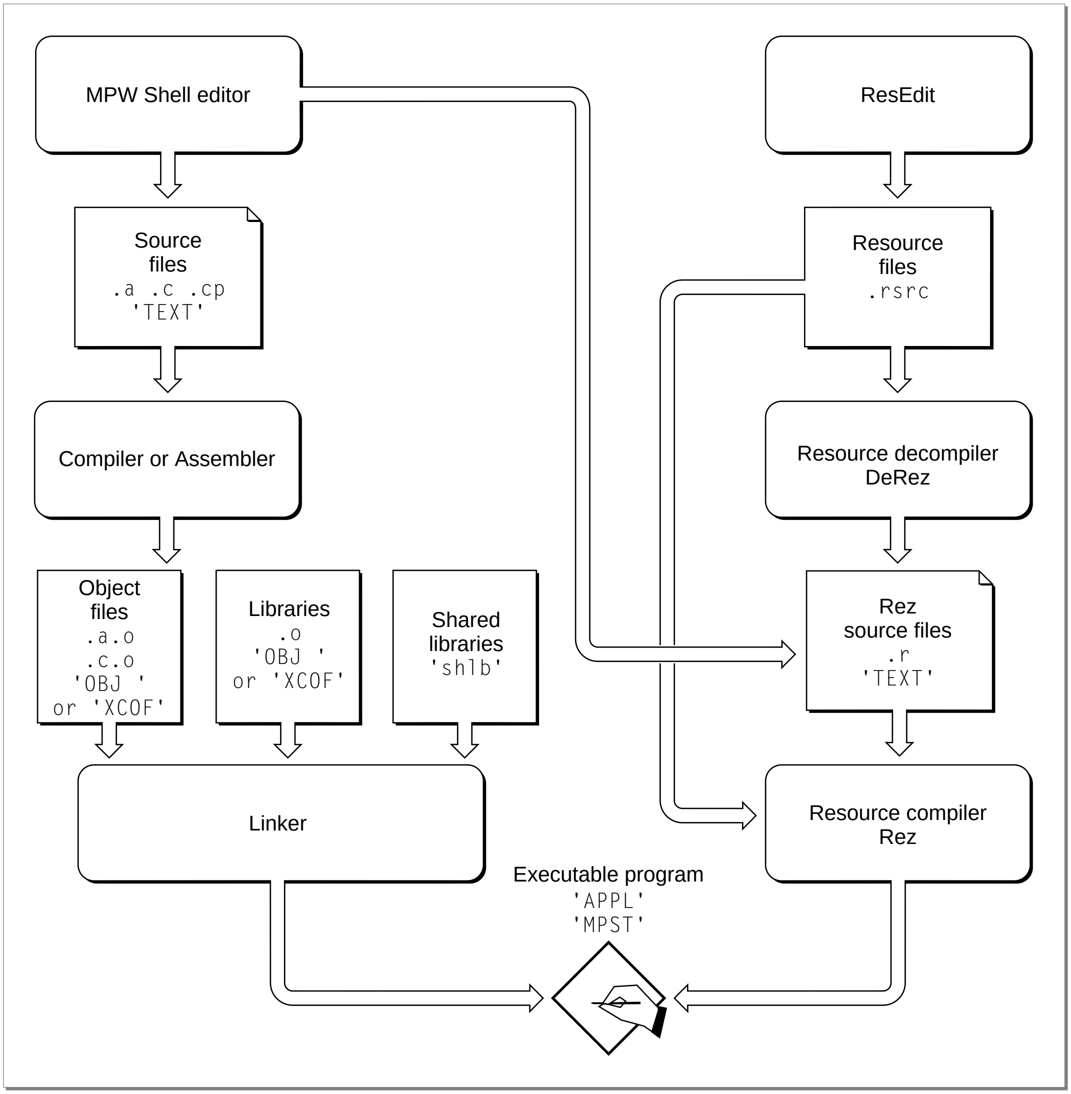
For more information about shared libraries, see *Inside Macintosh: PowerPC System Software* and the CFM-68K runtime architecture information in *Macintosh Runtime Architectures.*

Building a Macintosh Application

The general build procedure for a Macintosh application is shown in Figure 1-1. Since the build tools and options vary depending on which runtime architecture you are using, you should check the appropriate build chapter (for PowerPC runtime, classic 68K runtime, or CFM-68K runtime) for more specific information.

Note that the build procedure varies slightly for other types of code (MPW tools, drivers, and so forth). For more details, you can use the cross-references in Table 1-1 to determine the appropriate chapter to read.

Figure 1-1 The build process for an application



The Order in Which You Build an Application

As Figure 1-1 shows, the procedure for building a Macintosh application actually involves two parallel processes:

- building executable code using a compiler and linker
- building noncode resources using Rez or a resource editor

It does not matter which of these processes you complete first as long as you remember that Rez overwrites everything in the resource fork when writing output to a file. This affects even PowerPC runtime code since a `'cfrg' 0` resource is needed for execution.

To avoid this you must either make sure you execute Rez before doing any linking or else use the `-a` Rez option to append the resources to the file.

The following sets of commands produce the same executable file.

```
MrC mooProg.c -o mooProg.c.o    /* build process using Rez first */
Rez mooProg.r -o mooProg        /* no need to use the -a option */
PPCLink  ∂
        mooProg.c.o ∂
        "{SharedLibraries}"InterfaceLib ∂
        "{SharedLibraries}"StdCLib ∂
        "{PPCLibraries}"StdCRuntime.o ∂
        "{PPCLibraries}"PPCCRuntime.o ∂
        -o mooProg

MrC mooProg.c -o mooProg.c.o    /* build process using PPCLink first */
PPCLink  ∂
        mooProg.c.o ∂
        "{SharedLibraries}"InterfaceLib ∂
        "{SharedLibraries}"StdCLib ∂
        "{PPCLibraries}"StdCRuntime.o ∂
        "{PPCLibraries}"PPCCRuntime.o ∂
        -o mooProg
Rez mooProg.r -a -o mooProg     /* note use of the -a option */
```

Note that when ILink writes its output (68K-based code) to an existing file, it replaces only those executable resources (`'CODE'`, `'DRVR'`) of the type it is creating.

Note
It is a good idea to use the `-a` Rez option at all times to avoid unintentionally overwriting existing resources. ◆

Linking to Libraries

Many of the routines that you call from your program cannot be implemented without calling on routines stored in standard libraries. Standard libraries add the following:

- C, C++, or Pascal routines (for example, the `printf`, `open`, and `close` routines in C).
- Toolbox or operating-system routines that are not in ROM.
- Initialization for special program types. For example, you may need to link to special libraries when building MPW tools.
- Initialization for global data (for 68K-based code only).
- Special code that allows operating-system routines, which are register-based, to be called from stack-based languages such as C and Pascal. This type of library code is often referred to as *glue*.

The libraries you need to link with vary depending on the runtime architecture, the type of program, and the types of routines you call from your source code. The specific build chapters describe the standard libraries included for each runtime architecture. For more detailed information, you can refer to Chapter 9, "Standard Libraries."

About File Types and Creators

Every Macintosh file is assigned a **file type** by the build tools during the build process. The file type identifies the type of program (for example, an application, shared library, or MPW tool) so that other software (including system software) knows how to handle it.

Each step of the build process can produce a different file type, and keeping track of the types can be confusing at times. Tables 1-2, 1-3, and 1-4 describe file types that the build tools assign to output files.

Table 1-2 Default PowerPC file types

Program type	Output from compiler	Output from linker
Application	`'XCOF'`	`'APPL'`
Shared library	`'XCOF'`	`'shlb'`
Static library	`'XCOF'`	`'XCOF'`

Table 1-3 Default classic 68K file types

Program type	Output from compiler	Output from linker
Application	`'OBJ '`	`'APPL'`
Static library	`'OBJ '`	`'OBJ '`

Table 1-4 Default CFM-68K file types

Program type	Output from compiler	Output from linker	Output from MakeFlat
Application	`'OBJ '`	`'APPL'`	N.A.
Shared library	`'OBJ '`	`'SPEF'`	`'shlb'`

▲ **WARNING**
You should never change file types after linking a CFM-68K runtime fragment. CFM-68K applications and shared libraries have very different file structures, and changing the file type between type `'APPL'` and `'SPEF'` (or vice versa) can create serious runtime problems. ▲

Files can also be assigned a creator. Many applications have a signature creator type that the Finder uses to identify which files can be used with it. For example, if you wished to make sure the MPW Shell can recognize a tool you built for it, you should assign your tool the creator `'MPS '` and the file type `'MPST'`.

Creators are usually assigned using a linker option (check the appropriate build chapter or the *MPW Command Reference*) or you can use the SetFile tool. You can choose any four-letter combination for a creator type, although creators with all lowercase letters are reserved for use by the system. You should register a creator with Apple Developer Technical Support for every application you create. You can register a creator or view currently registered creators at http://dev.info.apple.com/cftype/main.html.

If you do not assign a creator, the linker assigns the default type, `'????'`.

Different Approaches to Building in MPW

There are three different approaches to building a program: using the command line, using the Build menu, and using the Make tool. Which approach you use depends on how often you plan to do the build and on the complexity of your program.

Using the Build menu and using the Make tool involve creating a makefile. A **makefile** is a text file that contains dependency information for the files that make up your program. It also specifies the commands required to build the program. Makefiles help automate the build process and help to minimize the time required to rebuild programs.

Using the Command Line

To build a program using the command line, you enter commands in any MPW window to compile your source files and resource description files, and link the object files with the appropriate libraries. For example, the following sequence illustrates the command line method:

```
MrC mooProg.c -o mooProg.c.o
PPCLink mooProg.c.o ∂
    "{SharedLibraries}"InterfaceLib ∂
```

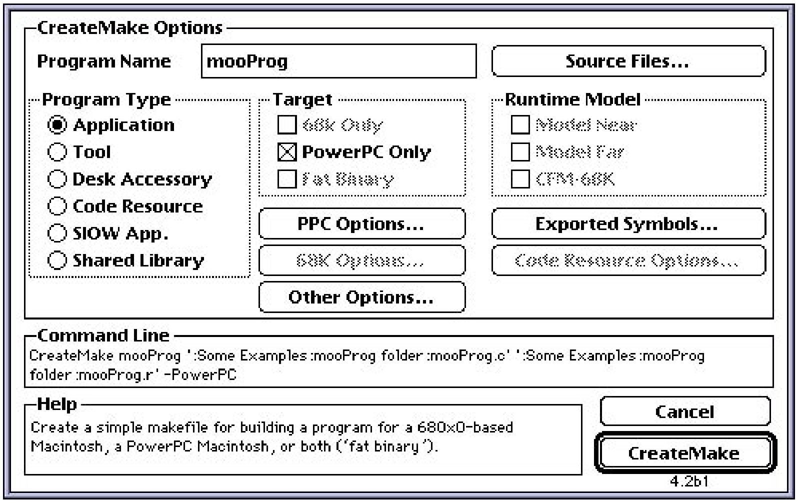
```
    "{SharedLibraries}"StdCLib ∂
    "{PPCLibraries}"StdCRuntime.o ∂
    "{PPCLibraries}"PPCCRuntime.o ∂
    -o mooProg
Rez mooProg.r -a -o mooProg
```

The command line method is best for one-time only builds or very simple programs.

Using the Build Menu

You can choose menu items from the Build menu to build a program. When you choose Create Build Commands from the Build menu, a dialog box appears that allows you to specify your program's source files, the type of program to build, and the runtime architecture you are building for. Based on this information, the CreateMake tool creates a makefile that contains file dependency information and the commands required to build your program. The CreateMake tool automatically selects the appropriate libraries to link with. Figure 1-2 shows the CreateMake dialog box. You can choose Show Build Commands or Show Full Build Commands from the Build menu to show the contents of the makefile.

Figure 1-2 The CreateMake dialog box



To actually build the program, you choose Build or Full Build from the Build menu (you can also use the keyboard shortcut Command-B).

Using the Build menu has some drawbacks:

- You cannot specify all the compiler, linker, or Lib tool options.
- You cannot specify dependencies between source files and header files.
- You cannot use the Build menu to build all program types.
- If you use transcendental numbers in your program, you may have to edit the makefile to include the appropriate libraries.

Despite some limitations, using the Build menu is convenient for building simple programs or to build the sample programs shipped with MPW.

Using the Make Tool

Instead of using the Build menu to create a makefile, you can also write your own. Then, you enter the Make command, specifying the makefile you created as your input file. The Make tool analyzes the dependencies listed in the makefile, determines what commands need to be executed, and displays these commands. To build your program, you execute the displayed commands.

If you prefer, you can use the BuildProgram tool instead of Make. BuildProgram analyzes dependencies and automatically executes the needed commands rather than simply displaying them.

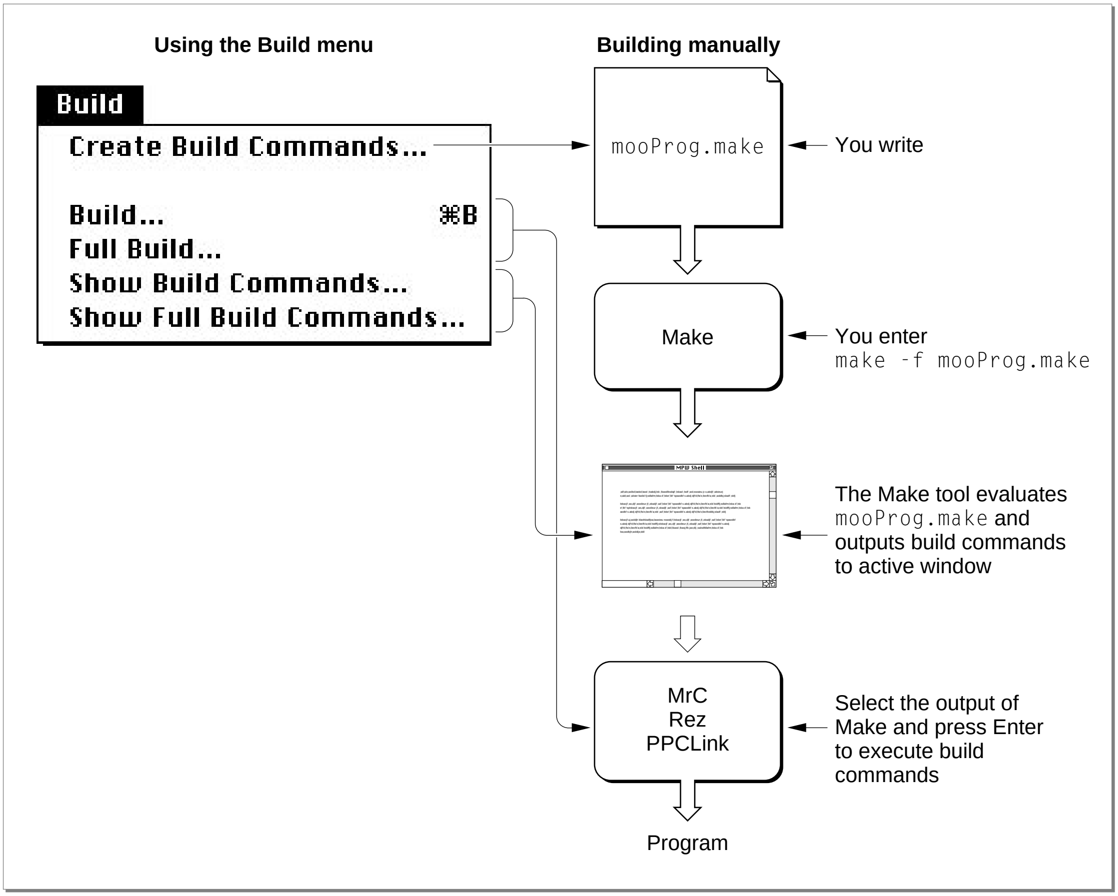
Creating a makefile is the preferred approach to use for any substantial development project because you have complete flexibility in selecting compiler, linker, or Lib tool options and specifying dependencies. You can also build any program type.

If you have never used makefiles, you might want to begin by using the Build menu approach. You can choose Show Build Commands to examine the makefile that CreateMake creates for you. You can also edit this makefile by adding dependency rules and using Make variables to create a more general makefile that can serve as the basis for more complex builds. See Chapter 10, “Make and Makefiles,” for detailed information about the form and content of makefiles.

How the Build Approaches Interrelate

Figure 1-3 shows how using the Build menu relates to the makefile method. As you can see, Make and the makefile form the cornerstone of both methods.

Figure 1-3 Relationship between build approaches



Where to Go Next

The material in this chapter provides only an overview of the build process. From here you should read the chapter that focuses on the runtime architecture you plan to use:

- Chapter 2, "Building PowerPC Runtime Programs."
- Chapter 3, "Building Classic 68K Runtime Programs."
- Chapter 4, "Building CFM-68K Runtime Programs."

If you want to build fat binary programs, you should read Chapter 5, "Building Fat Binary Files," in addition to the chapters that discuss the runtime programs you plan to combine.

For information about writing resources and using Rez, read Chapter 6, "Creating Noncode Resources and Manipulating Resources."

If you are importing source code from another environment, or are porting between different Macintosh runtime architectures, you should read Chapter 7, "Source Code Porting Checklists."

For more in-depth discussion of topics covered in this chapter, you should read the following:

- Chapter 8, "More About Linking," for information about import library version checking, segmentation, link maps, the Lib tool, and other useful linking details
- Chapter 9, "Standard Libraries," for information about the libraries included with MPW
- Chapter 10, "Make and Makefiles," which describes how to write makefiles to automate the build process

If you are working on a large project, you may want to read Chapter 16, "Managing Projects With Projector," which discusses the tools and scripts you can use to control your source files.

If you encounter problems when building your programs, you can consult the troubleshooting information in Appendix E.

Building PowerPC Runtime Programs

Contents

This chapter describes how to build programs to run in the PowerPC runtime environment. Step-by-step instructions are provided to build applications and shared libraries, including common compiler and linker options. Each section also includes sample makefiles that you can adapt for your own code.

PowerPC runtime programs store all their executable code in a PEF container in the data fork of a file, while noncode resources (along with the `'cfrg' 0` resource) are stored in the resource fork. PowerPC runtime code can run only on PowerPC-based computers. To make best use of this chapter, you should be familiar with the information in the books *Inside Macintosh: PowerPC System Software* and *Macintosh Runtime Architectures*.

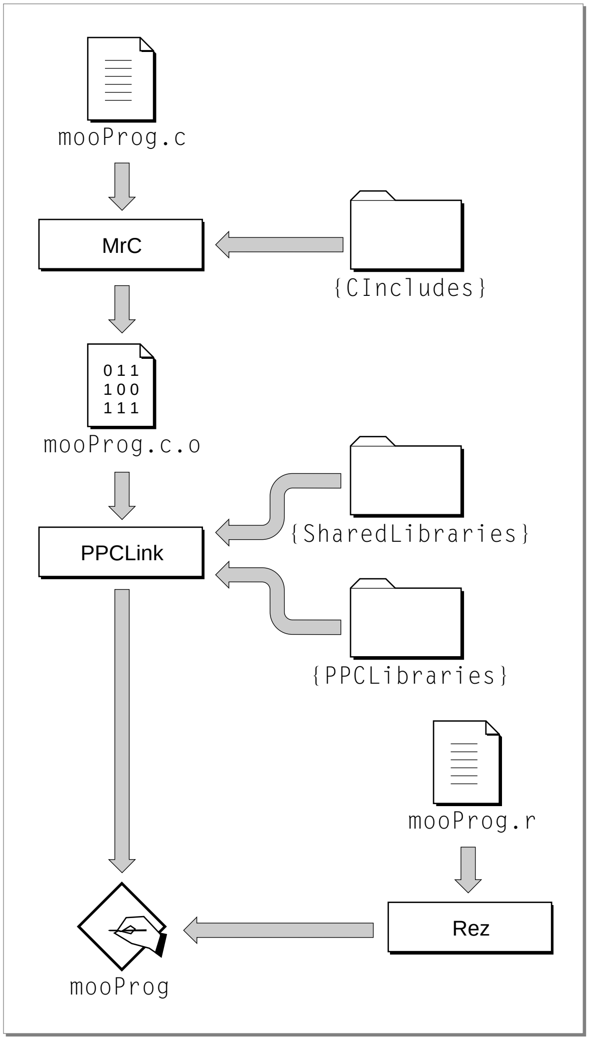
Note
The availability of a native MPW Shell and many native tools means that you can build programs much more quickly and efficiently on a PowerPC-based computer than on a 68K-based computer. However, it is possible to build your programs on a 68K-based machine and then transfer the file to a PowerPC-based computer for execution. In such cases, Apple recommends a 68030-based or 68040-based Macintosh computer for program development. ◆

You can find more detailed information about the MrC and MrCpp compilers in the book *MrC/MrCpp: C/C++ Compiler for Power Macintosh*. For information about other tools described in this chapter, see the *MPW Command Reference*.

Building a PowerPC Runtime Application

Figure 2-1 illustrates the build procedure for a PowerPC runtime application.

Figure 2-1 PowerPC runtime application build procedure



The following steps describe how to build a sample application, `mooProg`.

1. **Compile the source files with the MrC or MrCpp compiler.**

   You use MrC to compile C source code, MrCpp for C++ source code. For example, to compile the source file `mooProg.c` into an object file named `mooProg.c.o`, you can use the following command:

   ```
   MrC mooProg.c -o mooProg.c.o
   ```

2. **Link the application using PPCLink.**

   The PPCLink tool automatically creates a `'cfrg' 0` resource, so you do not have to create one with Rez. The default output is in **Preferred Executable Format (PEF),** which is the executable format used by PowerPC-based Macintosh computers. However, if desired you can also create files in the **Extended Common Object File Format (XCOFF)** by using the `-outputformat xcoff` option.

   To link the object file `mooProg.c.o` to the standard libraries to produce `mooProg`, you can use the following command:

   ```
   PPCLink  ∂
           mooProg.c.o ∂
           "{SharedLibraries}"InterfaceLib ∂
           "{SharedLibraries}"StdCLib ∂
           "{PPCLibraries}"StdCRuntime.o ∂
           "{PPCLibraries}"PPCCRuntime.o ∂
           -fragname mooCowApp ∂
           -c 'MOOF' ∂
           -o mooProg
   ```

   Note that the PowerPC runtime standard libraries include both static libraries and shared libraries. For more information on standard libraries and when to link to them, see “Runtime Libraries,” beginning on page 2-12, and Chapter 9, “Standard Libraries.” You can link any shared libraries you have created to your application by including their names (and pathnames) in the list.

   The `-fragname` option allows you specify a name for the fragment. If you don’t use this option, the fragment will have the same name as the output file (in this case, `mooProg`).

   The `-c` option specifies the creator.

   If you have written initialization or termination routines, you can have them override the default routines by using the `-init` or `-term` option. See “Initialization and Termination Routines (PowerPC and CFM-68K Only),” beginning on page 8-10, for more information.

3. **Use Rez to compile all resources used by the application and add them to the resource fork of the application file.**

   You can use the following command to compile the resource file `mooProg.r` and add it to the application `mooProg`.

   ```
   Rez mooProg.r -append -o mooProg
   ```

   For more detailed information about the Rez tool, see the *MPW Command Reference* and Appendix C in this book.

You can now load the completed application onto a PowerPC-based computer and execute it. To launch the application, simply double-click its icon.

Listing 2-1 shows a sample makefile for building a PowerPC runtime application.

Listing 2-1 Makefile for a PowerPC runtime application

```
# Makefile to build mooProg from    mooProg.c
#                                   mooProg.h
#                                   mooProg.r

# VARIABLE DEFINITIONS

# Define object files that PPCLink will combine
Objects = mooProg.c.o

# Define the standard libraries to link with.
PPCLibs =   "{SharedLibraries}"InterfaceLib ∂
            "{SharedLibraries}"StdCLib ∂
            "{PPCLibraries}"StdCRuntime.o ∂
            "{PPCLibraries}"PPCCRuntime.o

# DEFAULT BUILD RULE

all ƒ mooProg

# COMPILE DEPENDENCIES

mooProg.c.o ƒ mooProg.c mooProg.h
     MrC mooProg.c -o mooProg.c.o
```

```
# TARGET DEPENDENCIES
# Any shared libraries can be included in the link list or in the
# library definition above.
# Note PPCLink options -c to set creator, -fragname to name fragment

mooProg ƒƒ mooProg.make  {Objects} {PPCLibs}
    PPCLink -o mooProg ∂
    {Objects} ∂
    {PPCLibs} ∂
    -fragname mooCowApp ∂
    -c 'MOOF'

mooProg ƒƒ mooProg.make mooProg.r
    Rez mooProg.r -append -o mooProg
```

Building a PowerPC Runtime Shared Library

A shared library fragment exports functions and global variables to other fragments (which can be applications or other shared libraries). Both forms of shared libraries, the import library and the drop-in addition, are built the same way. However, if you are building a drop-in addition, you must then modify the `'cfrg' 0` resource of the shared library after you build it.

The process for building a PowerPC runtime shared library is very similar to that for building a PowerPC runtime application, but some of the options are different.

Building an Import Library

The following steps describe how to build an import library called `mooLib`.

1. **Compile the source files with the MrC or MrCpp compiler.**

   This procedure is identical to that for building a PowerPC runtime application.

   ```
   MrC mooLib.c -o mooLib.c.o
   ```

If you want to create a list of exported symbols for the import library, you should indicate the `-shared_lib_export on` option and specify the export filename with the `-export_list` option.

2. **Link the application using PPCLink.**

   The linking procedure is similar to that for an application, but you must specify the option `-xm s` to designate that you are building a shared library:

```
PPCLink  ∂
        mooLib.c.o ∂
        "{SharedLibraries}"InterfaceLib ∂
        "{SharedLibraries}"StdCLib ∂
        "{PPCLibraries}"StdCRuntime.o ∂
        "{PPCLibraries}"PPCCRuntime.o ∂
        -xm s ∂
        -fragname mooCowLib ∂
        -c 'MOOF'
        -o mooLib
```

   As with an application, the `-fragname` option specifies a name for the fragment and the `-c` option specifies the creator.

   If you want to indicate exported symbols, you can use the `-export` or `-@export` option.

   As with PowerPC runtime applications, if you have written initialization or termination routines, you can have them override the default routines by using the `-init` or `-term` option. See “Initialization and Termination Routines (PowerPC and CFM-68K Only),” beginning on page 8-10, for more information.

   If you are modifying an existing version of a shared library, you should include version number options to ensure compatibility with client programs. See “Import Library Version Checking (PowerPC and CFM-68K Only),” beginning on page 8-12, for details.

3. **Use Rez to compile all resources used by the application and add them to the resource fork of the application file.**

```
Rez mooLib.r -append -o mooLib
```

Listing 2-2 shows a sample makefile for building a PowerPC runtime import library.

Listing 2-2 Makefile for a PowerPC runtime import library

```
# Makefile to build mooLib from     mooLib.c
#                                   mooLib.h
#                                   mooLib.r

# VARIABLE DEFINITIONS

# Define object files that PPCLink will combine
Objects = mooLib.c.o

# Define the standard libraries to link with.
PPCLibs =   "{SharedLibraries}"InterfaceLib ∂
            "{SharedLibraries}"StdCLib ∂
            "{PPCLibraries}"StdCRuntime.o ∂
            "{PPCLibraries}"PPCCRuntime.o

# DEFAULT BUILD RULE

all ƒ mooLib

# COMPILE DEPENDENCIES

mooLib.c.o ƒ mooLib.c mooLib.h
     MrC mooLib.c -o mooLib.c.o

# TARGET DEPENDENCIES
# The shared library being built can be linked in turn to other shared
# libraries by including their names in the link list or in the library
# definition above.
# Note PPCLink option -xm s to build a shared library

mooLib ƒƒ mooLib.make  {Objects} {PPCLibs}
    PPCLink -o mooLib ∂
    {Objects} ∂
    {PPCLibs} ∂
    -xm s ∂
    -fragname mooCowLib ∂
    -c 'MOOF'

mooLib ƒƒ mooLib.make mooLib.r
    Rez mooLib.r -append -o mooLib
```

Building a Drop-In Addition

A drop-in addition is a shared library that the client application must explicitly load into memory. Sometimes called *plug-ins* or *application extensions*, drop-in additions contain code that extends the capabilities of the client application. For example, an application extension could contain data-conversion filters or a spelling checker. Unlike import libraries, drop-in additions are not linked to client applications and as such are not automatically loaded by the Code Fragment Manager. The application must make explicit calls to the Code Fragment Manager to load a drop-in-addition (by using the `GetDiskFragment` or `GetMemFragment` routine) and must then find the symbols associated with the library (by using the `CountSymbols` and `GetIndSymbols` routines, for example).

A drop-in addition is essentially an import library with a modified `'cfrg' 0` resource. To build a drop-in addition, first build your program as an import library. Then you must modify the `'cfrg' 0` resource to prevent the Code Fragment Manager from treating the drop-in addition as an import library.

To designate a drop-in-addition, you must change the `usage` field of the `'cfrg' 0` resource from `kImportLibraryCFrag` to `kDropInAdditionCFrag`. For more information about Code Fragment Manager routines and the structure of the `'cfrg' 0` resource, see the Code Fragment Manager chapter of *Inside Macintosh: PowerPC System Software*.

If desired, you can also create a drop-in addition by deleting the `'cfrg' 0` resource entirely. However, modifying the `usage` field allows more flexibility for future changes.

Merging Fragments

You can package several different program fragments in one file by using the MergeFragment tool. MergeFragment combines fragments in the data fork and then modifies the `'cfrg' 0` resource so the Code Fragment Manager can identify each fragment (other resources are not affected). You can use this method to package several shared libraries in one file. For example, given the two files `Cow`, which contains the fragment `mooLib`, and `Dog`, which contains the fragment `woofLib`, you can add the fragment `woofLib` to the file `Cow` by using the following command:

```
MergeFragment Dog Cow
```

The resulting file `Cow` contains both the fragment `mooLib` and the fragment `woofLib`.

MergeFragment is often used to build fat binary files (see Chapter 5, “Building Fat Binary Files,” for details). For general information about the MergeFragment tool, see the *MPW Command Reference*.

MPW Conventions for the PowerPC Runtime Environment

File-Naming Conventions

Table 2-1 lists the conventions for naming files when building for the PowerPC runtime environment. While not mandatory, following these naming conventions minimizes confusion and maximizes compatibility with other PowerPC build files.

Table 2-1 PowerPC runtime environment file-naming conventions

File type	Filename
C source file for MrC	*filename*`.c`
C++ source file for MrCpp	*filename*`.cp`
Rez source file	*filename*`.r`
Export symbols list from MrC/MrCpp	*filename*`.x`
Object file produced by MrC	*filename*`.c.o` or *filename*`.c.x`
Object file produced by MrCpp	*filename*`.cp.o` or *filename*`.cp.x`
Linker output file	*applicationName*
Shared library	*library*
Static library	*library*`.o`

Runtime Libraries

Table 2-2 lists the standard MPW PowerPC runtime libraries.

Table 2-2 PowerPC runtime libraries

Library for	PowerPC library	When to use
Toolbox and OS interfaces	InterfaceLib	When building any nondriver program.
C	StdCLib	If your fragment uses ANSI C symbols (such as those declared in the header file stdio.h) or integrated environment symbols (such as those declared in the header file fcntl.h).
	StdCRuntime.o	When building a C or C++ program; this library contains the entry point __start.
C++	MrCPlusLib.o	When building a C++ program.
	MrCIOStreams.o	When using C++ I/O streams classes.
MPW tools	PPCToolLibs.o	If your fragment uses a symbol declared in one of the header files CursorCtl.h, ErrMgr.h, Unmangler.h, Disassembler.h, DisAsmLookup.h, or MC68000Test.h.
Numerics	MathLib	If your fragment uses a symbol declared in one of the header files fp.h, fenv.h, or float.h.
Compiler support	PPCCRuntime.o	If you compiled your fragment with the MrC or MrCpp compiler.
SIOW	PPCSIOW.o	When building SIOW applications. This library must replace StdCRuntime.o when linking.

Libraries with a .o extension are static libraries, which are stored in the PPCLibraries folder. Those without an extension are shared libraries, which are stored in the SharedLibraries folder.

MPW Shell Variables for PowerPC

The MPW `Startup` file initializes the following variables for use with PowerPC builds when you launch the MPW Shell:

- `"{PPCLibraries}"`, which indicates the path to the standard PowerPC static libraries (for example, `PPCCRuntime.o` and `StdCRuntime.o`). The default location is `"{MPW}"Libraries:PPCLibraries:`.
- `"{SharedLibraries}"`, which indicates the path to the shared libraries (for example, `InterfaceLib`) used by both PowerPC runtime and CFM-68K runtime programs. The default location is `"{MPW}"Libraries:SharedLibraries:`.

You should use these variables in your makefiles when specifying library names. For example,

```
"{PPCLibraries}"PPCCRuntime.o
```

specifies the library `PPCCRuntime.o` in the folder pointed to by the `"{PPCLibraries}"` variable.

You can override the default shell variable definitions by using the `Set` command. See the *MPW Command Reference* for details.

Building Classic 68K Runtime Programs

Contents

This chapter provides instructions and examples for building programs to run in the classic 68K runtime environment. In addition to the standard compiler and linker options, this chapter also includes sample makefiles that you can adapt for use with your own code.

The classic 68K runtime architecture stores segmented code in the resource fork as `'CODE'` resources, and it does not rely on the data fork. Classic 68K code can run on 68K-based computers or under emulation on PowerPC-based computers. You can find details of the classic 68K runtime architecture in *Macintosh Runtime Architectures* as well as in many books of the *Inside Macintosh* series.

IMPORTANT
Future versions of the Mac OS will not support the older Apple Shared Library Manager (ASLM). If you want to run shared libraries on a 68K-based computer, you must use the CFM-68K runtime architecture. See Chapter 4, "Building CFM-68K Runtime Programs," for more information. ▲

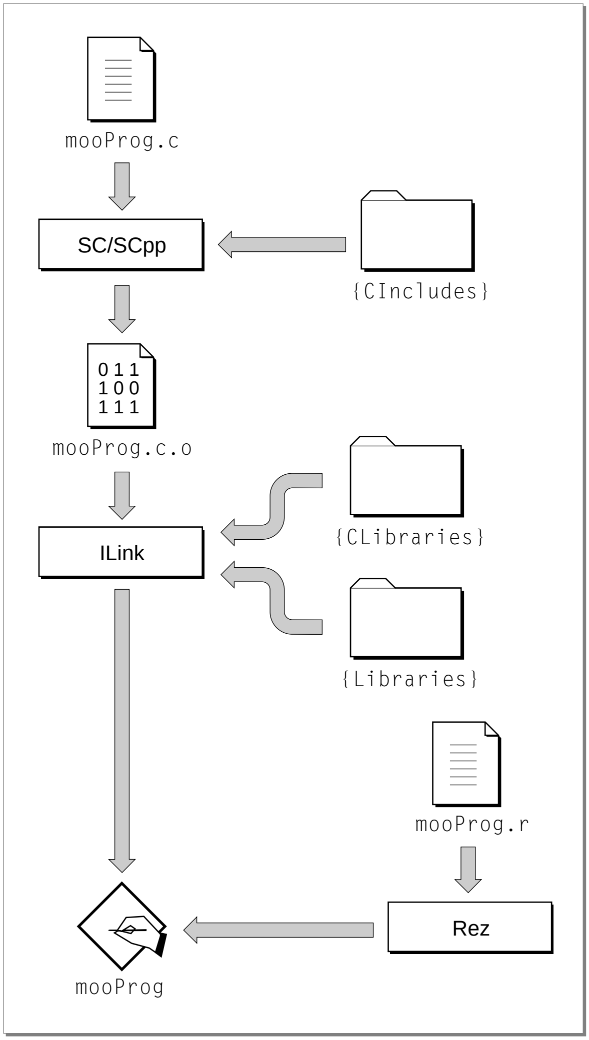
Note
You can build and execute classic 68K programs on both 68K-based and PowerPC-based computers. ◆

The book *SC/SCpp: C/C++ Compiler for 68K Macintosh* provides more information about the SC and SCpp compilers used in this chapter. For more details about other tools used in this chapter, see the *MPW Command Reference*.

Building a Classic 68K Runtime Application

Figure 3-1 illustrates the build procedure for a classic 68K application.

Figure 3-1 Classic 68K runtime application build procedure



The following steps describe how to build a sample application `mooProg`.

1. **Compile the source files with the SC or SCpp compiler.**

   For example, to compile the source file `mooProg.c` into an object file named `mooProg.c.o`, you can use the following command:

   ```
   SC mooProg.c -o mooProg.c.o
   ```

   If you are building a program designed to execute on a machine running an MC68881/68882 math coprocessor, you should compile using the `-MC68881` option.

2. **Link the application using ILink.**

   To link the object file `mooProg.c.o` to the standard libraries to produce the executable `mooProg`, you can use the following command:

   ```
   ILink ∂
       mooProg.c.o ∂
       "{Libraries}"MacRuntime.o ∂
       "{Libraries}"Interface.o ∂
       "{Libraries}"IntEnv.o ∂
       "{CLibraries}"StdCLib.o ∂
       -c 'MOOF' ∂
       -o mooProg
   ```

   If your program uses the MC68881/68882 math coprocessor, you need to substitute some standard libraries with versions built with the `-MC68881` option. See "Runtime Libraries," beginning on page 3-8, for information about libraries that support the `-MC68881` option and how to use them in your link lists.

   The ILink option `-c` specifies the creator.

   For your final link, you should add the `-pad 0` and `-compact` options. This removes any segment padding and compacts the resource fork. However, these optimizations slow down the linker, so you should specify them only for your final link.

3. **Use Rez to compile resources used by the application and add them to the resource fork of the application file.**

   You can use the following command to compile `mooProg.r` and add it to the application `mooProg`:

   ```
   Rez -o mooProg mooProg.r -append
   ```

   For a detailed description of the Rez tool, see Appendix C in this book or the *MPW Command Reference*.

You can now load the completed application onto a 68K-based or PowerPC-based computer and execute it. To launch the application, simply double-click its icon.

Listing 3-1 shows a sample makefile for building a classic 68K runtime application.

Listing 3-1 A makefile for a classic 68K application

```
# Makefile to build mooProg from         mooProg.c
#                                        mooProg.h
#                                        mooProg.r

# VARIABLE DEFINITIONS

# Define object files that ILink will combine.
Objects = mooProg.c.o

# Define standard libraries to link with.
68KLibs =       "{Libraries}"MacRuntime.o ∂
                "{Libraries}"Interface.o ∂
                "{Libraries}"IntEnv.o ∂
                "{CLibraries}"StdCLib.o


# DEFAULT BUILD RULE

all ƒ mooProg

# COMPILE DEPENDENCIES

mooProg.c.o ƒ mooProg.h mooProg.c
     SC mooProg.c -o mooProg.c.o

# TARGET DEPENDENCIES
# Note ILink -c option to set creator.
```

```
mooProg ƒƒ {Objects} {68KLibs} mooProg.make
    ILink -o mooProg ∂
    {Objects} ∂
    {68KLibs} ∂
    -c 'MOOF'

mooProg ƒƒ mooProg.r mooProg.make
    Rez -o mooProg mooProg.r -append
```

MPW Conventions for the Classic 68K Runtime Environment

File-Naming Conventions

Table 3-1 shows the suggested naming conventions for files when building programs for the classic 68K runtime environment.

Table 3-1 Classic 68K runtime environment file-naming conventions

File type	Filename
C source file for SC	*filename*`.c`
C++ source file for SCpp	*filename*`.cp`
Rez source file	*filename*`.r`
Object file produced by SC	*filename*`.c.o`
Object file produced by SCpp	*filename*`.cp.o`
Linker output file	*applicationName*
Static library produced by Lib	*library*`.o`

Runtime Libraries

MPW libraries for the classic 68K runtime environment are stored in several different folders. Table 3-2 lists the libraries stored in the Libraries folder.

Table 3-2 Classic 68K runtime libraries stored in Libraries

Library for	Classic 68K library	When to use
Toolbox and OS interfaces	`Interface.o`	Always.
MPW tools	`ToolLibs.o`	If your fragment uses a symbol declared in one of the header files `CursorCtl.h`, `ErrMgr.h`, `Unmangler.h`, `Disassembler.h`, `DisAsmLookup.h`, or `MC68000Test.h`.
	`Stubs.o`	When building an MPW tool. This library contains dummy routines to override standard routines that are not used by MPW tools.
Numerics	`MathLib.o`	If your fragment uses a symbol declared in one of the header files `fp.h`, `fenv.h`, or `float.h`.
	`MathLib881.o`	Substitute for `MathLib.o` if your program requires a machine containing an MC68881/68882 math coprocessor. This library must precede `MacRuntime.o` and `StdCLib.o` in your link list.

continued

Table 3-2 Classic 68K runtime libraries stored in Libraries (continued)

Library for	Classic 68K library	When to use
Classic 68K runtime	`MacRuntime.o`	When building an application.
	`IntEnv.o`	If your program uses integrated environment symbols (such as those declared in the header file `fcntl.h`).
	`RTLib.o`	If you need to control or access Segment Loader routines in a `-model far` environment. See *Macintosh Runtime Architectures* for more information.
SIOW	`SIOW.o`	When building an SIOW application. The library `SIOW.o` must precede `MacRuntime.o` in your link list.
Drivers	`DRVRRuntime.o`	When building a 68K device driver.
Perf	`PerformLib.o`	If you want to use MPW Perf to test the performance of your program.
Proff	`Proff.o`	If you want to use MPW Proff to test the performance of your program.

Table 3-3 lists the libraries stored in the CLibraries folder.

Table 3-3 Classic 68K runtime libraries in CLibraries

Library for	Classic 68K library	When to use
C	`StdClib.o`	If your program uses ANSI C symbols (such as those declared in the header file `stdio.h`).
	`CLib881.o`	If your C program requires a machine containing an MC68881/68882 math coprocessor. This library must precede all members of the `{CLibraries}` folder as well as `MacRuntime.o` in your link command line.
C++	`CPlusLib.o`	If you are using C++.
	`IOStreams.o`	If you are using C++ I/O streams classes.
	`IOStreams881.o`	Substitute for `IOStreams.o` if your C++ program requires a machine that has an MC68881/68882 math coprocessor.

In addition, if you need to link Pascal object files into your program, you must link with the library `PasLib.o` (and `ObjLib.o` for Object Pascal) found in the folder PLibraries.

Several of the libraries above (those ending with `881.o`) should be used when compiling for a machine that uses the MC68881 or MC68882 math coprocessor (that is, when compiling with the `-mc68881` option). You should keep the following in mind when building programs that require a math coprocessor:

- If you use any ANSI C functions, you must link with both `StdCLib.o` and `CLib881.o`.
- If you use C++ I/O streams classes, you must replace `IOStreams.o` with `IOStreams881.o`.

- `MathLib881.o` and `CLib881.o` must precede both `MacRuntime.o` and `StdClib.o` in your link command line. Note that you may get duplicate symbol definition warnings since the complex number libraries replace routines in `MacRuntime.o` and `StdCLib.o` with versions compiled with the `-mc68881` switch.
- You cannot use non-881 numerics functions (including conversion and file-access functions like `printf()`) from code compiled with the `-mc68881` switch. Similarly, you cannot call the 881 libraries from non-881 code.

MPW Shell Variables for Classic 68K

The MPW `Startup` file initializes two variables for the classic 68K runtime when you launch the MPW Shell.

- `"{Libraries}"`, which indicates the path to the standard classic 68K libraries (for example, `MacRuntime.o` and `ToolLibs.o`). The default location is `"{MPW}"Libraries:Libraries:`.
- `"{CLibraries}"`, which indicates the path to the classic 68K C libraries. The default location is `"{MPW}"Libraries:CLibraries:`.

You should use these variables in your makefiles when specifying library names. For example,

```
"{Libraries}"MacRuntime.o
```

specifies the library `MacRuntime.o` in the folder pointed to by the `"{Libraries}"` variable.

You can override the default shell variable definitions by using the `Set` command. See the *MPW Command Reference* for details.

Building CFM-68K Runtime Programs

Contents

This chapter gives step-by-step instructions for building programs to run in the CFM-68K runtime environment. These programs include both applications and shared libraries, with attention given to some of the compiler and linker options. Each section includes sample makefiles that you can adapt for your own code.

The CFM-68K runtime architecture is a hybrid of classic 68K and PowerPC runtime architectures that enables 68K Macintosh computers to use shared libraries. Shared libraries are stored in the data fork, as on PowerPC computers, but applications are still stored as `'CODE'` resources in the resource fork. To make best use of this chapter, you should be familiar with the CFM-68K runtime architecture, which is detailed in *Macintosh Runtime Architectures*.

IMPORTANT

CFM-68K runtime programs cannot run on PowerPC-based computers. ▲

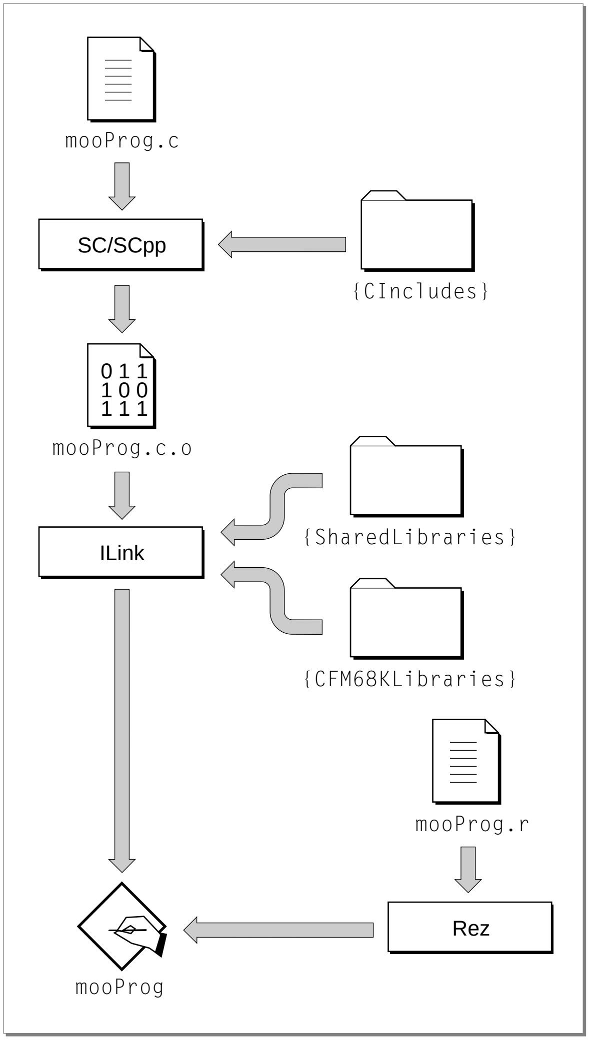
For more detailed information about the SC and SCpp compilers, see the book *SC/SCpp: C/C++ Compiler for 68K Macintosh*. For information about other tools used in this chapter, see the *MPW Command Reference*.

Building a CFM-68K Runtime Application

A CFM-68K runtime application is a stand-alone executable file. Much like its classic 68K counterpart, all the program information is stored in the resource fork. The executable code is segmented and stored as `'CODE'` resources. However, the CFM-68K runtime architecture allows the application to access data items and routines in shared library fragments, which are stored in the data fork.

Figure 4-1 illustrates the basic build procedure for a CFM-68K runtime application called `mooProg`.

Figure 4-1 Building a CFM-68K application



The following steps describe how to build a sample application, `mooProg`.

1. **Compile the source files with the SC or SCpp compiler.**

   You must use the option `-model cfmseg` to generate segmented code compatible with the CFM-68K runtime architecture (you must be running SC or SCpp version 8.0.1 or later). For example, to compile the source file `mooProg.c` into an object file named `mooProg.c.o`, you can use the following command:

   ```
   SC -model cfmseg mooProg.c -o mooProg.c.o
   ```

2. **Link the application using ILink.**

   As with the SC/SCpp compilers, you must use the `-model cfmseg` option to indicate the CFM-68K runtime architecture (you must use ILink version 2.0 or later). To link the object file `mooProg.c.o` to the standard libraries to produce `mooProg`, you can use the following command:

   ```
   ILink ∂
       mooProg.c.o ∂
       "{SharedLibraries}"InterfaceLib ∂
       "{SharedLibraries}"StdCLib ∂
       "{CFM68KLibraries}"NuMacRuntime.o ∂
       -model cfmseg ∂
       -fragname mooCowApp ∂
       -c 'MOOF' ∂
       -o mooProg
   ```

   Any shared libraries that you had previously created could be linked to `mooProg` in the command above by including their names (and pathnames) in the library list. However, the libraries must have been flattened using MakeFlat prior to linking.

   The `-fragname` option allows you specify a name for the fragment. If you don't use this option, the fragment will have the same name as the output file (in this case, `mooProg`).

   The ILink option `-c` specifies the creator.

   If you need to execute nonstandard initialization or termination routines, you must specify the routine names using the `-init` or `-term` option when you link. See "Initialization and Termination Routines (PowerPC and CFM-68K Only)," beginning on page 8-10, for more information.

   For your final link, you should add the `-pad 0` and `-compact` options. This removes any segment padding and compacts the resource fork. However, these optimizations slow down the linker, so you should specify them only for your final link.

3. **Use Rez to compile all resources used by the application and add them to the resource fork of the application file.**

   Note that, like a PowerPC application, all CFM-68K runtime applications must contain a `'cfrg'` resource to distinguish them from standard 68K code. However, ILink automatically creates a `'cfrg'` resource when you specify the `-model cfmseg` option, so you do not need to add one using Rez.

   To compile the resource file `mooProg.r` and add it to the application `mooProg`, use the following command:

   ```
   Rez -o mooProg mooProg.r -append
   ```

   For a detailed description of the Rez tool, see Appendix C in this book or the *MPW Command Reference*.

You can now load the completed application onto a 68K computer and execute it. To launch the application, simply double-click its icon.

Listing 4-1 shows a sample makefile for building a CFM-68K runtime application.

Listing 4-1 A makefile for an application

```
# Makefile to build mooProg from        mooProg.c
#                                       mooProg.h
#                                       mooProg.r

# VARIABLE DEFINITIONS

# Define object files that ILink will combine.
Objects = mooProg.c.o

# Define standard libraries to link with.
cfmLibs =       "{SharedLibraries}"InterfaceLib ∂
                "{SharedLibraries}"StdCLib ∂
                "{CFM68KLibraries}"NuMacRuntime.o

# DEFAULT BUILD RULE
```

```
all ƒ mooProg

# COMPILE DEPENDENCIES

mooProg.c.o ƒ mooProg.h mooProg.c
     SC mooProg.c -model cfmseg -o mooProg.c.o

# TARGET DEPENDENCIES
# Any created shared libraries can be included in the link list or in
# the library definition above.
# Note ILink -c option to set creator.

mooProg ƒƒ {Objects} {cfmLibs} mooProg.make
    ILink -o mooProg ∂
    {Objects} ∂
    {cfmLibs} ∂
    -fragname mooCowApp ∂
    -model cfmseg ∂
    -c 'MOOF'

mooProg ƒƒ mooProg.r mooProg.make
    Rez -o mooProg mooProg.r -append
```

The CFM-68K Runtime Enabler

In order to run CFM-68K runtime applications, the 68K-based computer must have the INIT extension CFM-68K Runtime Enabler installed. If the file is missing, attempting to launch the application displays the message

```
This application requires installation of the "CFM-68K Runtime Enabler."
```

If you want to create a custom version of this message, you must install an `'STR '` resource with ID –20029 in your CFM-68K runtime application.

Building a CFM-68K Runtime Shared Library

CFM-68K runtime shared libraries are compiled and linked much like CFM-68K applications, but libraries must also be "flattened" using the MakeFlat tool. This procedure converts the segmented library into a single fragment, which is then stored in the data fork. After flattening, the CFM-68K runtime library structure looks very similar to that of a shared library on the PowerPC platform.

Building an Import Library

The following steps show how to build an import library called `mooLib`.

1. **Compile the shared library sources using SC or SCpp.**

   ```
   SC -model cfmflat mooLib.c -o mooLib.c.o
   ```

   This is identical to the procedure for compiling an application, except that `-model cfmflat` is used in place of `-model cfmseg`. You can specify the `-model cfmseg` option if desired, but `-model cfmflat` is preferred when building shared libraries.

2. **Link the object files.**

   ```
   ILink ∂
       mooLib.c.o ∂
       "{SharedLibraries}"InterfaceLib ∂
       "{SharedLibraries}"StdCLib ∂
       -model cfmflat ∂
       -xm s ∂
       -fragname mooCowLib ∂
       -c 'MOOF' ∂
       -o mooLib.seg
   ```

   Note that since you are not building an application, you do not have to link with `NuMacRuntime.o`.

   The `-xm s` option specifies that you are building a shared library.

   The `-fragname` option allows you to specify a name for the fragment. If you don't use this option, the fragment will have the same name as the output file (in this case, `mooLib.seg`).

The `-c` option specifies the creator.

If you need to execute nonstandard initialization or termination routines, you must specify the routine names using the `-init` or `-term` option when you link.

If you are modifying an existing version of a shared library, you should include version number options to ensure compatibility with client programs. See "Import Library Version Checking (PowerPC and CFM-68K Only)," beginning on page 8-12, for details.

If this is your final link, you should add the `-pad 0` and `-compact` options. This sets the segment padding to 0 and compacts the resource fork. Since these options slow down the linker, you should use them only for your final link.

3. **Flatten the library using the MakeFlat tool.**

   ```
   MakeFlat mooLib.seg -o mooLib
   ```

   MakeFlat converts the library into a PEF fragment, which is then stored in the data fork. After flattening, you can link the library to a client application.

4. **Append resources to the shared library using Rez.**

   ```
   Rez mooLib.r -append -o mooLib
   ```

Listing 4-2 shows a makefile for building an import library.

Listing 4-2 A makefile for an import library

```
# Makefile to build mooLib from          mooLib.c
#                                        mooLib.h
#                                        mooLib.r

# VARIABLE DEFINITIONS

# Define object files that ILink will combine.
Objects = mooLib.c.o

# Define standard libraries to link with.
# Note no NuMacRuntime.o needed for shared libraries.
cfmLibs=        "{SharedLibraries}"InterfaceLib ∂
                "{SharedLibraries}"StdCLib
```

```
# DEFAULT BUILD RULE

all ƒ mooLib

# COMPILE DEPENDENCIES

mooLib.c.o ƒ mooLib.h mooLib.c
     SC mooLib.c -model cfmflat -o mooLib.c.o

# Target dependencies
# Any created shared libraries can be included in the link list or in
# the library definition above.
# Note ILink options -xm s to specify a shared library, -c option to set
# creator.

mooLib ƒƒ {Objects} {cfmLibs} mooLib.make
    ILink -o mooLib.seg ∂
    {Objects} ∂
    {cfmLibs} ∂
    -model cfmflat ∂
    -fragname mooCowLib ∂
    -xm s ∂
    -c 'MOOF'
    MakeFlat mooLib.seg -o mooLib

mooLib ƒƒ mooLib.r mooLib.make
    Rez mooLib.r -append -o mooLib
```

Building a Library Using -model cfmseg

An alternative to using the `-model cfmflat` option when building a shared library is to use the `-model cfmseg` option. This option creates slightly smaller and faster code but imposes the near addressing restrictions resulting from segmentation. Note that you must still run the MakeFlat tool on your code to create a flattened library.

Note
If you compile your source code using the `-model cfmflat` compiler option, you must use the `-model cfmflat` linker option. However, files compiled using the `-model cfmseg` option can be linked with the `-model cfmflat` option. ◆

Building an Application as a Shared Library

If you wish to create a nonsegmented application, you can do so by first building the application as a shared library (which ends up in the data fork). Then create a minimal application (resource fork–based) that merely calls the main routine of the shared library. The two files can then be combined into one application using the MergeFragment tool.

Building a Drop-In Addition

As in the PowerPC runtime environment, a drop-in addition is essentially an import library with a modified `'cfrg' 0` resource. To build a drop-in addition, first build your program as an import library. After flattening your library, you must modify the `'cfrg' 0` resource to prevent the Code Fragment Manager from treating the drop-in addition as an import library.

The procedure for creating a CFM-68K runtime drop-in addition is identical to that for the PowerPC runtime: change the `usage` field of the `'cfrg' 0` resource from `kImportLibraryCFrag` to `kDropInAdditionCFrag`, or delete the `'cfrg' 0` resource entirely.

See *Inside Macintosh: PowerPC System Software* or *Macintosh Runtime Architectures* for more information about the `'cfrg' 0` resource.

Merging Fragments

Just as in the PowerPC runtime environment, you can use MergeFragment to combine CFM-68K runtime fragments in the data fork. You can use this tool to package shared libraries together or to build fat binary files. See “Merging Fragments” on page 2-10 and Chapter 5, “Building Fat Binary Files,” for more information.

MPW Conventions for the CFM-68K Runtime Environment

File-Naming Conventions

Because CFM-68K runtime programs incorporate elements of both classic 68K and PowerPC runtime architecture, the naming conventions are similarly mixed. Table 4-1 shows the file-naming conventions for the CFM-68K runtime environment.

Table 4-1 CFM-68K runtime environment file-naming conventions

File type	Filename
C source file for SC	*filename*`.c`
C++ source file for SCpp	*filename*`.cp`
Rez source file	*filename*`.r`
Object file produced by SC	*filename*`.c.o`
Object file produced by SCpp	*filename*`.cp.o`
Linker output file for application	*applicationName*
Linker output file for shared library	*library*`.seg`
MakeFlat output file for shared library	*library*
Static library produced by Lib	*library*`.o`

Runtime Libraries

Table 4-2 lists the CFM-68K runtime libraries.

Table 4-2 CFM-68K runtime libraries

Library for	CFM-68K library	When to use
Toolbox and OS interfaces	`InterfaceLib`	Always.
C	`StdClib`	If your fragment uses ANSI C symbols (such as those declared in the header file `stdio.h`) or integrated environment symbols (such as those declared in the header file `fcntl.h`).
C++	`NuCPlusLib.o`	If you are using C++.
	`NuIOStreams.o`	If you are using C++ I/O streams classes.
“Tool” routines	`NuToolLibs.o`	If your fragment uses a symbol declared in one of the header files `CursorCtl.h`, `ErrMgr.h`, `Unmangler.h`, `Disassembler.h`, `DisAsmLookup.h`, or `MC68000Test.h`.
Numerics	`NuMathLib.o`	If your fragment uses a symbol declared in one of the header files `fp.h`, `fenv.h`, or `float.h`.
CFM-68K runtime	`NuMacRuntime.o`	When building an application.
	`NuRTLib.o`	If you need to patch the Segment Loader routines.
SIOW	`NuSIOW.o`	When building an SIOW application. `NuSIOW.o` must appear in the ILink command line prior to `NuMacRuntime.o`.

Libraries ending with the `.o` extension are static libraries, which means that the linker includes their code in the final linked file. Static libraries are stored in the folder CFM68KLibraries. `InterfaceLib` and `StdClib` are shared libraries, which are stored in the folder SharedLibraries.

IMPORTANT

Classic 68K runtime libraries cannot be linked to a CFM-68K runtime program. All static CFM-68K runtime libraries have the prefix `Nu` to distinguish them from their classic 68K counterparts. Shared libraries have no `.o` extension. ▲

Note

Although `NuToolLibs.o` is included among the CFM-68K libraries, you cannot build a CFM-68K runtime MPW tool. However, you can use any of the `NuToolsLibs.o` routines in a CFM-68K runtime application. ◆

MPW Shell Variables for CFM-68K

The MPW `Startup` file initializes two CFM-68K-related variables when you launch the MPW Shell.

- `"{CFM68KLibraries}"`, which indicates the path to the standard CFM-68K static libraries (for example, `NuMacRuntime.o` and `NuToolLibs.o`). The default location is `"{MPW}"Libraries:CFM68KLibraries:`.
- `"{SharedLibraries}"`, which indicates the path to the shared libraries (for example, `InterfaceLib`) used by both CFM-68K runtime and PowerPC runtime programs. The default location is `"{MPW}"Libraries:SharedLibraries:`.

You should use these variables in your makefiles when specifying library names. For example,

```
"{CFM68KLibraries}"NuMacRuntime.o
```

specifies the library `NuMacRuntime.o` in the folder pointed to by the `"{CFM68KLibraries}"` variable.

You can override the default shell variable definitions by using the `Set` command. See the *MPW Command Reference* for details.

Building Fat Binary Files

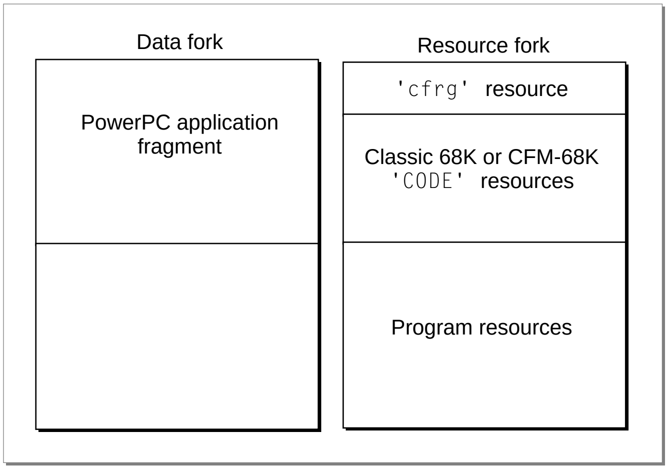
Contents

Fat binary files contain executable code for multiple runtime architectures. For example, a fat application can contain both 68K runtime and PowerPC runtime code, and as such can function on both types of Macintosh computers.

Fat applications combine either classic 68K runtime or CFM-68K runtime code with PowerPC runtime code. You can also build **fat shared libraries,** which combine CFM-68K runtime and PowerPC runtime code.

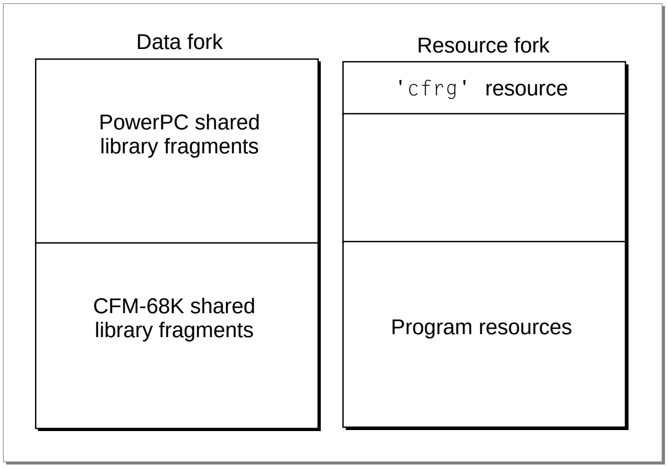
Fat binary applications are made possible by the way code is stored in a Macintosh file. PowerPC runtime applications and shared libraries store executable code in the data fork of a file, while classic 68K and CFM-68K runtime applications store their executable code in `'CODE'` resources in the resource fork. Therefore you can create a file that contains both types of executable code. Any noncode resources (`'wdef'`, `'vers'`, and so on) can be accessed from either type of runtime code. Figure 5-1 shows the structure of a fat binary application.

Figure 5-1 Structure of a fat application



The structure of a fat shared library is slightly different from that of a fat application. Shared library code is always stored as fragments in the data fork (whether for PowerPC runtime or CFM-68K runtime), so only noncode resources appear in the resource fork. However, since information about both types of code fragments is contained in the `'cfrg' 0` resource, the Process Manager can use the resource to determine which fragment to execute. Figure 5-2 shows the structure of a fat shared library.

Figure 5-2 Structure of a fat shared library



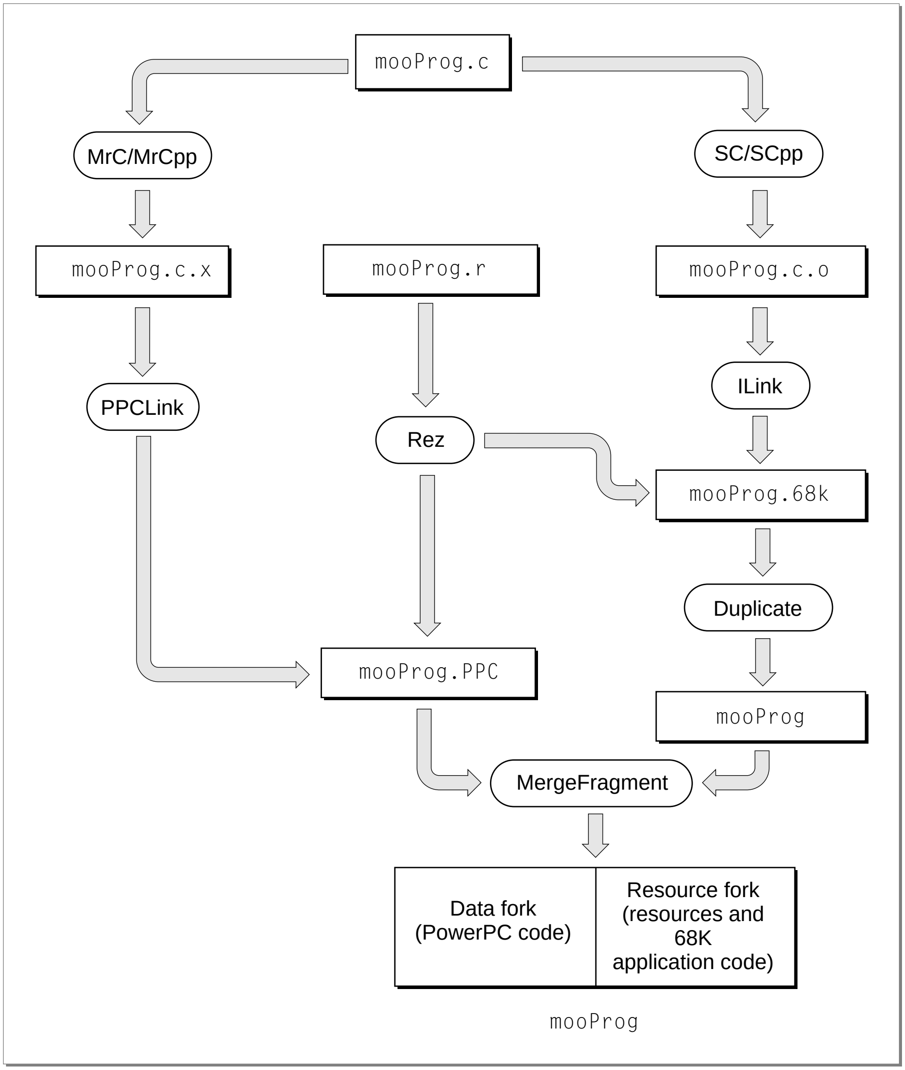
To create a fat binary version of a file, you use the MergeFragment tool to combine the 68K-based code with the PowerPC-based code. MergeFragment combines data fork information, leaving the resource fork untouched (except for the `'cfrg' 0` resource). If there are multiple fragments in the data fork, the tool reads information from each fragment's `'cfrg' 0` resource and creates a new `'cfrg' 0` resource that contains information about each fragment.

For detailed information about the MergeFragment tool, see the *MPW Command Reference*.

Building a Fat Application

The most straightforward way to build a fat program is to build the 68K and PowerPC versions separately and then merge them using the MergeFragment tool. Figure 5-3 illustrates the build procedure for a fat application.

Figure 5-3 Fat application build procedure



The following steps describe how to build a fat application:

1. **Compile and link the PowerPC code.**

```
MrC mooProg.c -o mooProg.c.x
PPCLink  ∂
        mooProg.c.x ∂
        "{SharedLibraries}"InterfaceLib ∂
        "{SharedLibraries}"StdCLib ∂
        "{PPCLibraries}"StdCRuntime.o ∂
        "{PPCLibraries}"PPCCRuntime.o ∂
        -fragname mooCowPPC ∂
        -o mooProg.PPC
Rez mooProg.r -append -o mooProg.PPC
```

2. **Compile and link the 68K code.**

```
SC mooProg.c -o mooProg.c.o
ILink  ∂
        mooProg.c.o ∂
        "{CLibraries}"StdCLib.o ∂
        "{Libraries}"IntEnv.o ∂
        "{Libraries}"MacRuntime.o ∂
        "{Libraries}"Interface.o ∂
        -o mooProg.68k
Rez mooProg.r -append -o mooProg.68k
```

3. **Combine the two files using MergeFragment.**

```
Duplicate -y mooProg.68k mooProg
MergeFragment mooProg.PPC mooProg
```

Note

The Rez step in creating the PowerPC runtime code above is actually superfluous when creating a fat program since MergeFragment does not copy any resources from `mooProg.PPC` except the `'cfrg' 0` resource. However, this procedure has the advantage of creating two separately executable programs in addition to the fat binary combination. ◆

Listing 5-1 shows a makefile that can build a classic 68K runtime application, a PowerPC runtime application, or a fat application.

If you specify `all` in the `Make` command line, then the file builds all three versions:

```
Make -f mooProg.make all
```

You can also build only one runtime version by specifying one of the following:

```
Make -f mooProg.make mooProg.68k
Make -f mooProg.make mooProg.PPC
```

You may also want to examine “Example 4—Multiple Folders and Multiple Makefiles,” beginning on page 10-48, for another example of a fat binary makefile.

Listing 5-1 A makefile for a fat application

```
# Makefile for a fat binary application containing both classic 68K
# and PowerPC code.

# This file first builds the 68K and PowerPC versions individually and
# then combines the two using the MergeFragment tool.

# VARIABLE DEFINITIONS

# Application information
appName =       mooProg
Creator =       'MOOF'


# Define the standard libraries to link with.
PPCLibs =   "{SharedLibraries}"InterfaceLib ∂
            "{SharedLibraries}"StdCLib ∂
            "{PPCLibraries}"StdCRuntime.o ∂
            "{PPCLibraries}"PPCCRuntime.o

68KLibs =   "{CLibraries}"StdCLib.o ∂
            "{Libraries}"IntEnv.o ∂
            "{Libraries}"MacRuntime.o ∂
            "{Libraries}"Interface.o
```

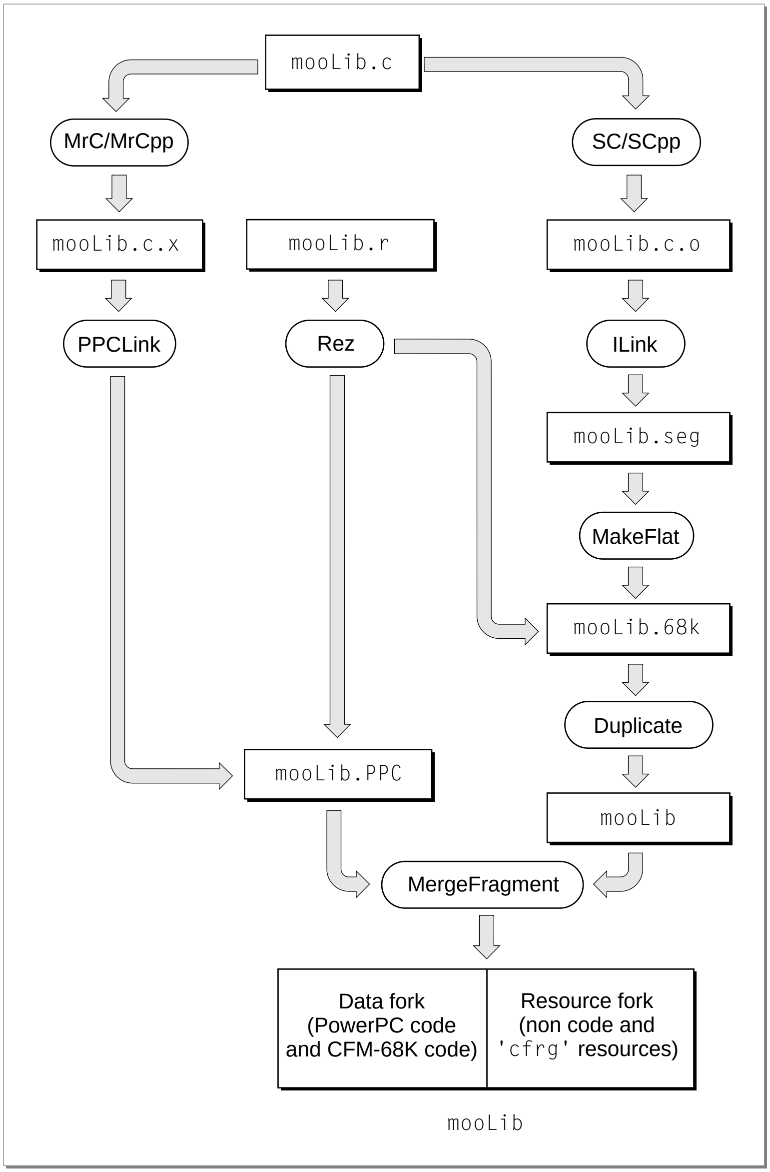
```
# Define link objects
68KObjects = mooProg.c.o
PPCObjects = mooProg.c.x


# DEPENDENCIES

# Compile dependencies
.c.x ƒ .c
     MrC {default}.c -o {default}.c.x

.c.o ƒ .c
     SC {default}.c -o {default}.c.o


# Target dependencies

all ƒ {appName}.PPC ∂
        {appName}.68k ∂
        {appName}


# Build the PowerPC version.
{appName}.PPC  ƒƒ {appName}.make  {PPCObjects} {PPCLibs}
    PPCLink  ∂
        {PPCObjects} ∂
        {PPCLibs} ∂
        -fragname mooCowPPC ∂
        -c {Creator} ∂
        -o {appName}.PPC

{appName}.PPC  ƒƒ {appName}.make {appName}.r
    Rez {appName}.r -append -o {appName}.PPC
```

```
# Build the classic 68K version.
{appName}.68k ƒƒ {appName}.make  {68KObjects} {68KLibs}
    ILink  ∂
        {68KObjects} ∂
        {68KLibs}∂
        -c {Creator}
        -o {appName}.68k

{appName}.68k ƒƒ {appName}.make {appName}.r
    Rez {appName}.r -append -o {appName}.68k


# Combine the two applications into a fat binary.
{appName} ƒƒ {appName}.PPC {appName}.68k
    Duplicate -y {appName}.68k {appName}
    Mergefragment {appName}.PPC {appName}
```

Building a Fat Shared Library

The build process for a fat shared library is very similar to that for building a fat application. However, only the CFM-68K/PowerPC combination is possible since the classic 68K runtime architecture does not support shared libraries. Figure 5-4 shows the build procedure for a fat shared library.

Figure 5-4 Fat shared library build process



The steps to build a fat shared library are as follows:

1. **Compile and link the PowerPC code.**

```
MrC mooLib.c -o mooLib.c.x
PPCLink  ∂
        mooLib.c.x ∂
        "{SharedLibraries}"InterfaceLib ∂
        "{SharedLibraries}"StdCLib ∂
        "{PPCLibraries}"StdCRuntime.o ∂
        "{PPCLibraries}"PPCCRuntime.o ∂
        -xm s ∂
        -fragname mooCowPPC ∂
        -o mooLib.PPC
Rez mooLib.r -append -o mooLib.PPC
```

2. **Compile and link the CFM-68K code.**

```
SC  -model cfmflat mooLib.c -o mooLib.c.o
ILink  ∂
        mooLib.c.o ∂
        "{SharedLibraries}"InterfaceLib ∂
        "{SharedLibraries}"StdCLib ∂
        -model cfmflat ∂
        -xm s ∂
        -fragname mooCow68k ∂
        -o mooLib.seg
MakeFlat mooLib.seg -o mooLib.68k
Rez mooLib.r -append -o mooLib.68k
```

3. **Combine the two files using MergeFragment.**

```
Duplicate -y mooLib.68k mooLib
MergeFragment mooLib.PPC mooLib
```

Listing 5-2 shows a sample makefile for building a fat shared library. You can create all three versions of the shared library by using the command

```
Make -f mooLib.make all
```

You can also build a single runtime version by specifying one of the following:

```
Make -f mooLib.make mooLib.68k
Make -f mooLib.make mooLib.PPC
```

Listing 5-2 A makefile for a fat shared library

```
# Makefile for a fat binary shared library containing both CFM-68K and
# PowerPC code.

# This file first builds CFM-68K and PowerPC versions individually and
# then combines the two using the MergeFragment tool.

# VARIABLE DEFINITIONS

# Shared Library Information.
libName =       mooLib
Creator =       'MOOF'


# Define the standard libraries to link with.
# Note that both the CFM-68K and PowerPC versions link with the fat
# shared libraries InterfaceLib and StdCLib.
PPCLibs =   "{SharedLibraries}"InterfaceLib ∂
            "{SharedLibraries}"StdCLib ∂
            "{PPCLibraries}"StdCRuntime.o ∂
            "{PPCLibraries}"PPCCRuntime.o

CFMLibs =   "{SharedLibraries}"InterfaceLib ∂
            "{SharedLibraries}"StdCLib


# Define link objects.
CFMObjects = mooLib.c.o
PPCObjects = mooLib.c.x


# DEPENDENCIES

# Compile dependencies.
.c.x ƒ .c
     MrC {default}.c -o {default}.c.x

.c.o ƒ .c
     SC  -model cfmflat {default}.c -o {default}.c.o
```

```
# Target dependencies.

all ƒ {libName}.PPC ∂
        {libName}.68k ∂
        {libName}


# Build the PowerPC version.
{libName}.PPC  ƒƒ {libName}.make  {PPCObjects} {PPCLibs}
    PPCLink  ∂
        {PPCObjects} ∂
        {PPCLibs} ∂
        -xm s ∂
        -fragname mooCowPPC ∂
        -c {Creator} ∂
        -o {libName}.PPC

{libName}.PPC  ƒƒ {libName}.make {libName}.r
    Rez {libName}.r -append -o {libName}.PPC


# Build the CFM-68K version.
{libName}.68k ƒƒ {libName}.make  {CFMObjects} {CFMLibs}
    ILink  ∂
        {CFMObjects} ∂
        {CFMLibs}∂
        -model cfmflat ∂
        -xm s ∂
        -fragname mooCow68k ∂
        -c {Creator} ∂
        -o {libName}.seg

MakeFlat {libname}.seg -o {libname}.68k

{libName}.68k ƒƒ {libName}.make {libName}.r
    Rez {libName}.r -append -o {libName}.68k


# Combine the two applications into a fat binary.
{libName} ƒ {libName}.PPC {libName}.68k
    Duplicate -y {libName}.68k {libName}
    Mergefragment {libName}.PPC {libName}
```

C H A P T E R 6

Creating Noncode Resources and Manipulating Resources

Contents

This chapter describes the process of building noncode resources—the resources that a Macintosh program uses to define windows, menus, dialog boxes, controls, icons, and the cursor. Before you read this chapter, you should be familiar with the Resource Manager (described in detail in *Inside Macintosh: More Macintosh Toolbox*) and you should have read through Chapter 1, “Building Macintosh Programs.”

A resource is a data structure comparable to a Pascal record or C struct; you declare and define this data structure using the Rez language just as you declare and initialize a record using Pascal or a struct using C. Resources are of two kinds: code and noncode. Code resources are created by the linker; noncode resources are built by Rez, an MPW tool, or by resource editing applications such as ResEdit.

The format of noncode resources commonly used in Macintosh programs is defined in type declaration files that are shipped with MPW. Since most programmers need to build these kinds of resources, this chapter focuses on how you create resources based on these standard types. In addition, this chapter

- describes the resource building cycle and explains how you identify a resource
- provides a summary of the standard resource type declarations shipped with MPW and explains how you write a resource definition that corresponds to a given type declaration
- explains how you can add code or noncode resources to your program and how you can delete them
- introduces the tools you use for checking the validity of a resource and for comparing resources
- explains how you can create a resource type that corresponds to your own data structure

If you are new to MPW, you should read the entire chapter except for the last section, “Creating Your Own Resource Types.”

If you are experienced using MPW and consider yourself fairly expert at writing resource definitions, you can skip this chapter and just use Appendix C, “The Rez Language,” for reference. If you are interested in creating your own resource types, you should read the last section; in this case, you will also need to refer to Appendix C, which provides complete reference information about the Rez language.

Since the 3.0 version of MPW, two changes have been introduced to the commands that relate to working with resources:

- The `ResEqual` command now also tells you whether the attributes of two resources are the same.
- The Rez and DeRez tools can now properly handle language systems that use 2-byte characters (such as Japanese or Korean) if you specify the `-script` option.

Overview

This section introduces the terms used in working with resources, discusses the tools that you use to create resources, describes the process of creating a resource based on a standard type declaration, and explains how a resource is identified.

Working With Resources

If you have never worked with resources before, it may be best to introduce the terms used in working with resources in light of a process with which you are already familiar, that of creating a program.

You write source code in a programming language such as C; you write a resource using the Rez language.

The file containing the source code for your program is called a source file; the file containing a textual description of your resource is called a **resource description file.**

Your program's source code is typically stored in two kinds of files: header files containing data type and routine declarations, and implementation files, containing the implementation of the routines. Similarly, the information describing your program's resources is also stored in two kinds of resource description files: **type declaration files**, containing the declaration of one or more resource types (for example `'MENU'`, `'WIND'`, `'DLOG'`, and so on), and **resource definition files,** containing data definitions for these types.

To create a program, you use a compiler or assembler to produce an object file, and you use the linker to link code segments into an executable program and to write these to the resource fork of the program file. To compile resources and store these in the resource fork of your program file, you use the Rez resource compiler, an MPW tool.

A program module is identified by a unique filename; a resource is identified by its type and ID.

The Resource Building Cycle

The steps required to create resources depend on the tools you choose. There are three tools you can use:

- the Rez tool, a resource compiler
- the DeRez tool, a resource decompiler
- a resource editor, an application that builds resources

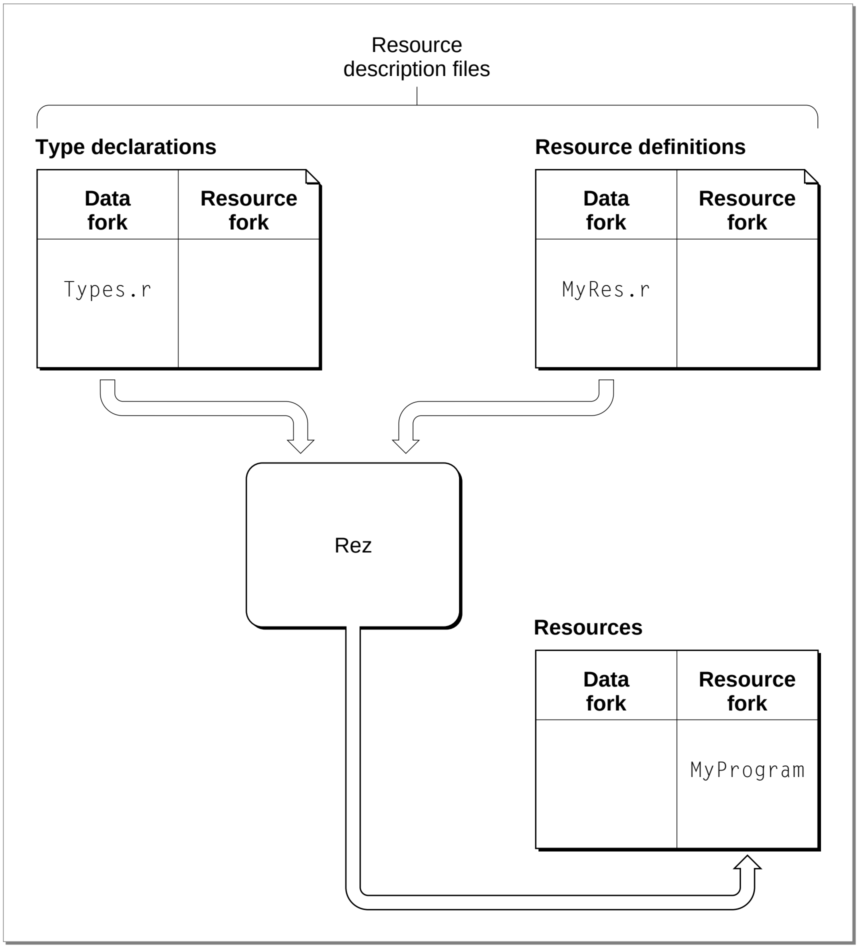
This section describes what these tools do and how you can use them singly or in combination to create resources. By learning how these tools work together, you can create resources in the way that is easiest and most convenient for you.

The Rez Tool

Rez compiles resources and writes them to the resource fork of a file. The input to Rez is one or more resource description files. A resource description file is a file you create using the MPW Shell editor; it includes the type declarations and resource definitions that Rez needs to compile the resource.

Figure 6-1 illustrates how Rez works: Rez uses the type declarations contained in the data fork of the `Types.r` file and the data definitions contained in the data fork of the `MyRes.r` file to compile a resource, which it then writes to the resource fork of the file `MyProgram`.

Figure 6-1 Creating resources with Rez



You use Rez to create a resource as follows:

1. Write a resource description file, identify the resources you have included in your resource description file, and set their attributes.
2. Use Rez to compile the resource description file.

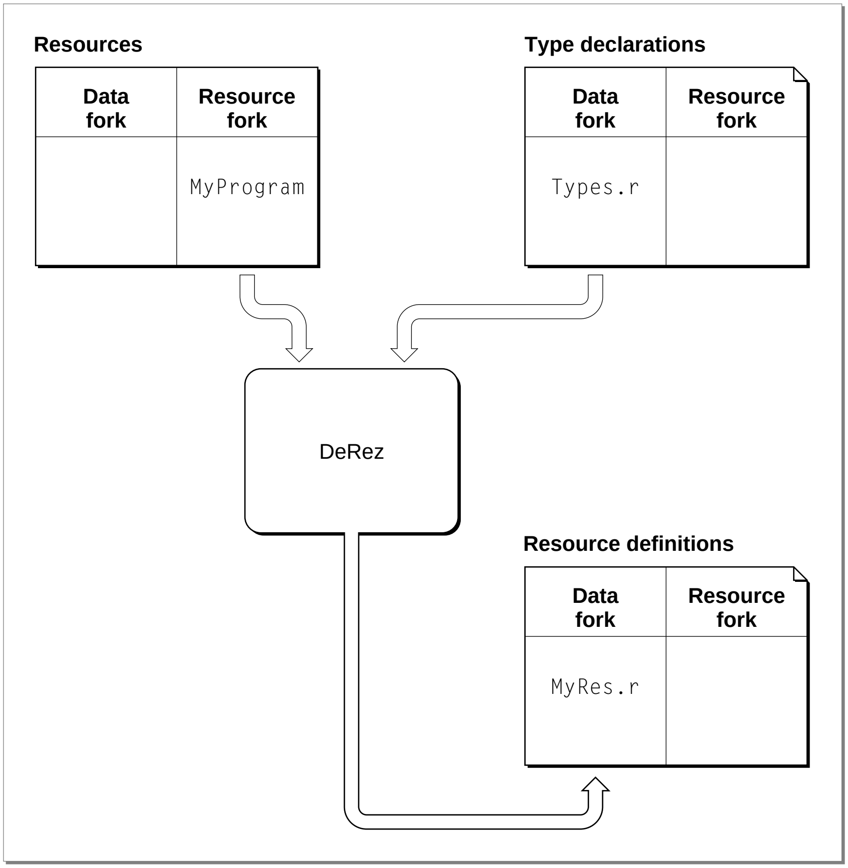
The DeRez Tool

DeRez decompiles resources according to the type declarations supplied by type declaration files that you specify as input to the tool. The resource description file produced by this decompilation contains the resource definitions associated with these type declarations. If you do not specify the appropriate type declaration files, DeRez generates hexadecimal data.

You can use DeRez to decompile the resources built by a resource editor, add comments to the resulting resource definition file, and retain the file as an archival copy.

Figure 6-2 illustrates how DeRez works: DeRez decompiles the resources in the resource fork of the file `MyProgram` and uses the type declarations contained in the data fork of the `Types.r` file to produce the resource definition file, `MyRes.r`. For additional information about DeRez, see “Using DeRez to Decompile Resources” on page 6-30.

Figure 6-2 Decompiling resources using DeRez



Resource Editors

Resource editors are interactive, graphics-oriented applications that you can use to build resources. They can be especially helpful for creating and changing graphic resources such as dialog boxes and icons.

Another advantage to using a resource editor is that you do not have to learn the Rez language. However, because you do not write a resource description file, you have no source file for your resources and no way of incorporating comments. Moreover, because the resources are written directly to the resource fork of your program file, you can lose all your resource definitions if that file is lost or becomes corrupted. For these reasons, if you plan to use a resource editor to create resources, the recommended steps are as follows:

1. Use the resource editor to create the resources you need.
2. Save these resources to a file. For historical reasons, this file was assigned a name with the suffix `.rsrc`. This convention is not enforced, but you might want to use it to remind yourself that this file contains noncode resources in its resource fork.
3. Use DeRez to decompile the resource fork of that file. DeRez outputs a resource description file to which you can add comments and which you can keep as an archival copy.
4. Use Rez to compile the resource description file produced by DeRez and to write the compiled resources to the resource fork of your program file.

If your program file is damaged, you can use Rez to recompile the resource description file you have archived and write it to the resource fork of your program file.

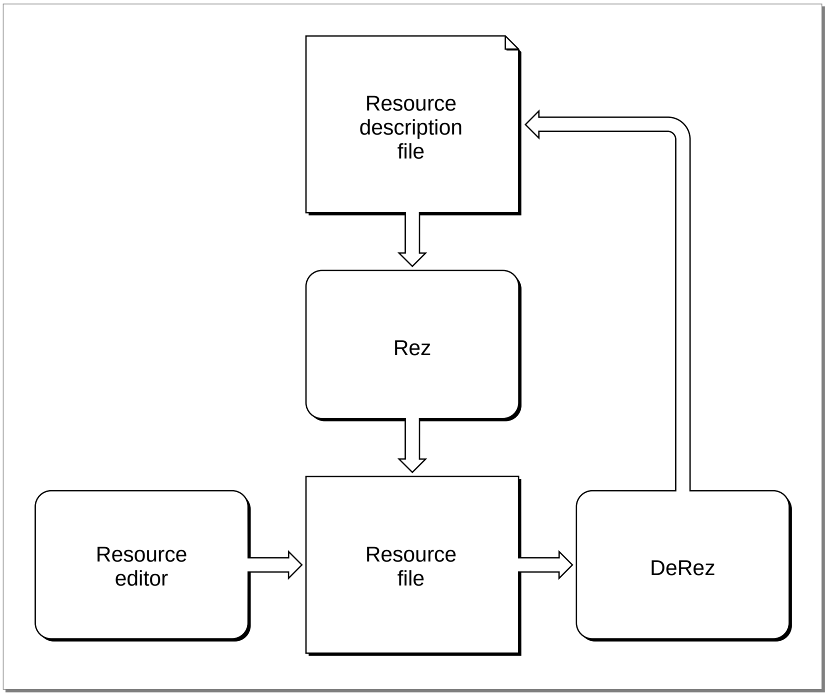
Putting It All Together

You can create resources by using just Rez or just a resource editor. The advantages of using a resource editor are that you do not have to learn the Rez language and that it is much easier to create graphic resources. However, you might not be able to use a resource editor to create all resource types; for example, ResEdit has no editors for `'crsr'`, `'errs'`, `'kscn'`, `'mach'`, `'mppc'`, and `'NFNT'` resources. You might also find that creating text resources is much easier with Rez. Experience will show which tool is more appropriate and convenient. What is important is to understand how you can use these tools together to make your work as easy as possible.

Figure 6-3 shows the resource building cycle and illustrates the complementary relationship between Rez, DeRez, and a resource editor.

Note
The term *resource file* refers to the resource fork of a file; it does not mean a special kind of a file. ◆

Figure 6-3 Resource development cycle



The Resource Description File

A resource description file is a text file that you write using the Rez language. A resource description file includes

- `Type` statements, which declare the resource type and specify the format of the data
- `Resource` statements, which define resources and specify the data as a sequence of formatted fields

A resource description file can also include other statements, comments, and directives. Table 6-1 lists the statements that can be used in a resource description file. Appendix C provides a detailed description of these statements and of the Rez directives. This chapter focuses on the `Type` and `Resource` statements because these are the statements that you use to define resources based on standard types. The `Include`, `Change`, and `Delete` statements are described in “Manipulating Resources” on page 6-31.

Table 6-1 Rez statements

Statement	Purpose
`Type`	Declares a resource type and specifies the format of the data.
`Resource`	Defines a resource and specifies the data for it as a sequence of formatted fields.
`Data`	Defines a resource and specifies the data for it as a sequence of hexadecimal bytes without any formatting.
`Read`	Defines a resource and reads the data fork of a file as the data for the resource.
`Include`	Includes previously compiled resources from the specified file and optionally changes the vital information of the resources.
`Change`	Changes the vital information of the specified resources.
`Delete`	Deletes the specified resources.

The following sections explain the conventions used to name resource description files and describe the format and contents of a resource description file.

Naming Resource Description Files

By convention, you identify a resource description file by appending a `.r` suffix to its name: for example, `MyTypes.r` and `MyResources.r`. Rez does not enforce this convention; however, using this convention is helpful in writing makefiles and in using the automated build tools.

Historically, the suffix `.rsrc` was used as a suffix to Rez output filenames. However, current practice is to write resources directly to the resource fork of the program file, which makes this intermediate file unnecessary.

The Format of a Resource Description File

This section describes the basic format of a resource description file. Listing 6-1 shows a very simple resource description file. Such a resource might be used to supply the text of a help message displayed by your application when a user is using Balloon Help.

Listing 6-1 A very simple resource description file

```
Type 'STR ' {
    pstring;
};
Resource 'STR ' (140) {
    "Click this button to display a list of files"
};
```

The sample resource in Listing 6-1 includes the following:

- One `Type` statement that defines the format of an `'STR '` resource. The resource includes one field and declares the format for that field to be a Pascal string.
- One `Resource` statement that defines the data whose format was declared by the `Type` statement and identifies the resource as a unique instance of that type by assigning it an ID, in this case 140.

Listing 6-2 shows a schematic representation of a resource description file to bring into relief the basic syntactic elements used in `Type` and `Resource` statements.

Listing 6-2 Resource description file: `Type` and `Resource` statements

```
Type 'EASY' {              // Opening brace: body of Type statement is
    data-declaration;      // enclosed in braces.
    data-declaration;      // Fields declared in Type statement are
    .                      // terminated with a semicolon (;).
    .                      // Comment can be terminated with carriage return
    .                      /* Here's another way of delimiting a comment. */
};                         // Closing brace must be followed by a semicolon.
```

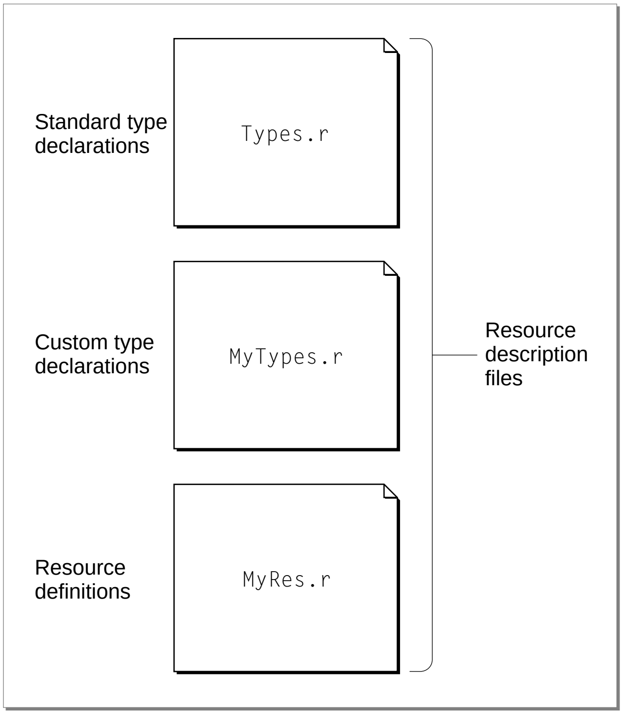
```
Resource 'EASY' (145, "My Definition") {                    // Opening brace
    data-definition,            // Body of Resource statement is enclosed in braces.
    data-definition,            // Fields defined in Resource statement are terminated
    .                           // with a comma.
    .
    .
};                              // Closing brace must be followed by a semicolon.
```

Using Separate Files for Type and Resource Statements

When you write a program, you typically divide program units into two parts. One is dedicated to declaring the routines used by the program, and the other contains the actual code to implement the routines. In the same way, when you write a resource description file, you can store `Type` statements in one file and `Resource` statements in another file. The file that contains `Type` statements is called a type declaration file; the file that contains `Resource` statements is called a resource definition file. Figure 6-4 shows three resource description files:

- A standard type declaration file, `Types.r`. If you are creating resources for types that are commonly used in Macintosh programs, you use standard type declaration files; these are kept in the MPW:Interfaces:RIncludes folder.
- A custom type declaration file, `MyTypes.r`. It contains type declarations for types not furnished by standard type declarations.
- A resource definition file, `MyRes.r`.

Figure 6-4 Resource description files



If you use separate files for `Type` and `Resource` statements, either you can specify both files as input to the Rez command when you compile the resource, or you can use the `#include` directive at the beginning of the resource definition file to specify the type declaration file. For additional information, see "Using Rez to Compile Resources" on page 6-30.

Scope of Type Statements

If you mix `Type` and `Resource` statements in the same file, the `Type` statement must precede the `Resource` statement whose format it defines; otherwise, their order does not matter.

If you provide more than one type declaration for a resource type, the last one read before the resource definition is the one that's used. This allows you to override declarations from include files or previous type declarations within the same file. Listing 6-3 shows how this might work in a sample resource description file.

Listing 6-3 Scope of `Type` statements

```
#Include MyTypes.r              // MyTypes.r includes Type statements
                                // for '1st ', '2nd ', and '4th '.
Resource '1st '(400) {
data-definition
};
Type '2nd '                 {   // Overrides Type '2nd ' declaration
data-declaration;               // from MyTypes.r.
};
Resource '2nd '(300){
data-definition
};
Type '4th '{
data-declaration;
};
Resource '4th ' {
data-definition
};
Type '2nd '                 {   // Overrides previous Type '2nd '.
data-declaration;
};
Resource '2nd ' (301) {
data-definition
};
```

Identifying a Resource and Setting Its Attributes

When your program needs to use the data contained in a resource and makes the appropriate call, the Resource Manager locates the specified resource and calls the Memory Manager to load it into memory. In order for the Resource Manager to find the resource in the resource fork of your file, you must identify it in a specific way when you create it.

This section describes how you identify a resource and what other information you need to specify so that the Resource Manager can load the resource at the right time and in the right place. Although this section duplicates information presented in *Inside Macintosh*, it focuses on how this information relates to building resources using Rez. You should skim through this section even if you are already familiar with the concepts as they are presented in *Inside Macintosh*.

Resource Type, ID, and Name

The information used to identify a resource is called the **resource specification.** A resource is uniquely identified by its type and ID or by its type, ID, and name:

- The type of a resource determines what data is in a resource and how the data is formatted.
- The ID of a resource identifies one instance of that type.
- The name of a resource is required only if you call the `GetNamedResource` function to load the resource; otherwise, the name is optional.

You specify the type, ID, and name of a resource in the `Resource` statement used to define the resource. The syntax for the identifying information is

`Resource` *resource-type* (*ID* [, *resource-name*] [, *attribute* [`|` *attribute*]...])

Here are two examples:

```
Resource 'DLOG' (1000)
Resource 'MENU' (300, "Edit Menu")
```

Each resource must have an ID and a type. Resource IDs and names must be unique for a given type. Resource names are not case sensitive. However, there are some requirements and restrictions for the type, ID, and name specifications. These are described in Table 6-2.

Table 6-2 Resource identification information

Information	Type and storage	Restrictions
Type	Four-character literal, long expression	Case sensitive; enclose in straight single quotation marks.
ID	Integer, word expression	Must be in the range 128 through 32767. Other values are reserved for system use. See the Resource Manager chapter of *Inside Macintosh* for additional information.
Name	String	Enclose the string in straight double quotation marks.

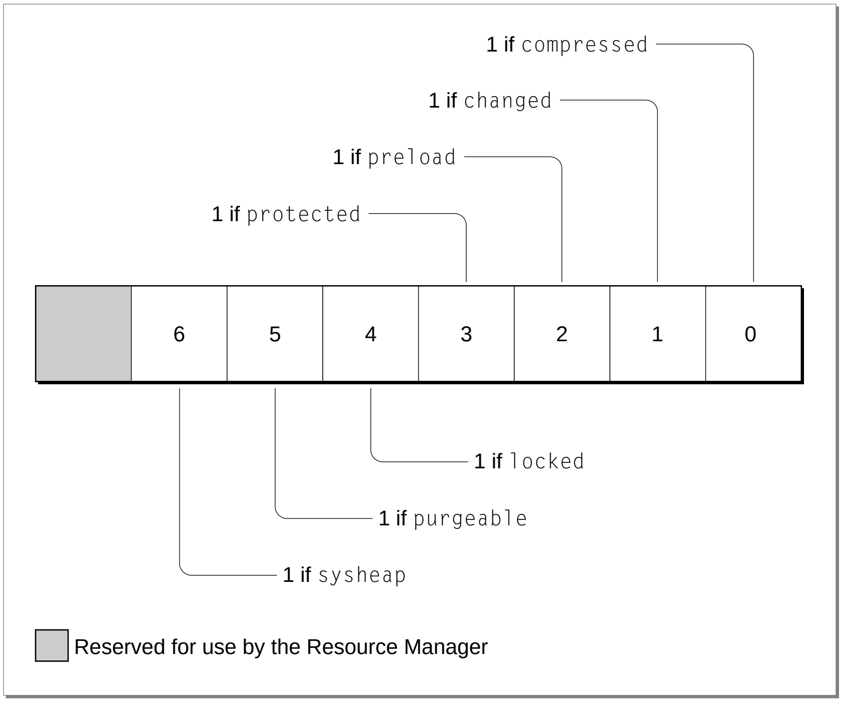
If your program accesses a resource by calling the `GetResource` function, which locates the resource based on its type and ID, the name is optional. However, even in this case, assigning a name to the resource is a good idea. For example, if your application uses lots of menus and you use MacsBug to find out whether they are being loaded into memory, you get more useful information if each `'MENU'` resource has a name that identifies which menu it is.

Resource Attributes

The attributes of a resource are used by the Memory Manager to determine how the resource is to be handled: for example, where and when it is to be loaded, whether it can be changed, and whether it can be purged. Each attribute of a resource is specified by a bit in the low-order byte of a word.

You can specify an attribute by using its constant name or its numeric value. For example, to specify that the resource should be locked, you can use the constant name `locked` or the numeric value 16. Figure 6-5 shows the resource attribute bits.

Figure 6-5 Resource attribute bits



If you do not specify any settings for these attributes, Rez sets all the bits to 0 by default. The meanings of the default settings are shown in the last column of Table 6-3.

You can change these default settings when you define the resource with a `Resource` statement or after the resource has been compiled, using the `Include` or `Change` statement. See "Manipulating Resources" on page 6-31 for information on how you use the `Include` or `Change` statement to reset attributes.

In setting a resource's attributes, you can specify one of the following:

- a single constant, shown in the first two columns of Table 6-3
- a numeric value, shown in the third column of Table 6-3
- two or more keywords or numeric values in an expression (you use the `OR` logical operator to combine values)

Table 6-3 Resource attributes

Constant Names			
Default (0)	**Alternate (1)**	**Set value**	**Meaning**
`appheap`	`sysheap`	64	Specifies whether the resource is to be loaded into the application heap or the system heap.
`nonpurgeable`	`purgeable`	32	Specifies whether the Memory Manager can purge the resource.
`unlocked`	`locked`	16	Specifies whether the resource is locked. Locked resources cannot be moved by the Memory Manager. The `locked` attribute overrides the `purgeable` attribute because a locked resource cannot be purged.
`unprotected`	`protected`	8	Specifies whether the resource is protected. Protected resources cannot be modified by the Resource Manager.
`nonpreload`	`preload`	4	Specifies whether the resource is to be preloaded. Preloaded resources are placed in the heap when the Resource Manager opens the resource file.
`unchanged`	`changed`	2	Tells the Resource Manager whether the resource has been changed. Rez does not allow you to set this bit, but DeRez displays it if it is set.
`uncompressed`	`compressed`	1	Specifies whether the resource is compressed. This bit is defined only for System 7 or later. Although DeRez displays the setting for this bit, you should not set it.

The following three examples have the same effect; the first two are preferable for stylistic reasons.

```
sysheap | purgeable | protected

64 | 32 | 8

sysheap | 32 | protected
```

You specify the resource's attributes following its ID or name. In the following two examples, one using keywords and the other numeric values, the same attributes are set:

```
Resource 'DLOG' (1000, "First Dialog", preload | purgeable)
Resource 'DLOG' (1000, "First Dialog", 4 | 32 )
```

Creating a Resource Based on a Standard Type

You use standard resource types, which are stored in standard type declaration files that are shipped with MPW, to declare the format of resources your application uses to define windows, menus, dialog boxes, icons, cursors, and other commonly used data.

The first subsection, "Standard Type Declaration Files," describes the contents of the type declaration files found in the RIncludes folder. The following subsection, "A Cookbook Example," offers a hypothetical `Type` statement with instructions on how to write a corresponding `Resource` statement.

Standard Type Declaration Files

The RIncludes folder contains standard type declaration files. Table 6-4 lists these and briefly describes their contents.

Table 6-4 Standard type declaration files

Filename	Contents
`BalloonTypes.r`	Type declarations for Balloon Help. For information on writing Balloon Help, see the Help Manager chapter in *Inside Macintosh*.
`Cmdo.r`	Type declarations for Commando resources. For information on writing Commando resources, see Chapter 14, "Creating Commando Dialog Boxes for Tools and Scripts."
`CTBTypes.r`	Type declarations for the Macintosh Communications Toolbox.
`InstallerTypes.r`	Type declarations for installer script templates.
`MPWTypes.r`	Type declaration used to build 68K-based drivers written in high-level languages.
`Pict.r`	`'PICT'` type declarations used with DeRez for decompiling and archiving PICT files.
`SIOW.h`	Header file used for tools that can run outside of MPW. Although not a `.r` file, it is in this folder because it defines constants used in the `siow.r` file.
`siow.r`	Resource file for tools that can run outside of MPW. For additional information, see Chapter 15, "Building SIOW Applications."
`SysTypes.r`	Type declarations for resources used by the Mac OS.
`Types.r`	Type declarations for resources used by applications.

- If you are using Rez to compile a resource, you need to specify the name of the type declaration file that enables Rez to interpret the resource definition file. You specify the name of this file either on the Rez command line or in the resource definition file. For more information, see "Using Rez to Compile Resources" on page 6-30.
- If you are using DeRez to decompile a resource, you need to specify the name of the type declaration file that enables DeRez to produce a resource definition file. You must specify the name of the file on the DeRez command line. If you do not, DeRez will output the data in hexadecimal form without any additional format information. For more information, see "Using DeRez to Decompile Resources" on page 6-30.

In creating resource description files for an application, you are most likely to use the `Types.r` and `SysTypes.r` files. To make your work easier, the resource type names are already marked. So, for example, if you need to write a resource definition for a dialog box, open the `Types.r` file, and choose the DLOG item from the Mark menu. The MPW Shell editor finds the `'DLOG'` type declaration in the file and moves the cursor to the beginning of that `Type` statement. You can now write a `Resource` statement that corresponds to the `Type` statement for the `'DLOG'` type. It is essential that you understand the `Type` statement in order to write a valid `Resource` statement. The next section explains how to match the data definitions in the `Resource` statement to their corresponding declarations in the `Type` statements.

Standard type declaration files other than `Types.r` and `SysTypes.r` are provided for more specialized tasks: for example, building drivers or Commando dialog boxes. The cross-references provided in Table 6-4 tell you where to get more information on how to build and use these resources.

A Cookbook Example

Once you have found the standard type declaration that describes the type of resource you want to create, you need to write a `Resource` statement in which you define the data for the fields declared in the `Type` statement.

The `Type` statement shown in Listing 6-4 is a hypothetical example, designed to contain every kind of data declaration you would encounter in a standard type declaration file. The sections following Listing 6-4 explain the structure of the `Type` statement and how to write a `Resource` statement based on that type. You can get the most out of this example if you try to write your own `Resource` statement for the type shown in Listing 6-4 as you read through these sections. Compare your solution to the one offered in the section “Sample 'BOOM' Resource Statement” on page 6-29.

Entries in Listing 6-4 are shown in normal and underlined text. Only the underlined entries are required in the corresponding `Resource` statement. Comments indicate the name of the section that describes the options you have in providing data for the corresponding `Resource` statement field.

Listing 6-4 A cookbook example: `Type` statement for `'BOOM'`

```
#ifndef SystemSevenOrLater                      //"Resolving Defines" on page 6-28
#define SystemSevenOrLater 0                    //"Resolving Defines" on page 6-28
#endif                                          //"Resolving Defines" on page 6-28
Type 'BOOM' {
        rect = {0,0,16,16};             //"Format of Data Declarations" on page 6-24
        point = {1,2};                  //"Format of Data Declarations" on page 6-24
        integer;                        //"Format of Data Declarations" on page 6-24
        byte   invisible ,visible;      //"Format of Data Declarations" on page 6-24
        boolean;                        //"Format of Data Declarations" on page 6-24
        fill byte;                      //"Data Alignment" on page 6-25
        char = "$";                     //"Format of Data Declarations" on page 6-24
        integer = $$CountOf(SomeArray); //"Defining Structured Data" on page 6-25
        wide array SomeArray{           //"Defining Structured Data" on page 6-25
            integer;
            literal longint;
        };
        mylabel:                                //"Labels" on page 6-28
        integer = mylabel;                      //"Labels" on page 6-28
        switch {                                //"Switch Types" on page 6-27
            case Griffon:
                key integer = 1;
                integer;
            case Jabberwocky:
                key integer = 2;
                integer;
        };
        align word;                     //"Data Alignment" on page 6-25
        pstring [255];                  //"Format of Data Declarations" on page 6-24
#ifdef SystemSevenOrLater               //"Resolving Defines" on page 6-28
        align word;                     //"Data Alignment" on page 6-25
        unsigned integer        noAutoCenter       =  0x0000,
                                centerMainScreen = 0x280a,
                                alertPositionMainScreen = 0x300a,
                                staggerMainScreen = 0x380a;
#endif
};
```

Note
The Rez language does not include a continuation character. The use of delimiters (braces, commas, and semicolons) determines how Rez interprets the information provided. ◆

Format of Data Declarations

Look at the `Type` statement in Listing 6-4 and notice that each data declaration begins with a keyword indicating the declaration type (`byte`, `boolean`, `integer`, `char`, `pstring`, `rect`, `point`, `array`, or `switch`).

A data declaration can take one of three forms; here is an example of each:

```
rect = {0, 0, 16, 16};
byte     invisible, visible;
boolean;
```

The first data declaration

```
rect = {0, 0, 16, 16};
```

specifies a `rect` structure and a value for the `rect`. In this case, no corresponding data definition would appear in the `Resource` statement because a value has already been assigned with the `Type` statement. You can think of the `Type` statement as a template; it is possible that some fields of this template, as is the case here, have already been initialized.

The second data declaration

```
byte    invisible, visible;
```

uses an enumeration to list possible values for the field. The corresponding data definition in the `Resource` statement can specify one of these constant names or its corresponding value. In an enumeration, each item is implicitly equated with its ordinal place in the list, beginning with 0. In this case, you would specify either `invisible` (0) or `visible` (1) for this field.

Some enumeration lists specify values for one or more items in the list, as in this example:

```
byte    GrowBox, ShrinkBox, NoBox = 8, Color;
```

In this case, `GrowBox` equals 0 and `ShrinkBox` equals 1. If an item is assigned a specific value in the list (`NoBox = 8`), the item following is implicitly equated to the previous value plus one; thus, `Color` is equal to 9.

The third data declaration

```
boolean;
```

declares a Boolean field. You must supply the data for this field in the `'BOOM'` `Resource` statement—for example, `False` or `True`.

The Boolean, numeric, or string data you specify for a data definition can be expressed in a variety of formats; for more information, see the description of the `Resource` statement on page C-25 in Appendix C.

Data Alignment

Although resources always start on an even boundary, no implicit alignment is provided for the data fields defined in the resource. The resource is treated as a bit stream; integers and strings can start at any bit. The `align` and `fill` types are used in `Type` statements to provide explicit alignment. You should not include these entries in your `Resource` statement.

See the description of the `Type` statement on page C-32 in Appendix C for additional information about data alignment.

Defining Structured Data

The `Type` statement can also contain declarations for structured data types: `rect`, `point`, and `array`. The data you specify for these structures in the corresponding `Resource` statement must always be enclosed in braces. The closing brace must be followed by a comma. (Data specified for a `switch` type must also be enclosed in braces.)

In Listing 6-4, values are provided for the `rect` and `point` declarations. Since no values are provided for the elements in the array, you must do so in the `Resource` statement.

An `array` declaration can specify that the array contains a predetermined number of elements; if it does, that number is declared following the array name. The array `SmallArray`, shown in the next example, contains three elements; each element contains two fields, `integer` and `char`.

```
array SmallArray[3] {
    integer;
    char;
    };
```

You must specify exactly three elements in the corresponding `resource` statement, as in this example:

```
{3, "j"; 15, "a"; 54, "b"},
```

An array can also be declared to be of variable size. In this case, the array name is not followed by a number, but the array declaration is usually preceded by a statement like

```
integer = $$CountOf(SomeArray)
```

Here's an example that declares the `SomeArray` array to be of variable size:

```
integer = $$CountOf(SomeArray)
    wide array SomeArray{
        integer;
        literal longint;
    };
```

The `$$CountOf` entry is a Rez function that returns the number of elements in an array. You can recognize Rez functions by the `$$` prefix. For additional information about Rez functions that are used with arrays, see "Array Information" on page C-7 in Appendix C.

The array declared in the `'BOOM'` type has an indeterminate size, so you can assign as many values as you need. Each element of the array contains two fields, `integer` and `literal longint`. A valid entry for the array in the `Resource` statement could look like this:

```
{1, 'MENU'; 22, 'DLOG'; 13, 'DITL'; 45, 'CODE'},
```

Note that the array data is enclosed in braces, that elements of the array are separated by semicolons, and that fields are separated by commas. For additional information about arrays, see the description of the `Type` statement on page C-32 in Appendix C.

Switch Types

The `Type` statement can also contain a `switch` type declaration. The `Switch` declaration in Listing 6-4 looks like this:

```
switch {
        case Griffon:
                key integer = 1;
                integer;
        case Jabberwocky:
                key integer = 2;
                integer;
};
```

Like the data supplied for the `array` definition, the data for the `switch` definition must be enclosed in braces. To complete the corresponding field in the `Resource` statement, you must select the case you want and then supply the data for it. A possible data definition that corresponds to this declaration might look like this:

```
Jabberwocky { 201 },
```

Note that the case selector (`Jabberwocky`) is *not* followed by a colon in the `Resource` statement.

The entry

```
key integer = 1;
```

is called a *key definition*. The key definition is compiled by the resource compiler into the resource to identify the case variant; it is also used by DeRez to decode the variant. For more information, see the description of the `Type` statement on page C-32 in Appendix C.

Labels

Any field that begins with a word followed by a colon, for example,

```
mylabel:
```

is a label used to calculate the offset of data in a resource. You can use labels only in `Type` statements. Do not include labels in the `Resource` statement. For additional information about labels, see page C-9 in Appendix C.

Resolving Defines

In addition to the keywords described so far, the `Type` statement can also include keywords that are preceded by a number sign (#). These are preprocessor directives for the Rez compiler. The only directives you need to be concerned about are those that conditionally compile some part of the resource if a term has been defined. A common example is shown in Listing 6-5.

Listing 6-5 Conditional compilation directives

```
#ifndef SystemSevenOrLater
#define SystemSevenOrLater 0
#endif
Type 'BOOM'     {
                        //part of resource that is compiled
                        //whether or not SystemSevenOrLater is defined
.
.
#ifdef SystemSevenOrLater
        align word;
        unsigned integer        noAutoCenter = 0x0000,
                        centerMainScreen = 0x280a,
                        alertPositionMainScreen = 0x300a,
                        staggerMainScreen = 0x380a;
#endif
};
```

The `#ifdef` directive says that if the identifier `SystemSevenOrLater` has been defined with a `#define` directive in the corresponding `Resource` statement, then the data definitions between the `#ifdef` and `#endif` directives should be compiled. If you do define `SystemSevenOrLater`, you will need to supply a value for the additional `unsigned integer` field: for example,

```
noAutoCenter,
```

To trigger the compilation of this field, you can use the `#define` directive before the `#include` directive that merges the `Types.r` file into the resource definition file, as in this example:

```
#define SystemSevenOrLater
#include Types.r
Resource 'BOOM' (1000, "One Possible Definition") {
...
};
```

Or, you can use the `-d` option in the Rez command line to define the specified identifier, as in this example:

```
Rez MyResource.r -d SystemSevenOrLater
```

For information about preprocessor directives, see the section "Preprocessor Directives" on page C-12 in Appendix C.

Sample 'BOOM' Resource Statement

Here is one possible `Resource` statement based on the `'BOOM'` type. If you tried writing your own, check the structure of your statement, especially the use of delimiters (braces, commas, and semicolons), against the following statement. The data you specified may be different.

```
#define SystemSevenOrLater;
#include "MyBoomType";
Resource 'BOOM' (1000, "One Possible Definition") {
        8,
        invisible,
        true,
        {10, 'MENU'; 2, 'DLOG'; 28, 'APPL'},
        Jabberwocky { 201 },
        "Almost the end of the solution",
        noAutoCenter
};
```

Using Rez to Compile Resources

Once you have created a resource description file, you use the Rez compiler to compile the resources described in the file and write these to the resource fork of your program file. This section describes the Rez tool and provides a summary of its options. For more detailed information, see the description of the Rez tool in *MPW Command Reference*.

The syntax of the Rez tool is

```
Rez  [option...] [resource-description-file...]
```

The *resource-description-file* parameter specifies the file or files that serve as input to Rez. If you don't specify a file, Rez takes its input from standard input. By default, Rez writes its output to the file `Rez.out`. To have Rez write the output to another file, use the `-o` option and specify the name of the output file.

You can specify the resource description files that are to be compiled in one of two ways:

- Specify in the Rez command line the names of all the files; for example:

  ```
  Rez Types.r MyTypes.r MyRes.r
  ```

- Use the `#include` directive to merge the specified files into the compilation. For example, if the file `MyRes.r` contained the directives

  ```
  #include Types.r
  #include MyTypes.r
  ```

  you could compile the resource using the command

  ```
  Rez MyRes.r
  ```

For a summary of options for the Rez command, see the *MPW Command Reference*.

Using DeRez to Decompile Resources

In addition to using DeRez to decompile resources you have created using a resource editor, you can also use it to do the following:

- Decompile resources you have not written yourself to see what they contain and then, if you like, to change them.

- Decompile resources you have written yourself *without* specifying a type declaration file in order to check the storage of data in a resource. This is a good technique to use for checking the allocation of data for resource types you have created yourself. For additional information, see "Creating Your Own Resource Types" on page 6-36.

This section describes the DeRez tool and provides a summary of its options. For more detailed information, see the description of DeRez in *MPW Command Reference*.

The syntax of the DeRez tool is

DeRez [*option ...*] *resourceFile* [*typeDeclarationFile...*]

The *resourceFile* parameter specifies the name of the resource file you want to decompile. The *typeDeclarationFile* parameter specifies the file that contains the type declarations that DeRez uses when decompiling the resources. If you do not specify a type declaration file, DeRez output consists of data statements that give the resource data in hexadecimal form with no format information.

The DeRez command sends output to standard output. To write the output to a file, use a redirection operator, as in this example:

```
DeRez MyProgram Types.r > MyResources
```

See the *MPW Command Reference* for a list of options for the DeRez tool.

Manipulating Resources

In addition to using Rez to compile resources and write them to the resource fork of a file, you can also use it to modify that file. You can use the statements described in Table 6-5 to change the identifying information of resources contained in the file, to delete resources, or to include additional resources. Although you can use Rez to build only noncode resources, you can use it to manipulate any type of resource, whether it has been built by Rez, by a third-party resource compiler, or by a linker.

Table 6-5 Changing an existing resource file

Statement	Description
`Change`	Changes a resource's identifying information.
`Delete`	Deletes one or more resources from the output file.
`Include`	Merges one or more (compiled) resources from the specified file into the Rez output file. Syntax variations allow you to change the type and attributes of the resources to be merged; for an example, see the section "Functions That Return Information About the Current Resource," beginning on page 6-35.

▲ **WARNING**

When Rez writes its output to a file, it overwrites everything in the file. For this reason, when using the `Change` and `Delete` statements, you must use the `-a[ppend]` option with Rez. You are changing an existing resource, not creating a new one from scratch. Note also that when you use the `Delete` and `Change` statements, you do not retain an original of the resource you are changing. However, when you use the `Include` statement, only a copy of the original resource is included in the output file. ▲

There are two ways you can change, delete, or include resources. If you are dealing with only a few changes, you can enter commands like the following:

```
Echo "Change 'MENU' (135) to 'MENU' (300);" | Rez -a -o MyFile
```

In this example, Rez takes its input from the console (that is, the output of the `Echo` command). This `Change` statement changes the ID of the `'MENU'` resource in `MyFile` from 135 to 300.

In many cases, you may need to make a number of changes to the resource fork of a file. In such cases, you might want to create a file containing all your `Change`, `Delete`, and `Include` statements and input that file to Rez. For example, a file called `ChangeFile` might include the following statements:

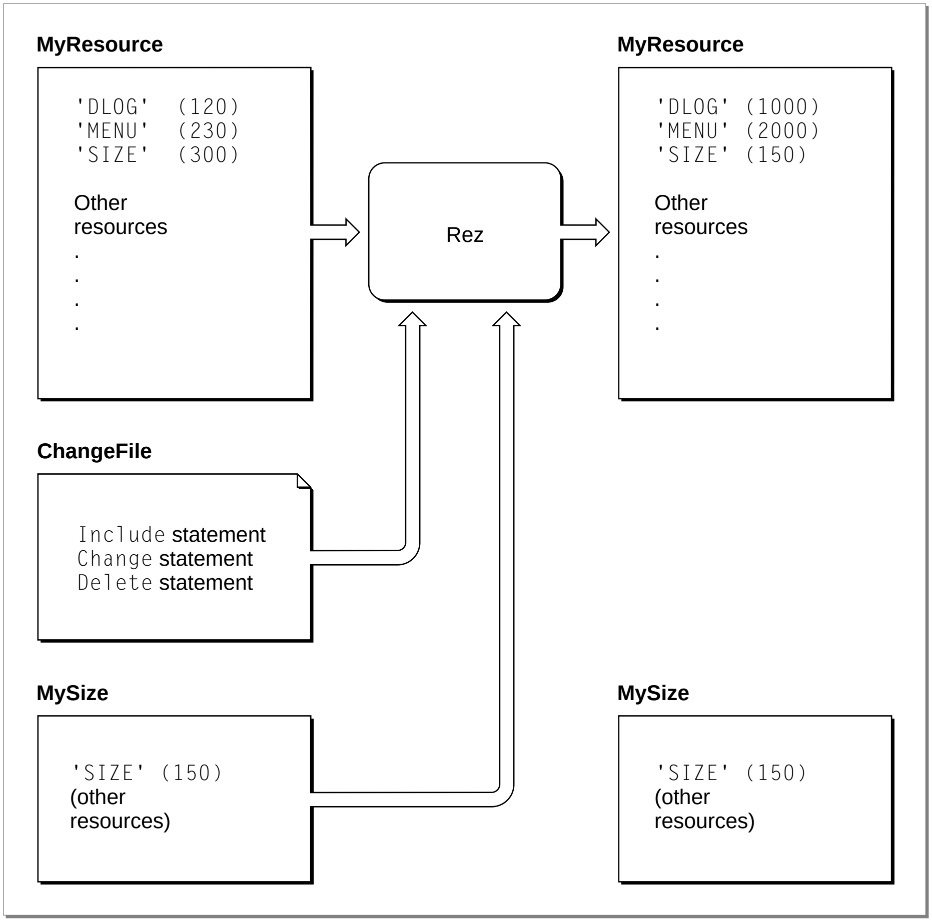
```
Include "MySize" 'SIZE' (150);              /* include new 'SIZE' res */
Delete 'SIZE' (300);                        /* delete old 'SIZE' res */
Change 'DLOG' (120) to 'DLOG' (1000);       /* change ID of 'DLOG' */
Change 'MENU' (230) to 'MENU' (2000);       /* change ID of 'MENU' */
```

To have Rez implement these changes in a resource file called `MyResource`, use this command:

```
Rez ChangeFile -a -o MyResource
```

Note the use of the `-a` option to prevent Rez from completely overwriting the other resources in the file `MyResource`. The effect of this command is shown in Figure 6-6.

Figure 6-6 Effect of `Change`, `Delete`, and `Include` statements



For more detailed information about the syntax of these statements, see the descriptions of the `Change` (page C-19), `Delete` (page C-21), and `Include` (page C-22) statements in Appendix C.

Functions That Return Information About the Current Resource

Four Rez functions return information about the current resource—that is, the resource that you are including or changing. (Table 6-6 summarizes these functions.) Using these functions can save you a lot of time if you need to change the identifying information for a lot of resources in the same file.

Table 6-6 Rez functions that return information about the current resource

Function	Returns
`$$Attributes`	Numeric value that specifies the attributes of the current resource.
`$$ID`	Numeric value that specifies the ID of the current resource.
`$$Name`	String that specifies the name of the current resource.
`$$Type`	Numeric value that specifies the type of the current resource.

You can use these functions with `Include`, `Change`, or `Delete` statements. The following two examples illustrate some possible uses.

This `include` statement merges `'DRVR'` resources from `MyFile`.

```
Include "MyFile" 'DRVR' (0:40) AS

'DRVR' ($$ID, $$Name, $$Attributes | 64);
```

The statement causes '`DRVR`' resources that are from `MyFile` and have ID numbers 0 through 40 to be merged into the Rez output file. The ID range sets up an implicit loop; each time through the loop another `'DRVR'` resource is merged. The merged resources have the same name and ID as the resources in `MyFile`, but also have the `sysheap` attribute set.

You can specify an attribute as a numeric expression, or you can set them individually by specifying one or more of the keywords from any of the pairs listed in Table 6-3. In the example just given, the `sysheap` attribute (64) is combined (using the `OR` logical operator) with the current value of `$$Attributes` because `$$Attributes` is a numeric expression. For additional information about resource attributes, see the section “Resource Attributes” on page 6-17.

The following command sets the protected bit on all `'CODE'` resources:

```
Change 'CODE' to $$Type ($$Id,$$Name, $$Attributes | 8);
```

For additional information about Rez functions, see “Rez Functions” on page C-5 in Appendix C.

Checking Resources

MPW provides two tools that you can use to compare resource files and to check their validity:

- The `ResEqual` tool compares the resources in two files and writes their differences to standard output. The command checks that the specified files contain resources of the same type and ID, that the sizes of corresponding resources are the same, that their attributes are the same, and that their contents are the same.
- The `RezDet` tool checks the resource fork of specified files for damage or inconsistencies.

For additional information about the output of these tools, please see the description of each command in *MPW Command Reference.*

Creating Your Own Resource Types

The files `SysTypes.r` and `Types.r` provide resource type declarations for the standard parts of the user interface and for some of the data structures that are common to all applications running on the Mac OS. However, you might want to write resource type declarations that describe the format of data structures that are specific to your program. For example, if you have global data that you don’t need to store in memory all the time, or if you have data that changes and you do not want to have to rebuild your program as a result, you should consider storing this data in resources. Of course, when you create your own resource types, you need to call routines that load the resources and implement your own routines to manipulate data stored in the resources just as the Window Manager or the Menu Manager does for data stored in `'WIND'` and `'MENU'` resources.

If you do want to create your own resource types, you also need to read through Appendix C, especially the information on the `Type` statement on page C-32.

Structured Data Types and Resources

You normally store data in structured data types (C structs) and access this data by referencing the name of a field of the record or struct. To store this data in a resource, to load the resource into memory, and to manipulate the fields of the resource programmatically, you must do the following:

1. Write the resource type declaration so that it stores data in the same way as your record does. The data in the resource must be allocated in the same order and must take up the same amount of space as the data in your record.
2. Write a resource definition that assigns initial values to the fields of the resource and identifies the resource.
3. Use Rez to compile the resource.
4. Call the Resource Manager (from your program) to load the resource and return a handle to the resource.
5. Associate the resource handle with the corresponding record or struct definition.
6. Manipulate values stored in the record or struct as needed by your program.

The trickiest part of this process is the first step. You must understand how the Rez compiler stores data and how the C compilers store data. The next section, "A Resource Type Based on a C Struct," describes how you can use the Rez language to create a data structure that corresponds in size, type, and location to that created by the MPW C compilers. But unavoidably, you will have to do a lot of fine-tuning by building your type declarations step by step, writing a resource definition file using recognizable values, and then getting a hex dump of the object file to see exactly how your data was allocated.

To avoid excruciating adjustment problems between C structs and Rez data types, you should note the following when creating your resources:

- If your data structure must include a variable-length field, make it the last field so that you don't have to dynamically calculate the starting address of the next field.
- If you must include a variable-length field and cannot make it the last field, see the discussion of labels in the description of the `Type` statement on page C-32 in Appendix C for information on how you can calculate the starting address of the next field.

A Resource Type Based on a C Struct

Figure 6-7 shows a `Type` statement based on a C struct. The items in the list that follows correspond to the line numbers shown in the figure; they explain some of the more troublesome issues in representing C data types in the Rez language.

Note that there is no way to represent floats and doubles in the Rez language except through the use of hex strings.

2. A C `short` corresponds to a Rez integer.
6. A C `long` or `int` corresponds to a Rez `longint`.
10. The Rez `switch` type corresponds to a C `union`, where the tag field (however you differentiate) is the key field in each part of the `switch`.

Figure 6-7 A resource type based on a C struct

	C `struct` declaration	Rez `Type` statement
1	`typedef struct {`	`Type 'THEr' {`
	`char aChar;`	`byte;`
		`align word;`
2	`short anInt;`	`integer;`
		`align word;`
3	`long aLongInt;`	`longint;`
4	`unsigned char myChar;`	`unsigned byte;`
		`align word;`
5	`unsigned short anInt;`	`unsigned integer;`
		`align word;`
6	`unsigned long aLongInt;`	`unsigned longint;`
		`align word;`
7	`short anIntArray[3];`	`array [3] {`
		`integer;`
		`};`
		`align word;`
8	`char aBooleanArray[2];`	`array [2] {`
		`byte;`
		`};`
		`align word;`
9	`String255 aPString;`	`pstring[255];`
10	`short recordType;`	
	`union {`	`switch {`
		`hasInteger:`
		`key integer = 1;`
		`align word;`
	`integer innerInteger;`	`integer;`
		`hasChar:`
		`key integer = 2;`
		`align word;`
	`char innerChar;`	`byte;`
	`} inner;`	`};`
	`} theRecord;`	`};`

Rez stores bit fields in the same order as it expresses them. The following C declaration and Rez `Type` declaration allocate space in the same way:

```
struct {
        int a: 1;
        int b: 2;
}
Type 'FRED' {
        bitfield[1]
        bitfield[2]
}
```

Notice the use of `align` types in the `Type` statement in Figure 6-7. The C compiler aligns to byte boundaries variables of type `char` and variables of type `enum` that require only a single byte of memory. C aligns all other types on word boundaries. Signed and unsigned types have the same size and alignment.

Each new field of a structure begins on an even address except single-byte fields, which begin on an odd address if the previous field contained an odd number of bytes.

Arrays of records are word aligned; each record in an array is word aligned.

In Figure 6-7 the only `align` types actually needed to make sure that the data defined by the struct and that defined by the resource occupy identical locations in memory are the `align word` declaration following the `byte` field (item 1) and the `align word` declaration following the `unsigned byte` field (item 4). It is recommended that you align all fields in the resource to word boundaries. This might sometimes prove unnecessary because the data that follows already starts on a word boundary. If this is the case, the explicit alignment might be redundant, but it cannot cause any harm.

Note
All discussion of C in this section refers to the MrC/MrCpp and SC/SCpp compilers. A different C compiler might do things differently. Please check the documentation provided with your compiler for information on the storage and size of data types. ◆

Source Code Porting Checklists

Contents

This chapter contains information on adapting your source code to run under a different runtime architecture (for example, if you are adapting classic 68K code to compile and run in the PowerPC runtime environment). In addition, some information is given about porting your code from non–Mac OS environments, such as UNIX®.

You should also read this chapter if you are planning to write fat binary programs, as the information on adapting code can also be used to create portable code that can compile for either PowerPC-based or 68K-based machines. A list of conditional compilation variables is included to aid in this process.

If you are porting between Mac OS runtime architectures, you should also check "Switching Between Libraries," beginning on page 9-10, to determine the equivalent libraries for each architecture.

In addition to the checklists below, you can read other documents that describe aspects of the porting process. Some useful ones are *Moving Your Source to PowerPC* and various *develop* journal reprints found in the Develop folder.

Porting to the PowerPC Runtime Environment

Here are issues you should keep in mind when adapting code from the 68K platform to the PowerPC platform.

Upgrade to the Latest Headers

If you have not already done so, you should update to use the universal C headers found in the folder CIncludes.

Remove Inline Machine or Assembly Code

You should replace instances of inline code with lines written in C.

Be Explicit When Making References to Type int

You should not assume the size of an integer. For example, THINK C assumes a type `int` is 16 bits, while the MrC compiler assumes 32 bits. Remember that functions without a declared return type are assumed to return a type `int`, so all return values should be stated explicitly.

You may find it useful to create a header file that sets explicit type definitions (for example, 16-bit `int` or 32-bit `int`) for each compiler by using conditional compilation variables. You can then add this header to your source code whenever you need to be sure of the size of an integer variable.

Check Floating-Point Formats

The Motorola 68881/68882 floating-point math coprocessors use an 80-bit or 96-bit floating-point data type, while the PowerPC microprocessor uses an IEEE 754 standard 64-bit `double` data type. (It also supports a `long double` of 128 bits.) If your code contains 80-bit or 96-bit extended numbers, you should convert these to type `double` or `long double`.

Check Data Structure Alignment

PowerPC microprocessors can process data most efficiently if the data is aligned according to size. All 2-byte values should begin on an even address and 4-byte values should begin on an address that is a multiple of four. Dummy bytes are often added by the compiler to preserve the proper alignment. Note that MrC has a `pragma options align` directive that you can use to impose 68K alignment rules. You should use this pragma whenever you have data structures that you will pass between 68K and PowerPC code. (The universal headers already include such directives for Mac OS data structures.)

Remove the pascal Keyword

In the PowerPC runtime environment, Pascal and C calling conventions are identical. If you declare a function with the `pascal` keyword, the PowerPC compiler uses the same calling convention as if you had declared it without the `pascal` keyword. In PowerPC compilers, function parameters are pushed onto the stack from left to right.

Check enum Type Size Dependencies

Classic 68K compilers create the smallest type `enum` variables necessary (either 1-, 2-, or 4-byte) by default, while PowerPC compilers have a default `enum` size of 4 bytes. If your code depends on the sizes of the variables declared as type `enum` (for example, in data structures for Toolbox calls), you should explicitly specify the `enum` size to the compiler in your build.

Adhere to Strict ANSI Function Declarations and Prototypes

The MrC compiler is a true ANSI compiler, so it does a better job of type checking and compiles more efficiently if you use ANSI prototypes in your code. Note that mixing ANSI prototypes with older Kernighan and Ritchie–style definitions may result in compiler errors.

Check All Pragmas and Conditional Statements

Most pragmas are compiler specific, so be sure that the pragmas that exist are compatible with MrC. In general, conditional statements should be compiler specific, except for those that check particular aspects of the runtime architecture. See Table 7-1 on page 7-14 for a list of recommended conditional compilation variables.

Make Sure All Code Is 32-Bit Clean

The 24-bit compatibility that was installed for use with the 68000 processor is no longer available.

Address Low-Memory Global Variables Indirectly

Access to low-memory global variables should go through accessor functions defined in the universal header files. This procedure ensures that your code can continue to access them as the Mac OS evolves, even if the variables move to other locations in memory.

Remove or Change Hardware-Dependent Code

Code that depends on specific hardware addresses will probably not run under the PowerPC runtime.

Revise References to the A5 Register

The "global data" TOC pointer in the Table of Contents register (GPR2) is set automatically when a native routine is called. This means that you do not need to save and restore the TOC pointer in the manner of A5 on 68K-based machines. You can conditionalize references to A5 in your code to compile only when building 68K-based applications (that is, classic 68K or CFM-68K). However, if your code sets up a runtime environment for 68K externals, you can use `SetA5` and `Set CurrentA5` to manipulate the emulated A5 register.

Replace Procedure Pointers With Universal Procedure Pointers

PowerPC code must often call classic 68K code running under emulation or vice versa, such as in the case of Macintosh Toolbox callback routines. To make sure the Mixed Mode Manager handles such calls properly, you must change all procedure pointers (that is, direct references to code addresses) in such calls to **universal procedure pointers.**

For example, the following classic 68K runtime function call

```
AEInstallEventHandler (kCoreEventClass, kAEOpenApplication,
                           HandleOapp,0,false);
```

must be changed to

```
UniversalProcPtr myHandleOappProc;
myHandleOappProc = NewAEEventProc (HandleOapp);
AEInstallEventHandler (kCoreEventClass,kAEOpenApplication,
                           myHandleOappProc,0,false)
```

The `NewAEEventProc` macro (defined in `AppleEvents.h`) calls the Mixed Mode Manager's `NewRoutineDescriptor` function to create a routine descriptor for `HandleOapp`.

For more details about the Mixed Mode Manager and routine descriptors, see the chapter "Mixed Mode Manager" in *Inside Macintosh: PowerPC System Software*.

Porting to the CFM-68K Runtime Environment

Because the CFM-68K runtime architecture differs from that of the classic 68K runtime, you may have to modify your classic 68K source programs to get them to run properly.

Many of the changes are very similar to those needed to convert classic 68K programs to the PowerPC architecture, so if you have experience in adapting code for PowerPC (or are porting PowerPC code to CFM-68K), these changes should be familiar and straightforward.

The pascal Keyword

As in the PowerPC runtime environment, Pascal and C calling conventions are identical in the CFM-68K runtime environment. If you declare a function with the `pascal` keyword, the CFM-68K compiler uses the same calling convention as if you had declared it without the `pascal` keyword. Note that the CFM-68K compiler pushes function parameters onto the stack from left to right (which is the reverse of classic 68K Pascal compilers).

A-Line Instructions

CFM-68K runtime code should not directly invoke A-line instructions (often called *A-traps*). Direct calls to the Mac OS must now be routed indirectly using the routines in `InterfaceLib`. If you are using assembly-language source code or C/C++ source code functions that use A-line instructions, the simplest way to revise your code is to make sure your routine names match the standard names declared in the universal header files.

Callback Routines

Code running under the CFM-68K runtime architecture often must call classic 68K runtime code (such as when using the Macintosh Toolbox callback routines), or vice versa. Such calls go through the 68K Mixed Mode Manager, which requires universal procedure pointers instead of procedure pointers. This indirect addressing method is identical to that used in the PowerPC runtime environment. See "Replace Procedure Pointers With Universal Procedure Pointers" on page 7-6 for an example of how to implement universal procedure pointers.

The UnloadSeg Routine

The argument to an `UnloadSeg` statement should never be the address of a routine within a shared library. Applications should never unload shared libraries, because multiple clients may be accessing them. You can still use `UnloadSeg` to unload application segments, however.

Marking Imports and Exports With Pragmas

The SC and SCpp compilers support three pragmas that specify routines or variables to be imported or exported from other code fragments (such as import libraries) during the link procedure. These pragmas are

- `import`, which specifies symbols to be imported from another fragment
- `export`, which specifies symbols that will be exported from the current fragment
- `internal`, which specifies symbols that are referenced only in the current fragment

The SC and SCpp compilers assume that all functions reside in the current fragment, so you must use the `import` pragma to mark any that are cross-fragment (that is, those functions whose definition resides in another fragment, such as a shared library). The compilers also assume that all data items reside in the current fragment. This is different from the PowerPC compilers, which assume that all data items are cross-fragment. Data items defined outside the current fragment must also be declared as such using the `import` pragma.

The compiler creates indirect references to all functions and data declared as imported, which the ILink tool can then resolve. See the CFM-68K runtime architecture information in *Macintosh Runtime Architectures* for more information about how the compiler creates indirect references in the CFM-68K runtime environment.

In many cases, commonly used symbols are already tagged with the appropriate pragmas in Apple's universal header files. If you are making use of standard shared libraries, you should check the header files before adding pragmas to your code.

Note
The `import` and `export` pragmas supersede the older `lib_export` pragma. Although the SC and SCpp compilers currently support `lib_export`, later versions may not, so you should use `import` and `export`. ◆

The import Pragma

The `import` pragma tells the compiler which variables or routines must be imported from other code fragments, thus eliminating possible unresolved reference errors. The syntax of the `import` pragma is as follows:

`#pragma import on|off`

`#pragma import list` ***name1*** `[`***,name2***`]`***...***

In the first syntax form, the compiler tags all symbols defined or declared after `#pragma import on` as imports until it encounters the `#pragma import off` statement. In the second form, you list by name the code and data items you want to have imported.

IMPORTANT
References to imported functions and data items must be indirect. The linker cannot convert a direct reference to an indirect one, so a direct reference to an imported symbol generates a linker error. Therefore, you must use the `import` pragma to define all imported functions and data items. ▲

You should generally use the `import` pragma on symbol declarations rather than definitions. You should put the `pragma import` statement in a public header file so that any routine that needs to can access the imported symbols.

The export Pragma

The `export` pragma tells the compiler which routines or data items will be exported from the fragment being created. For example, shared libraries declare many of their symbols as exported since (by definition) they will be called by routines from other fragments. The syntax of the `export` pragma is as follows:

```
#pragma export on|off

#pragma export list name1 [,name2]...
```

As with the import pragma, you can either declare or define the exported symbols between the `#pragma export on` and `#pragma export off` statements, or list them with `#pragma export list`.

Aside from marking symbols to be exported, the `export` pragma has no effect on the generated code or data. The `export` pragma has no effect on static functions or data items.

Unlike the `import` pragma, you should generally use the `export` pragma on definitions rather than declarations. Also, you should place this pragma in source code files rather than public header files.

The internal Pragma

The `internal` pragma simply limits the scope of the defined functions or data items to the fragment that contains them. Items declared `internal` cannot be exported to, or imported from, other fragments. The syntax for the `internal` pragma is identical to those for the `import` and `export` pragmas.

```
#pragma internal on|off

#pragma internal list name1 [,name2]...
```

You can use this pragma to allow optimizations that might have otherwise violated language semantics. For example, routines declared `internal` do not have an XVector and can use local calling conventions.

Note
The `internal` pragma has no effect on static variables since they are implicitly internal. However, the `internal` pragma allows the compiler to omit the XVectors for a file-scoped routine (since they will never be called via a pointer). You cannot take the address of an `internal` routine because it does not have an XVector. ◆

You must declare internal routines and data items no later than the definition and before you reference them.

The `internal` pragma is well suited for situations where you cannot apply the standard `static` declaration. For example, if you have variables and routines that are referenced from multiple source files (and therefore cannot be declared `static`), you can still ensure optimum code generation by declaring them `internal` in a private header.

You should not use the `internal` pragma in public headers except when applied to variables during an internal build. In that case, you should make the pragma declarations conditional using `#ifdef` statements. A routine declared internal in a public header causes a linker error because the routine is missing an XVector.

Some Pragma Examples

The public header file in Listing 7-1 indicates symbols imported from another (possibly private) routine.

Listing 7-1 A public header `Library.h`

```
#include <IncludeFile.h>

#pragma import on
int mooInt;
void libProc(void);
#pragma import off
```

The private header file in Listing 7-2 exports the symbols `mooInt` and `libProc` for public use, but restricts access of `PrivInt` and `PrivProc` to source files that include `internal.h`.

Listing 7-2 A private header `Internal.h`

```
#include <Library.h>

#pragma export list mooInt, libProc

#pragma internal on
in PrivInt;
void PrivProc(void);
#pragma internal off
```

IMPORTANT

If you are writing C++ code, you should use the `on|off` form of the pragmas to avoid name-mangling issues. The `list` form does not support mangled names or full-function method signatures. ▲

Pragma Compatibility Issues

The `import` and `export` pragmas are completely independent of each other. You can declare symbols as both imported and exported without running into compile or link problems.

The `import` and `internal` pragmas, however, are mutually exclusive. Activating one pragma implicitly deactivates the other, but deactivating one does not implicitly activate the other.

The `export` and `internal` pragmas are independent when used to declare variables, but not functions. Exported routines need an XVector to export, but the `internal` declaration removes this requirement.

Physical Size Limitations in CFM-68K Applications

CFM-68K runtime applications are typically larger than classic 68K runtime applications, but they have the same size limitations as a program compiled with the `-model near` option. Since CFM-68K runtime code can grow larger than the corresponding 68K code due to segment structure, more global data, or more jump table entries, you must be careful to stay within the `-model near` limits. A classic 68K application that approaches the `-model near` size limits may go over the limit when adapted for the CFM-68K runtime architecture. You should note the following when building CFM-68K runtime applications:

- The maximum segment size for a `-model cfmseg` application is 32 KB (the same as `-model near`).
- The code generation model for a CFM-68K runtime application adds a prologue and epilogue to every routine. This overhead is 0 to 6 bytes per routine. In addition, the CFM-68K runtime segment header is 36 bytes larger than the `-model near` segment header.
- Calling routines via function pointers (which is done for all virtual C++ methods) creates an indirect call site (method dispatch) that is 0 to 6 bytes larger than under `-model near`.
- Every imported routine requires a 4-byte XPointer. Every imported data item adds a 4-byte XDataPointer. These pointers reside in the near global data area.
- The maximum jump table size for a `-model cfmseg` application is 4091 entries. The first three jump table entries are reserved for use by the CFM-68K runtime architecture.
- The maximum size of the near global data area for a `-model cfmseg` application is 32 KB (the same as `-model near`).

The 32 KB global data area is sometimes referred to as the *near data area* because the entire 32 KB area can be accessed using a 16-bit offset from A5. Global data items that are accessed only via 32-bit pointers (such as C++ VTables) are considered far data. Far data items can be placed outside the near data space and thus do not affect the near data size.

To reduce the size of the near global data area, you can do the following:

- Use the `-autoimport` compiler option. This option allows you to specify the data addressing mode (16 bit or indirectly through 32-bit pointers) based on the size of the data structure.
- Access large data structures indirectly via 32-bit pointers.
- Allocate data dynamically on the heap.

To increase the available jump table space, you can do the following:

- Reduce the number of segments, which reduces the number of references between segments. This method reduces the number of jump table entries.
- Specify the `-wrap` option when linking. If the jump table size exceeds 32 KB, then this option instructs the linker to place excess jump table entries in the global data area (but only if there is space).

In addition, you can also specify the `-bigseg` compiler option, which removes the 32 KB segment size restriction and allows 32-bit addressing of routines and variables at the expense of slightly larger and slower code. When compiling with the `-bigseg` option, all function calls are encoded with the 68020 `BSR.L` instruction, which is a PC-relative instruction with a 32-bit offset. Applications built with the `-bigseg` option can have a maximum segment size of 2 MB. Shared libraries built with `-bigseg` can have a maximum segment size of 32 MB.

For additional information about the `-autoimport` and `-bigseg` compiler options, see the book *SC/SCpp: C/C++ Compiler for 68K Macintosh.*

Conditional Compilation Variables

Table 7-1 lists several conditional compilation variables you can use when writing your source code. These variables are defined in the `ConditionalMacros.h` header file.

Table 7-1 Conditional compilation variables

Variable name	When set
`GENERATINGPOWERPC`	When compiler generates PowerPC instructions.
`GENERATING68K`	When compiler generates 68K instructions (both classic 68K and CFM-68K runtime).
`GENERATINGCFM`	When generated code uses CFM calling conventions (both CFM-68K and PowerPC runtime).
`CFMSYSTEMCALLS`	When code cannot use A-line instructions. All system calls are made through universal procedure pointers (both CFM-68K and PowerPC runtime).

These variables, when used in conjunction with `#ifdef` or `#ifndef` statements, makes it easy to isolate elements specific to a particular runtime architecture. For example, the following code activates the `options align` pragma only when compiling for the PowerPC runtime architecture.

```
/* Maintain 68K data structure alignment when compiling for PowerPC */
#ifdef GENERATINGPOWERPC
#pragma options align = mac68k
#endif
```

You can use these conditional compilation variables to create source code that can compile under several different runtime architectures. This is especially useful when building fat binary programs. Instead of trying to manage separate files for each runtime architecture, you can maintain just one.

Porting From Other Runtime Environments

Sometimes you may want to port C source code that was developed in a non-Macintosh runtime environment. In such cases, adhering to strict ANSI C standards will minimize the changes needed to run properly. This section lists other issues you should keep in mind when converting simple C programs (SIOW applications, for example).

You should keep the following issues in mind when porting programs to the Macintosh environment:

- Some other environments (such as UNIX) do not have a resource fork.
- Memory is not protected on Macintosh computers. Errors such as writing to or through a `NULL` value, writing past the end of an array, or leaving pointers dangling may cause erratic, unreproduceable behavior or a system crash.
- On PowerPC-based machines, there is a 16 KB limit on the number of 4-byte TOC pointers for data with external linkage (in the ANSI C sense).
- On 68K-based machines compiled under model near, there is a 32 KB limit on the total size of data items.
- On Macintosh computers, the application stack and heap both exist in the same address space. Executing a highly recursive program can cause the two to collide, causing a system crash. The Macintosh does use an interrupt-driven stack sniffer to avoid this, but it may not catch all instances of

stack-heap collisions. You can increase the stack size at startup by using `SetApplLimit`.

- The stack grows downward (toward lower memory) on Macintosh computers.
- On 68K-based machines, you can write to all program code. The code and data items occupy the same address space.
- Macintosh computers have no notion of local and global memory. Near and far pointers have no meaning since pointers are always 32-bit values.
- Macintosh computers have a flat memory architecture instead of a segmented one, so DOS/Windows memory models, small, compact, medium, large, and huge have no meaning.
- Large programs (and MPW tools) written for 68K-based computers can be segmented for better performance. Segments can be unloaded (using `UnloadSeg`) when not in use to save application memory.
- Macintosh compilers swap the definition of the characters `\n` (new line) and `\r` (carriage return) with respect to other operating systems. This switch is important only if you are concerned about the absolute ASCII values of the two characters.

You should also note the following MPW-related issues:

- You must rewrite your makefiles to work with MPW. MPW Make cannot recursively execute or depend on other Make steps.
- MPW tools cannot execute other tools or scripts.
- When specifying pathnames, you must use a colon (`:`) as the directory separator. MPW does not recognize a slash (`/`), either as a separator or to indicate the root directory.
- The directory `/sys` does not exist on Macintosh computers.
- MPW wildcard characters are different from their UNIX counterparts. For example, the asterisk (`*`) does not function in the same manner as in UNIX. See Appendix C in the *MPW Command Reference* for more information.
- UNIX shell scripts must either be rewritten as an MPW script or compiled as an MPW tool.
- MPW does not support the `fork`, `exec`, `system`, `join`, and `setenv` commands. If your UNIX tool uses these commands, you must rewrite it as an MPW script.
- MPW does not comply with POSIX standards.

More About Linking

Contents

The linker combines a group of MPW object files (for example, compiled source code, shared libraries, and static libraries) into a Macintosh program. MPW includes two linkers: PPCLink for PowerPC runtime, ILink for classic 68K and CFM-68K runtime. This chapter expands on some of the concepts presented in Chapter 1 and introduces new ones that surround the link process. These discussions include

- an overview of the link process
- live and dead modules
- resolving references to symbols
- module sorting
- version checking for shared libraries
- linking to third-party libraries
- weak links and stub libraries
- creating initialization and termination routines and designating the main symbol
- how to build your code for debugging
- link maps
- creating static libraries with PPCLink or the Lib tool

Some sections apply only to a particular runtime environment; when this is the case, the name of the environment is included in the heading.

For link troubleshooting information, you can check “Linking Problems,” beginning on page E-3 in Appendix E.

For more information about specific PPCLink or ILink options, see the *MPW Command Reference*.

Note
ILink replaces the Link tool for MPW versions 3.4 or later. ◆

What Happens When You Link

After you have compiled or assembled a source file into an object file, it contains

- object code (relocatable machine language)
- symbolic references to all identifiers whose locations are not known at compile time; these include references to global variables as well as to routines in libraries or other compilation units

When you link your object files together, the linker performs the following functions:

- Resolves all the symbol references. See "Resolving References to Symbols" on page 8-6 for details.
- Omits unused code and data modules from the output file. See the next section, "Live and Dead Modules" on page 8-5, for more details.
- Sorts the code and data modules according to the runtime architecture being used.
- Creates jump table or TOC entries (depending on the runtime architecture being used) when necessary. These entries are used to access cross-fragment or cross-segment calls.
- Edits instructions to use the proper addressing mode. This is mainly used in classic 68K and CFM-68K programs, which support both PC-relative and A5-relative addressing. If desired, you can enforce a certain addressing mode with assembler or compiler options.
- For classic 68K runtime programs, ILink provides support (with the data initialization interpreter) for the initialization of global data at runtime. The data initialization interpreter, `_DataInit`, is located in `MacRuntime.o`. For PowerPC and CFM-68K runtime programs, the Code Fragment Manager handles the initialization of global data.
- Stores the linked code as either data fork fragments or `'CODE'` resources depending on the runtime environment and the type of program.

In addition, for debugging or performance purposes, you can generate a link map that lists the routines and variables used and where they are stored.

Live and Dead Modules

When you link object files together, the linker determines which code and data modules are *live* and which are *dead*. How this is done depends on which runtime architecture you are building for. For classic 68K runtime (the simplest case), the linker begins with the module containing the main entry point (or main symbol) and marks it as live. It then marks all code and data modules referenced by the main module as live. It continues recursively until all the referenced code and data modules are marked as live. For example, 500 modules may be submitted in the link process when only 100 of them will actually be used by the final linked program. The 400 remaining modules contain **dead code** or data that cannot be reached from the main symbol.

When you link a PowerPC or CFM-68K program, the linker uses the main symbol, the initialization routine, the termination routine, and any exported symbols to isolate the live code and data.

By default, ILink writes only the live modules to the output file. PPCLink writes only live modules to the output file in all cases except when building a static library. In static library construction, PPCLink includes all the modules presented for linking in the output file. However, you can override these defaults with the option `-dead`, which specifies whether the dead modules should be excluded from the output file. See the *MPW Command Reference* for details.

Note
When building classic 68K stand-alone code or drivers, you must indicate the main symbol with the `-m` option. This option gives the linker a reference point from which to isolate the live modules. The module you specify with the `-m` option does not have to be the main entry point to your program. That is, the module specified with the `-m` option is only used to determine what modules the linker should include; it is not used to determine which instruction executes first. If you do not specify a main symbol, all of the code and data modules are included in the output file. ◆

Resolving References to Symbols

This section describes how the linker resolves references to symbols.

Symbols in object files are either local or external. Exported symbols in shared libraries are always external.

- A local module or entry point can be referenced only from within the file where it is defined.
- An external module or entry point can be referenced from other files.

An entry point is a location (offset) within a module. (The module itself is treated as an entry point with a zero offset.) A reference is a location within one module that will contain the address of another module or entry point.

The linker first assumes that a symbol is local (that is, if it finds a reference to a symbol, it will first try to match it with a definition in the same file). If the linker cannot find the symbol locally, it looks for it externally (that is, it looks for a definition in the other files).

Multiple External Symbol Definitions

If the object files contain more than one definition for an external symbol, the first definition is used, and all references resolve to that first definition. The linker generates a warning if it encounters subsequent definitions of the same symbol.

You can take advantage of this treatment to override symbol definitions in libraries or object files. For example, if you write a function `mooFunc`, contained in the file `Moo.o`, and the library `Cow.o` also contains a `mooFunc` function, you can have your definition override the library version by giving the file `Moo.o` precedence in the link list:

```
PPCLink  ∂
        mooProg.c.o ∂
        "{myLibs}"Moo.o ∂
        "{OtherLibs}"Cow.o ∂
...
```

This overriding process works for both static and shared libraries. However, libraries can only override symbols in libraries of the same type. That is, a shared library cannot override symbols in a static library and vice versa.

If you are overriding symbols in the PowerPC runtime environment, you should also read “Overriding Entry Points in PowerPC Runtime Programs” on page E-2.

IMPORTANT

If you are building a shared library and wish to allow the overriding of functions in that library, you must export the symbol names by either using the `-shared_lib_export on` option when compiling (for PowerPC runtime) or by marking exports in your source files using `#pragma export` (for CFM-68K runtime). You can also use the `-export` or `-@export` linker option. ▲

▲ **WARNING**

If you override a module, then all entry points within the overridden module disappear. Therefore, you should be sure that any referenced entry points in the overridden module are also defined in the new (overriding) module. ▲

Weak Imports and Libraries (PowerPC and CFM-68K Only)

When using import libraries, you have the option of declaring some imported symbols as weak. A **weak import,** also called a *soft import*, does not have to be present in any of the client’s import libraries at runtime. You designate weak imports using the `-weak` linker option (for both PPCLink and ILink).

You can also indicate a **weak library** with the `-weaklib` linker option. A weak library is an import library that does not have to be present at runtime in order for the client program to run (for example, if your application uses, but does not require, QuickTime).

IMPORTANT

The Code Fragment Manager does not generate any warnings if a weak import or library is not found. Therefore, your program must check for the presence of any weak imports before attempting to use them. ▲

Stub Libraries (PowerPC and CFM-68K Only)

Stub libraries are import libraries that export symbols but do not contain any code. You can use stub libraries at link time to take the place of import libraries needed at runtime (those in ROM or in the Extensions folder). For example, `InterfaceLib` exists on ROM in some machines, so you do not need to ship the library. However, when building your program, you need a file to link against, so you can use a stub version of `InterfaceLib` at link time.

Stub libraries are also useful for resolving circular dependencies. For example, if the library `mooLib` imports symbols from `cowLib` and `cowLib` imports symbols from `mooLib`, then a problem arises: you cannot build `mooLib` without linking with `cowLib` and you cannot build `cowLib` without linking to `mooLib`. The solution is to begin by linking against a stub version of one library. You can build `mooLib` by linking to a stub of `cowLib` (which allows you to resolve imports from `cowLib`), and then you can build the real `cowLib` by linking it to `mooLib`.

Unresolved External Symbols

Occasionally you may find that an external symbol is unresolved because the reference and the definition were generated with different case sensitivity rules. If you are using ILink, you can avoid recompiling by using the `-ma` (module alias) option. When using this option, if the linker encounters an unresolved symbol, it checks the list of module aliases in an attempt to resolve it.

The PPCLink tool does not support the `-ma` option.

Sorting Code Modules

After the linker has isolated the live code modules, it sorts them and writes them to the output file. The way in which the modules are sorted depends on the runtime architecture and (in the case of PPCLink) the linker options used.

You can use the PPCLink option `-codeorder` to specify how the code modules should be sorted. See the *MPW Command Reference* for details.

ILink assigns the live code modules to segments and then sorts the segments according to the following rules:

- If a module is the main entry point, it appears first in the segment.
- Modules that are accessed only via 32-bit references (that is, *far* modules) are sorted after modules accessed via 16-bit references (that is, *near* modules).

- Modules of the same type (that is, all near or all far) are sorted according to the order in which their object files appear on the command line.
- Modules of the same type from the same object file are sorted according to their offset within the object file.

You can specify that certain modules be written to certain segments by using `#pragma segment` in your source code.

IMPORTANT

This sorting algorithm does not apply to the data segment. ▲

Special Symbols

The linker includes several special symbols when you build your Macintosh program. If desired, you can override these with your own symbols.

The Main Symbol

A main symbol is required for applications; for other programs the main symbol is optional. The defaults are as follows:

- for PowerPC runtime, the entry point `__start`, defined in `StdCRuntime.o`
- for classic 68K runtime, the `%__MAIN` routine, defined in `MacRuntime.o`
- for CFM-68K runtime, the `%__MAIN` routine, defined in `NuMacRuntime.o`

The linker does not include the default main symbol when linking a shared library. You can specify your own main symbol with the `-m` linker option.

An application's main symbol must be an executable routine, but a shared library's main symbol (if defined) does not have this restriction.

If you use the `-m` linker option to specify a main symbol for an application, the routine must be declared as

```
void routineName(\void);
```

If you declare parameters to the routine for PowerPC or classic 68K runtime, the parameters will reference undefined data. However, if you declare parameters for CFM-68K runtime, the routine will pop parameters that the system had not passed in, corrupting the stack.

If you use the default main routine from the appropriate runtime library (`StdCRuntime.o`, `MacRuntime.o`, or `NuMacRuntime.o`), your application's main routine may take `argc`, `argv`, and `env` as parameters, or it may take no parameters. The routine may return an `int` (as is traditional in C), but this value is ignored.

For information on restrictions when using a user-defined main routine with MPW libraries, see "Using Standard Libraries With User-Defined Main Symbols," beginning on page 9-11.

Initialization and Termination Routines (PowerPC and CFM-68K Only)

Initialization and termination routines are optional routines you can add when building PowerPC and CFM-68K applications and shared libraries. You can specify either or both by using the `-init` or `-term` linker option.

If you link to `MrCPlusLib.o` (PowerPC) or `NuCPlusLib.o` (CFM-68K), the linker includes default initialization and termination routines that call the C++ static constructors and destructors at the proper time. To make sure that the C++ static constructors and destructors are still executed at the proper time when you add your own routines, you must call the default initialization or termination routines from within your own. Specifically, you must do the following:

- Call the default initialization routine (`__init_app` for applications, `__init_lib` for shared libraries) at the start of your own initialization routine.
- Call the default termination routine (`__term_app` for applications, `__term_lib` for shared libraries) at the end of your own termination routine.

In essence, you should use your own routines only to supplement the default routines, not to replace them. Listing 8-1 shows a sample user-defined initialization routine that calls the default routine `__init_app`.

Listing 8-1 Sample initialization routine for an application

```
OSErr initialize(CFragInitBlockPtr initBlock)
{
    OSErr anErr = noErr;
    anErr = __init_app(initBlock);

    // Your initialization code goes here.
    return (anErr);
}
```

An initialization routine receives a single parameter, a pointer to a `CFragInitBlock` structure. This structure (defined in `CodeFragments.h`) contains a file specification that the code fragment can capture or open to locate its own resource file before making any Resource Manager calls.

IMPORTANT

You must declare the `CFragInitBlockPtr` parameter, even if you do not use it. Keep in mind that when an import library runs, the current resource file is set by the application accessing the library, not by the library itself. An import library must manage resource access for itself, without disturbing its clients' resource chain settings. ▲

An initialization routine must return an `OSErr` result. A result of `noErr` indicates successful execution and any other result indicates failure. A result other than `noErr` causes the entire load to fail and generates an error that is returned to the code that requested the root load (usually the Process Manager for applications).

Termination routines are executed as part of the process of unloading a fragment, and they are executed in the reverse order of the initialization routines. A termination routine has no parameters and returns no result. Listing 8-2 gives an example.

Listing 8-2 Sample termination routine for an application

```
void Terminate (void)
{
    // Your termination code goes here.
    __term_app();
}
```

Import Library Version Checking (PowerPC and CFM-68K Only)

When you execute a program that requires shared libraries, conflicts may occur if the runtime copy (the **implementation version**) of an import library is different from the link-time copy (the **definition version**). For example, say that in order for your application `mooProg` to run, you need the most current version of the import library `mooLib`. Without version checking, if you move `mooProg` over to a computer that contains an older, incompatible copy of `mooLib`, errors will occur when `mooProg` tries to execute.

The Code Fragment Manager relies on version numbers to make sure the import library is compatible with the client application. Both PPCLink and ILink allow you to set version numbers when you build a shared library.

- The `-vercur` option specifies the current version number. This version number is stored in the library itself, as well as in the library's `'cfrg' 0` resource. The default version number is 0.
- The `-verdef` option specifies the oldest link-time (that is, definition) library that is compatible with the runtime library you are currently creating. This version number (default 0) is stored in the library and in the library's `'cfrg' 0` resource.
- The `-verimp` option specifies the oldest runtime (that is, implementation) library that is compatible with the link-time library you are currently creating. This version number is stored in the library but not in the library's `'cfrg' 0` resource. The default is 0.

When you link to a shared library, the linker stores the library's version information in the output file for use in compatibility checks at runtime.

IMPORTANT

XCOFF output files do not contain version information. Preferred Executable Format (PEF) files can contain version information. ▲

Figure 8-1 shows an example of an import library that is linked to an application and then updated in several different ways.

Note
Although the following example uses integers, you should use binary-coded decimal values (as used in the `'vers'` resource) to designate the version numbers of your import libraries. ◆

Figure 8-1 Using import library version options

	`int fun()` `/*bug*/`	`int fun()` `/*bug fix*/`	`int fun()` `/*no change*/` `int new_fun()`	`int new_fun()` `/*no change*/`
Current version (`-vercur`)	1	2	3	4
Link-time (definition) version (`-verdef`)	1	1	1	4
Runtime (implementation) version (`-verimp`)	1	1	3	3

When you create the first version of the library, you do not need to worry about compatibility with previous versions, so you specify

```
PPCLink -o mooLib -vercur 1 -verdef 1 -verimp 1...
```

Now suppose you find a minor bug in the function `fun` in your first version. After fixing the bug, you create version 2. Applications linked with version 1 will run on machines that contain version 2 without having to be updated, because the two versions of `fun` are compatible. Similarly, applications linked with version 2 can still run on machines that contain version 1 (even though it contains a bug). Therefore, when you build version 2 of the import library, you specify

```
PPCLink -o mooLib -vercur 2 -verdef 1 -verimp 1...
```

Now, suppose you update the library again to add a different implementation of function `fun`, called `new_fun`. The definition and implementation for function `fun` remain the same. This becomes version 3 of the import library. Applications linked with either of the older versions will still run on machines that have version 3 because they won't look for the function `new_fun`. However, applications linked with version 3 cannot run on machines containing older versions of the library because they will not be able to find an implementation for the function `new_fun`. Therefore, when you build version 3 of the import library, you specify

```
PPCLink -o mooLib -vercur 3 -verdef 1 -verimp 3...
```

to say that you can run with version 3 if you are linked with an older version but you can't run with anything other than the current version if you link with version 3.

Finally, you remove function `fun` so that only `new_fun` is supported and build version 4 of the import library. Applications linked with older versions of the import library won't run with version 4 because they expect `fun` to be present. However, applications linked with version 4 will run on machines that contain any version that has an implementation for `new_fun` (in this case, version 3 or version 4). Therefore, when you build version 4 of the import library, you specify

```
PPCLink -o mooLib -vercur 4 -verdef 4 -verimp 3...
```

For more information on the PPCLink and ILink tools, see the *MPW Command Reference*. You can also see *Inside Macintosh: PowerPC System Software* for details of how the Code Fragment Manager checks for version information.

About Third-Party Libraries

If you are writing PowerPC or CFM-68K runtime code, you may wish to use shared libraries that are supplied by other software manufacturers. If you need to distribute their shared libraries with your application, you may need to license the library or otherwise obtain permission to use it with your software. Be sure to check the documentation that comes with third-party libraries for licensing information.

Program Segmentation (Classic 68K and CFM-68K Only)

On 68K-based Macintosh computers, segmenting an application or tool makes it possible for temporarily unneeded parts of the program to be unloaded and purged from memory, thus freeing memory. This section explains

- how segments and their corresponding code resources are named
- the creation of segments for the jump table and global data area
- how the ILink linker numbers the code resources

Segment Names and Code Resource Names

You can specify the name of a segment by using the `#pragma segment` directive in your program's source file (see the book *SC/SCpp: C/C++ Compiler for 68K Macintosh* for details). If you don't specify a segment name before the first routine in your file, the segment name `Main` is used by default. You can specify the same segment name more than once and in more than one file. The linker collects code for a given segment name from all of the input files and places it into a single segment in the linked output. This procedure lets you organize your source code for maximum efficiency.

▲ **WARNING**

Segment names are case sensitive. For example, "`Seg1`" and "`SEG1`" are not equivalent names. Trailing spaces are also recognized by the linker; ILink interprets "`Seg1`" and "`Seg1 `" as two different segments. ▲

The linker writes each segment into a separate code resource. By default, these resources are given resource names identical to the corresponding segment names. ILink provides the `-sg` and `-sn` options for combining and renaming segments at link time.

Segments With Special Treatments

When linking a classic 68K application or tool, the linker creates a jump table segment (`'CODE' 0`) that has no name. This segment does not appear in the input object files.

There are also two segments that have special conventions:

- The segment that contains the main entry point (`'CODE' 1`), usually named `Main`.
- A segment named `%A5Init`, which contains the compressed global data image and code that decompresses the data image and sets up the global data area below A5. To save memory, your application should unload this segment before allocating any memory. You can unload the `%A5Init` segment by calling the `UnloadSeg` routine with the address of the entry point `_DataInit` as its parameter.

```
extern void _DataInit
...
void main(void)
{
    UnloadSeg((Ptr)_DataInit);
...
}
```

  This call should be the first statement in the application.

Note
The global data symbols appear in the link map under a segment named `%GlobalData` even though no such segment exists in the linked output. ◆

Numbering 'CODE' resources

ILink automatically assigns `'CODE'` resource numbers to segments in your program. Aside from the `Main` segment, which has ID 1, all other normal program segments are assigned sequential resource numbers beginning with 10. You cannot control the numbering of the segments.

Resource numbers 2 through 9 (that is, the `'CODE' 2` through `'CODE' 9` resources) are reserved for special segments such as the `%_Static_Constructor_Destructor_Pointers` segment.

Linking for Debugging

Debuggers require you to create symbolic information about the object files you are linking. This procedure requires extra options when you compile and link and may also require some file conversion.

When you compile your program for debugging, you must turn on the `-sym` option, so the compiler can generate the proper symbolic information. This option is identical for both the MrC/MrCpp and SC/SCpp compilers:

```
MrC mooProg.c -o mooProg.c.o -sym on
SC mooProg.c -o mooProg.c.o -sym on
```

If you are linking a PowerPC runtime program, you must specify the `-sym on` option when you use PPCLink. The linker then includes debugging information in the output file:

```
PPCLink mooProg.c.o ∂
        -o mooProg ∂
        -sym on ∂
...
```

IMPORTANT

The file containing the symbolic information must be in XCOFF format. If you are building an application or shared library with PPCLink, the output file's default output format is PEF. If you specify `-sym on`, PPCLink creates a file called *outputfile*.`XCOFF` in addition to the normal PEF output. This file contains only symbolic information. If you specify PPCLink's output to be in XCOFF format (using the `-outputformat xcoff` option), then the output file contains the symbolic information (as well as the linked code and data). ▲

The Power Mac Debugger can read XCOFF files directly, and the 68K Mac Debugger reads the ILink state file and its associated object files for symbolic information. However, third-party debuggers require you to create a symbolic information file, or SYM file. To create a SYM file for a PowerPC program, you must run the MakeSYM tool on the XCOFF file:

```
MakeSYM mooProg.XCOFF -o mooProg.xSYM
```

For 68K-based programs, ILink ignores the `-sym` linker option.To create a SYM file, you must run the ILinkToSYM tool on the ILink state file (`mooProg.NJ`):

```
ILinkToSYM mooProg.NJ -o mooProg.SYM
```

IMPORTANT

Although only the ILink state file is indicated on the ILinkToSYM command line, all the object files that were used during the link are also required by the ILinkToSYM tool when creating a SYM file. Therefore, do not move the state file or any of the object files before making a SYM file. ▲

Your third-party debugger can then use the symbolic information contained in the SYM file to aid in debugging.

Link Maps

You can generate a list of your program's symbols and their locations by using the `-map` option. These lists are known as *link maps*, and they are very useful for debugging programs. The format of a link map depends on the runtime architecture. Note that all numerical values are given in hexadecimal.

The PPCLink Link Map (PowerPC Only)

The `-map` option in PPCLink requires you to specify a filename for the link map output.

Listing 8-3 shows a sample PPCLink link map. The map lists the symbols associated with the output file, including TOC entries, global data, glue routines, and code modules.

Listing 8-3 A PPCLink link map

```
===== Link map for c_sample.ppc =====

  Address    Length   CL  TY   Vis   E/M  Symbol Name        Object File

0x00000000 0x0000AC  PR  SD  globl       .main              c_sample1.c.ppc.o
0x000000AC 0x0000D0  PR  SD  globl       .showMessage       c_sample2.c.ppc.o
0x0000017C 0x000044  PR  SD  globl       .myPause           c_sample2.c.ppc.o
0x000001C0 0x0000A0  PR  SD  local       .__start           StdCRuntime.o
0x000001C0           PR  LD  globl       .__start           StdCRuntime.o
0x00000260 0x000018  GL  SD  globl       ._BreakPoint       •••synthesized glue•••
0x00000278 0x000018  GL  SD  globl       .__setjmp          •••synthesized glue•••
0x00000290 0x000018  GL  SD  globl       .InitGraf          •••synthesized glue•••
0x000002A8 0x000018  GL  SD  globl       .InitFonts         •••synthesized glue•••
0x000002C0 0x000018  GL  SD  globl       .exit              •••synthesized glue•••
0x000002D8 0x000018  GL  SD  globl       .SetRect           •••synthesized glue•••
0x000002F0 0x000018  GL  SD  globl       .NewWindow         •••synthesized glue•••
0x00000308 0x000018  GL  SD  globl       .InitWindows       •••synthesized glue•••
0x00000320 0x000018  GL  SD  globl       .InitCursor        •••synthesized glue•••
0x00000338 0x000018  GL  SD  globl       .SetPort           •••synthesized glue•••
0x00000350 0x000018  GL  SD  globl       .StringWidth       •••synthesized glue•••
0x00000368 0x000018  GL  SD  globl       .MoveTo            •••synthesized glue•••
0x00000380 0x000018  GL  SD  globl       .TextFont          •••synthesized glue•••
0x00000398 0x000018  GL  SD  globl       .DrawString        •••synthesized glue•••
0x000003B0 0x000018  GL  SD  globl       .Button            •••synthesized glue•••
0x000003C8 0x000018  GL  SD  globl       .SystemTask        •••synthesized glue•••

0x0000003C 0x000004  TC  SD  local       _IntEnv            StdCRuntime.o
0x00000040 0x000004  TC  SD  local       __C_phase          StdCRuntime.o
0x00000044 0x000004  TC  SD  local       __target_for_exit  StdCRuntime.o
0x00000048 0x000004  TC  SD  local       _exit_status       StdCRuntime.o
0x00000050 0x000004  TC  SD  local       __NubAt3           StdCRuntime.o
0x00000054 0x000004  TC  SD  local       .stringBase0       c_sample1.c.ppc.o
0x00000058 0x000004  TC  SD  local       .TMP0              c_sample2.c.ppc.o
0x0000005C 0x000004  TC  SD  local       <no symbol>        StdCRuntime.o
0x00000060 0x000004  TC  SD  local       gRect              c_sample1.c.ppc.o
0x00000064 0x000004  TC  SD  local       qd                 c_sample1.c.ppc.o
0x00000068 0x000004  TC  SD  local       gWindow            c_sample1.c.ppc.o
0x0000006C 0x000000  TO  SD  local       TOC                StdCRuntime.o
0x0000006C 0x00000C  DS  SD  globl    M  __start            StdCRuntime.o
```

```
0x00000078 0x000018  RO  SD  local       .stringBase0       c_sample1.c.ppc.o
0x00000090 0x000027  RW  SD  local       .TMP0              c_sample2.c.ppc.o
0x000000B8 0x000008  RO  SD  local       <no symbol>        StdCRuntime.o
0x000000C0 0x000008  BS  CM  local       gRect              c_sample1.c.ppc.o
0x000000C8 0x0000CE  BS  CM  local       qd                 c_sample1.c.ppc.o
0x00000198 0x000004  BS  CM  local       gWindow            c_sample1.c.ppc.o
```

The fields in the link map are as follows:

- `Address` lists the symbol's absolute address.
- `Length` specifies the length of the data.
- `CL` indicates the symbol's storage mapping class as shown in Table 8-1.
- `TY` indicates the symbol's type, as shown in Table 8-2.
- `Vis` indicates the visibility (or scope) of the symbol (local or global).
- `E/M` indicates whether the symbol is exported (`E`) or the main entry point (`M`).
- `Symbol Name` indicates the name of the symbol.
- `Object File` specifies the file that contains the symbol definition.

Table 8-1 Storage mapping classes

Class	Description
BS	Zero-initialized data
DB	Debug dictionary table
DS	Transition vector
GL	Cross-TOC glue code
PR	Program code
RO	Read-only constants
RW	Read/write data
SV	Supervisor call descriptor
TB	Traceback table
TC	TOC entry

continued

Table 8-1 Storage mapping classes (continued)

Class	Description
`TO`	TOC anchor
`TD`	Scalar data entry in the TOC
`TI`	Traceback index
`UA`	Unclassified
`UC`	Unnamed FORTRAN common
`XO`	Extended operation

Table 8-2 Symbol types

Type	Description
`CM`	Common
`ER`	External reference
`HL`	Hidden label
`LD`	Label definition
`SD`	Section definition
`US`	Uninitialized storage

The ILink Link Maps (Classic 68K and CFM-68K Only)

The link map output is automatically directed to standard output when you use the `-map` option.

The Classic 68K Link Map

The first element of the classic 68K link map is the jump table map, as shown in Listing 8-4.

Listing 8-4 A classic 68K jump table map

```
Jump table segment 'CODE'(0) ""
Size = 0x0040

-- Segment Main --
0x0020  +0x000C  %__MAIN

-- Segment Utils --
0x0028  +0x000C  showMessage
0x0030  +0x0098  myPause

-- Segment %A5Init --
0x0038  +0x000C  _DataInit
```

Jump table entries are grouped by segment, with the fields defined as follows:

- The first field is the A5 offset of the jump table entry.
- The second field is the segment offset of the code module or entry point.
- The third field is the name of the code module or entry point.

For example, in Listing 8-4, the jump table entry for module `%__MAIN` resides at A5+0x0020. The module `%__MAIN` itself resides at offset 0x000C in the code segment named `Main`.

Next, each code segment is described (there are three in this example), as shown in Listing 8-5.

Listing 8-5 A classic 68K segment map

```
Code segment 'CODE'(1) "Main"
Size = 0x0AF4
# JT Entries = 0x0001
0x000C  0x0026  JT=0x0020  extern  %__MAIN             MacRuntime.o
0x0032  0x0054             extern  main                c_sample1.c.o
0x0086  0x013B             extern  __CplusInit         MacRuntime.o
0x01C2  0x00FE             local   GetAppName          MacRuntime.o
0x02C0  0x017C             extern  _GetProgramGlobals  MacRuntime.o
0x043C  0x0134             extern  _RTInit             MacRuntime.o
0x0570  0x0060             extern  _RTExit             MacRuntime.o
```

```
0x05D0  0x0056              extern  _memcpy             MacRuntime.o
0x0626  0x004C              extern  __NUMTOOLBOXTRAPS   MacRuntime.o
0x0672  0x0038              extern  __GETTRAPTYPE       MacRuntime.o
0x06AA  0x0092              extern  TRAPAVAILABLE       MacRuntime.o
0x073C  0x0078              extern  ___MAIN             MacRuntime.o
0x07B4  0x0026              extern  __setjmp            MacRuntime.o
0x07DA  0x0038              extern  longjmp             MacRuntime.o
0x0812  0x0052              extern  _DoExitProcs        MacRuntime.o
0x0864  0x006C              extern  _zero               MacRuntime.o
0x08D0  0x015C              extern  _INTENVINIT         MacRuntime.o
0x0A2C  0x0054              extern  _INTENVTERM         MacRuntime.o
0x0A80  0x000E              extern  GETHANDLESIZE       Interface.o
0x0A8E  0x0014              extern  NGETTRAPADDRESS     Interface.o
0x0AA2  0x0036              extern  C2PSTR              Interface.o
0x0AD8  0x001C              extern  P2CSTR              Interface.o

Code segment 'CODE'(10) "Utils"
Size = 0x00A6
# JT Entries = 0x0002
0x000C  0x008C  JT=0x0028  extern  showMessage  c_sample2.c.o
0x0098  0x000E  JT=0x0030  extern  myPause      c_sample2.c.o

Code segment 'CODE'(11) "%A5Init"
Size = 0x01BA
# JT Entries = 0x0001
0x000C  0x005A              extern  _DATAINIT         MacRuntime.o
0x000C          JT=0x0038   extern  _DataInit         MacRuntime.o
0x0066  0x0060              local   uncompress_world  MacRuntime.o
0x00C6  0x0054              local   get_rl            MacRuntime.o
0x011A  0x005C              local   relocate_world    MacRuntime.o
0x0176  0x0042              local   ZEROBUFFER        MacRuntime.o
0x01B8  0x0002              local   #0001             MacRuntime.o
0x01BA                      extern  _A5Init3          MacRuntime.o
```

The fields under each code segment (from left to right) are as follows:

- the segment offset of the module or entry point
- the size of the module
- the A5 offset of the module's or entry point's jump table entry, if it exists (`JT=`)

- the scope of the module or entry point (local or external)
- the name of the module or entry point
- the name of the object file that contains the module or entry point definition

The next section lists all the global data, as shown in Listing 8-6.

Listing 8-6 A classic 68K global data map

```
Data segment %GlobalData
Size = 0x0296
-0x00DE  0x00DE  extern  _SAGlbls                    MacRuntime.o
-0x00E2  0x0004  extern  StandAlone                  MacRuntime.o
-0x0116  0x0034  extern  _IntEnv                     MacRuntime.o
-0x0156          extern  __target_for_exit           MacRuntime.o
-0x0156  0x0040  local   #0002                       MacRuntime.o
-0x015A  0x0004  extern  _macpgminfo                 MacRuntime.o
-0x015E  0x0004  extern  _IsStandAlone               MacRuntime.o
-0x0162  0x0004  local   __TableEntries              MacRuntime.o
-0x0164  0x0002  local   __myResFile                 MacRuntime.o
-0x0168  0x0004  extern  _#_IntEnv                   MacRuntime.o
-0x016C          extern  _EnvP                       MacRuntime.o
-0x0170          extern  _ArgV                       MacRuntime.o
-0x0174          extern  _ArgC                       MacRuntime.o
-0x0178          extern  __TheExitAddr               MacRuntime.o
-0x0178  0x0010  local   #0001                       MacRuntime.o
-0x01A0  0x0027  local   _TMP0                       c_sample2.c.o
-0x01B8  0x0017  local   %?Anon                      c_sample1.c.o
-0x0286  0x00CE  local   qd                          c_sample1.c.o
-0x028E  0x0008  local   gRect                       c_sample1.c.o
-0x0292  0x0004  local   gWindow                     c_sample1.c.o
-0x0296  0x0004  local   _ArgvLBracketArgcRBracket   MacRuntime.o  *
```

The fields (from left to right) are as follows:

- the A5 offset of the variable
- the size of the variable
- the scope of the variable (local or external)

- the name of the variable
- the name of the object file that contains the variable definition

An asterisk (*) at the end of an entry (the `_ArgvLBracketArgcRBracket` variable in this list) means that the variable is always accessed through 32-bit pointers and can therefore be more than 32 KB away from A5.

The CFM-68K Link Map

The link map for a CFM-68K fragment adds several new elements specific to the CFM-68K runtime environment. These additions include XVectors, XPointers, and XDataPointers, as well as the names of imported and exported symbols.

Note the following CFM-68K–specific prefixes that identify pointers and entry points associated with code and data modules:

- `_%` indicates an XVector. For example, the XVector pointing to the code module `mooProc` would be `_%mooProc`.
- `_@` indicates an XPointer. An XPointer for the XVector to `mooProc` would be `_@mooProc`.
- `_#` indicates an XDataPointer. The XDataPointer for the data module `mooData` would be `_#mooData`.
- `_$` indicates the internal entry point of a code module, which is the entry point for direct (that is, in-fragment) calls. For example, a direct call to `mooProc` would enter at `_$mooProc`. See *Macintosh Runtime Architectures* for more information about internal and external entry points.

Listing 8-7 displays the jump table map.

Listing 8-7 A CFM-68K jump table map

```
Jump table segment 'CODE'(0) ""
Size = 0x0048

-- Segment Utils --
0x0038  +0x0032  _$showMessage
0x0040  +0x00FA  _$myPause
```

The format of this listing is identical to that for the classic 68K runtime environment:

- The first field is the A5 offset of the jump table entry.
- The second field is the segment offset of the code module or entry point.
- The third field is the name of the code module or entry point.

Next, in Listing 8-8, each code segment is described (there are two in this example). This is similar to the classic 68K runtime version except space has been added to indicate XVector entries.

Listing 8-8 A CFM-68K code segment map

```
Code segment 'CODE'(1) "Main"
Size = 0x0698
# JT Entries = 0x0000
# XV Entries = 0x0004
0x0030  0x0022             XV=0x0048  extern  %__MAIN               NuMacRuntime.o
0x0058  0x0094                        extern  main                  c_sample1.c.o
0x005A                                extern  _$main                c_sample1.c.o
0x00F0  0x003C                        local   GetAppName            NuMacRuntime.o
...
0x04D0  0x008E                        extern  TrapAvailable         NuMacRuntime.o
0x04D2                                extern  _$TrapAvailable       NuMacRuntime.o
0x0560  0x008C                        extern  ___MAIN               NuMacRuntime.o
0x0562                                extern  _$___MAIN             NuMacRuntime.o
0x05F0  0x0044             XV=0x006C  extern  _memcpy               NuMacRuntime.o
0x0638  0x0026             XV=0x0060  extern  __setjmp              NuMacRuntime.o
0x0660  0x0038             XV=0x0054  extern  longjmp               NuMacRuntime.o

Code segment 'CODE'(10) "Utils"
Size = 0x0128
# JT Entries = 0x0002
# XV Entries = 0x0000
0x0030  0x00C2                        extern  showMessage    c_sample2.c.o
0x0032          JT=0x0038             extern  _$showMessage  c_sample2.c.o
0x00F8  0x0030                        extern  myPause        c_sample2.c.o
0x00FA          JT=0x0040             extern  _$myPause      c_sample2.c.o
```

The fields under each code segment (from left to right) are as follows:

- the segment offset of the module or entry point
- the size of the module
- the A5 offset of the jump table entry, if it exists (`JT=`)
- the A5 offset of the module's or entry point's XVector, if it exists (`XV=`)
- the scope of the module (local or external)
- the name of the module or entry point
- the name of the object file that contains the module or entry point definition

Note
In the CFM-68K runtime, the `'rseg' 1` resource handles the duties of the classic 68K runtime's `%A5Init` segment. See *Macintosh Runtime Architectures* for more details. ◆

The next section, as shown in Listing 8-9, displays the global data map.

Listing 8-9 A CFM-68K global data map

```
Data segment %GlobalData
Size = 0x01D0
...
-0x0038  0x0004  extern  _@DrawString      <none>
-0x003C  0x0004  extern  _@C2PStr          <none>
-0x0040  0x0004  extern  _@TextFont        <none>
-0x0044  0x0004  extern  _@ExitToShell     <none>
-0x0048  0x0004  extern  _@NewWindow       <none>
-0x004C  0x0004  extern  _@InitGraf        <none>
-0x0050  0x0004  extern  _@_memcpy         NuMacRuntime.o
-0x0054  0x0004  local   _#qd              c_sample1.c.o
-0x0058  0x0004  extern  _@__setjmp        <none>
-0x005C  0x0004  extern  _@longjmp         <none>
...
-0x00FC  0x0008  local   gRect             c_sample1.c.o
-0x0100  0x0004  local   gWindow           c_sample1.c.o
-0x01D0  0x00CE  local   qd                c_sample1.c.o  *
```

The fields (from left to right) are identical to those for the classic 68K version:

- the A5 offset of the variable
- the size of the variable
- the scope of the variable (local, external, or exported)
- the name of the variable
- the name of the object file that contains the variable definition

In the global data map, a `<none>` listing (for the object file containing the variable definition) indicates that the variable is a linker-generated XPointer.

The next section, shown in Listing 8-10, displays the XVector map.

Listing 8-10 An XVector map (CFM-68K only)

```
XVector segment %XVectors
Size = 0x003C
0x0048  0x000C  extern  _%%__MAIN   NuMacRuntime.o
0x0054  0x000C  extern  _%longjmp   NuMacRuntime.o
0x0060  0x000C  extern  _%__setjmp  NuMacRuntime.o
0x006C  0x000C  extern  _%_memcpy   NuMacRuntime.o
0x0078  0x000C  export  _%myPause   c_sample2.c.o
```

The entries for each XVector (from left to right) are as follows:

- the A5 offset of the XVector
- the size of the XVector
- the scope of the XVector (external or exported)
- the XVector name
- the name of the object file containing the XVector definition

Finally, all imported symbols are listed, as shown in Listing 8-11.

Listing 8-11 An imported symbols map (CFM-68K only)

```
Imported Symbols
TVect        _%InitGraf        InterfaceLib  InterfaceLib
TVect        _%NewWindow       InterfaceLib  InterfaceLib
TVect        _%ExitToShell     InterfaceLib  InterfaceLib
...
TVect        _%MoveTo          InterfaceLib  InterfaceLib
TVect        _%NGetTrapAddress InterfaceLib  InterfaceLib
TVect        _%Button          InterfaceLib  InterfaceLib
TVect        _%P2CStr          InterfaceLib  InterfaceLib
Data         _SAGlbls          NuIntEnv      StdCLib
```

The entries for each symbol (from left to right) are as follows:

- the type of symbol
- the name of the symbol
- the name of the shared library fragment that exports the symbol
- the name of the file that contains the shared library

Note that the `TVect` (transition vector) symbol is equivalent to an XVector.

Optional Map Formats for Compatibility

The `-l` and `-la` options for ILink produce a link map in an obsolete format. This format has been retained only for compatibility with MPW performance tools that read the map files to determine module locations. You should not create tools that depend on the format of the link map since it is likely to change in the future.

If your tools need information about module locations, they should read symbolic information (SYM) files.

Optimizing Your Links

Learning to link efficiently can reduce the time required to build your program. You can check Technical Note 313, *MPW Performance Issues*, for specific details. In general, however, the following steps can speed up the performance of the linker tools:

- Use a disk (RAM) cache.

  The linker must open and close many object files, so a disk cache can improve performance. You should experiment with cache sizes to determine what is optimum for your environment. Use the Memory control panel to check your disk cache settings. (If you change the setting, you must restart your Macintosh computer before the new setting takes effect.)

- Use a RAM disk.

  Storing your object files on a RAM disk minimizes access time and, consequently, the link time. Use the Memory control panel to allocate memory for a RAM disk. (You must restart your Macintosh computer to implement or change the size of the disk.)

- Turn off virtual memory.

  If you are linking on a PowerPC-based machine, time spent paging code into memory from the hard drive can add substantially to the link time. Turning virtual memory off will eliminate this slowdown (at the expense of somewhat larger memory requirements).

- Build static libraries with PPCLink or Lib.

  See the next section, “Static Library Construction,” for details.

- Eliminate unneeded files.

  You should analyze your source and include files so you can give the linker only the files it needs to link.

- Eliminate unneeded routines.

  The linker will not include unneeded routines in the output file, but determining which routines are unnecessary adds to the link time.

Static Library Construction

You can build your own static libraries by combining object code from different files and languages into a single object file. For example, you can combine assembly-language code with C or Pascal or combine multiple object code files produced by a particular compiler. The PPCLink tool creates static libraries when you specify the `-xm l` option, but you must use the Lib tool when building static libraries for the classic 68K and CFM-68K runtime environments.

Why You Should Create Static Libraries

Object files combined into libraries may result in faster links than the raw object files produced by the compiler. There are several reasons for the speed improvement:

- There are fewer files to read from disk.
- When several object files are combined, multiple instances of a symbol definition are replaced with a single definition. This reduces the size of the file and simplifies processing by the linker.

Choosing Files for a Specialized Library

The best files to combine in a specialized library are those that are unlikely to change over many builds. Stable files include the library files provided by Apple for the ROM interfaces and for language support. Files currently under development should be left as single files.

Building Libraries With PPCLink (PowerPC Only)

The PPCLink tool allows you to build static libraries by merely specifying the `-xm l` option (as opposed to `-xm e` for applications and `-xm s` for shared libraries). For example, to build the library `mooLib.o` from the files `Cow1.o` and `Cow2.o`, you can specify

```
PPCLink Cow1.o Cow2.o -xm l -o mooLib.o
```

The output file `mooLib.o` has a file type of `'XCOF'`, and you can specify a creator if desired (using the `-c` option).

Building Libraries Using Lib (Classic 68K and CFM-68K Only)

To create static libraries for 68K-based Macintosh computers, you must use the Lib tool. For example, to build the library `mooLib.o` from the files `68kCow1.o` and `68kCow2.o`, you can specify

```
Lib 68kCow1.o 68kCow2.o -o mooLib.o
```

In addition to combining files into a library, you can also use Lib to

- change the segmentation (using the `-sg` and `-sn` options)
- change the scope of a symbol from external to local (using the `-dn` option)
- delete unneeded modules (using the `-dm` or `-df` option)
- rename modules (using the `-rn` option)
- discard symbolic information (using the `-sym off` option)

The DumpObj tool may be useful for exploring the content and structure of a static library. For more details about the Lib and DumpObj tools, see the *MPW Command Reference*.

IMPORTANT

You must use Lib and DumpObj versions 3.4 or later if you are using CFM-68K object files. ▲

Standard Libraries

Contents

This chapter gives an overview of the various libraries available in the MPW environment, how they vary according to runtime environment, and related build information.

Note
When discussing libraries, a name in computer voice, for example, "`MathLib`," refers to the specific runtime environment library, while "MathLib" implies the entire class of libraries with the same functionality. ◆

Library Folders

The MPW standard libraries are located in several different folders, divided by type of library and runtime architecture:

- SharedLibraries, which contains fat shared libraries for both PowerPC and CFM-68K runtime environments. These files are in PEF format.
- PPCLibraries, which contains static libraries for the PowerPC runtime environment (XCOFF format).
- CLibraries, which contains C- and C++-specific static libraries for the classic 68K runtime environment.
- Libraries, which contains all the other classic 68K runtime libraries.
- CFM68KLibraries, which contains static libraries for the CFM-68K runtime environment.

When writing makefiles, you should always indicate the library pathname in your command line, either directly or (preferably) by using the MPW Shell variables. For example, the default value for the `{Libraries}` variable is the path to Libraries, so you could put `MacRuntime.o` in your command line by specifying

```
"{Libraries}"MacRuntime.o
```

MPW Libraries

This section discusses each of the standard MPW libraries, the variations depending on the runtime architecture, and the routines they contain.

CPlusLib (and IOStreams)

The C++ libraries are divided into two files for each runtime/compiler combination. The IOStreams libraries contain the C++ I/O Streams support functions and their associated static objects. The CPlusLib libraries contain the remaining C++ support functions, such as the function `new`, compiler support routines, and runtime initialization and termination routines (CPlusLib libraries do not contain any static objects).

The C++ library files are as follows:

- `MrCPlusLib.o` and `MrCIOStreams.o` for PowerPC runtime (using the MrCpp compiler).
- `CPlusLib.o` and `IOStreams.o` for classic 68K runtime (using the SCpp compiler).
- `IOStreams881.o` for classic 68K runtime with MC68881/68882 support. Note that `CPlusLib.o` does not contain floating-point objects and can be used with MC68881/68882 support.
- `NuCPlusLib.o` and `NuIOStreams.o` for CFM-68K runtime (using the SCpp compiler).

Note that the following dependencies apply when linking these libraries:

- If you link with an IOStreams library, you must also link with the corresponding CPlusLib library.
- If you link with a CPlusLib library, you must also link with the corresponding StdCLib library.
- `IOStreams881.o` replaces `IOStreams.o` when building programs that require the MC68881/68882 math coprocessor.

InterfaceLib

The InterfaceLib libraries provide access to the Macintosh Toolbox. Note that in the case of the PowerPC and CFM-68K runtime environments, you cannot make A-line instruction calls from within your program but must call the Toolbox indirectly through `InterfaceLib`.

The InterfaceLib libraries are as follows:

- `InterfaceLib` for PowerPC and CFM-68K runtime (the library is a fat binary file)
- `Interface.o` for classic 68K runtime

IntEnv.o (Classic 68K Only)

This library contains various low-level I/O and utility routines that were originally contained in `StdCLib.o`, `PasLib.o` (for Pascal support), and `Runtime.o`. If your classic 68K program requires modules from `stdio.h` or `fcntl.h`, you must link with `IntEnv.o`.

IMPORTANT

Many of the routines in `IntEnv.o` use global variables, so you should be careful when using this library to build stand-alone code. ▲

Note

In the PowerPC and CFM-68K runtime environments, equivalent routines to those in `IntEnv.o` are contained in `StdCLib`. ◆

MathLib

The MathLib libraries contain implementations of standard C mathematical routines as defined in the universal headers `fp.h`, `fenv.h`, and `float.h`.

The libraries are as follows:

- `MathLib` for PowerPC runtime (shared library).
- `MathLib.o` for classic 68K runtime.

- `MathLib881.o` for classic 68K runtime using the 68881/68882 math coprocessor (that is, compiled with the `-mc68881` option).
- `NuMathLib.o` for CFM-68K runtime. Note that you cannot use a math coprocessor when using CFM-68K runtime programs.

See *Inside Macintosh: PowerPC Numerics* for a listing of functions contained within the MathLib libraries.

The MathLib libraries supersede the older SANE libraries (`CSANELib.o` and so on), encompassing most of the routines and adding some new ones. The universal header `fp.h` is a superset of `math.h` and also includes all the routines in `sane.h`. A new header `fenv.h` contains all the environmental controls.

Note that there have been some name and parameter order changes between the existing 68K-based SANE functions and their MathLib counterparts. For example, the SANE function `expl` is now `expml` in MathLib, and the `copysign(x,y)` function is now `copysign(y,x)`.

The libraries `MathLib.o` and `NuMathLib.o` do not conform to the FPCE (Floating-Point C Extensions) standards used by the PowerPC-based `MathLib` library. Where the SANE standard and the FPCE standard differ, the 68K-based versions match SANE (and violate the new standard). For example, the function `pow(0,0)` returns 1 on PowerPC-based machines (conforming to FPCE), but returns `NaN` ("not a number") on 68K-based machines.

Also, `MathLib.o` and `NuMathLib.o` interpret types `double_t` and `float_t` to be the same as type `extended`. The `Types.h` header file contains the type definitions for the various runtime environments. Note that both `float_t` and `double_t` are defined in `Types.h` as type `long double` when not compiling for the PowerPC runtime environment.

MathLib libraries do not contain functions to replace the SANE routines for halt-handling (a hardware-dependent capability) and precision control. If you need these functions, you must use the SANE libraries.

For more detailed information about differences between the PowerPC numerics standard (FPCE) and SANE, see *Inside Macintosh: PowerPC Numerics.*

RTLib (Classic 68K and CFM-68K Only)

The RTLib libraries contain routines you can use to patch nonstandard Segment Loader routines. That is, if you need to control the Segment Loader routines in the classic 68K `-model far` environment or the CFM-68K runtime environment (both of which have a segment structure different from the standard `-model near` version), you must use RTLib.

The RTLib libraries are as follows:

- `RTLib.o` for classic 68K runtime (`-model far` only)
- `NuRTLib.o` for CFM-68K runtime

Both libraries require the header file `RTLib.h`.

For detailed information about using RTLib, see *Macintosh Runtime Architectures.*

Runtime

The Runtime libraries contain routines and data specific to each runtime architecture, including application initialization and termination routines. In addition, `PPCCRuntime.o` contains support routines for the MrC and MrCpp compilers. You should always include the corresponding library when building an application.

The libraries are as follows:

- `PPCCRuntime.o` and `StdCRuntime.o` for PowerPC runtime
- `MacRuntime.o` for classic 68K runtime
- `NuMacRuntime.o` for CFM-68K runtime

SIOW

These libraries contain the routines that define the Simple Input/Output Window (SIOW) package. They provide a simple event loop and routines for reading input from, and writing output to, a pseudo-console window. Other routines redirect I/O from standard files (for example, `stdin`, `stdout`, and `stderr` in C) to the console window.

The SIOW libraries are as follows:

- `PPCSIOW.o` for PowerPC runtime
- `SIOW.o` for classic 68K runtime
- `NuSIOW.o` for CFM-68K runtime

IMPORTANT

`PPCSIOW.o` must replace `StdCRuntime.o` in the command line when building a PowerPC runtime SIOW application. In classic 68K runtime, `SIOW.o` must precede `MacRuntime.o` in the command line. In the CFM-68K runtime, `NuSIOW.o` must precede `NuMacRuntime.o`. ▲

For more information about SIOW applications, see Chapter 15, "Building SIOW Applications."

StdCLib

The StdCLib libraries contain most of the ANSI-compliant routines of the standard C library. They also contain some Apple extensions to the standard C library and some compiler support routines.

The StdCLib libraries are as follows:

- `StdCLib` for PowerPC and CFM-68K runtimes (the library is a fat binary file)
- `StdClib.o` for classic 68K runtime

Note that the `StdCLib` library for the PowerPC and CFM-68K runtime environment actually combines the functionality of the classic 68K libraries `StdClib.o` and `IntEnv.o`.

IMPORTANT

The classic 68K library `StdCLib.o` uses global variables, so you should be extremely careful if you want to use this library for stand-alone code. ▲

For specific information about the standard ANSI C routines, see the *MPW Standard C Library Reference*.

Both the PowerPC and CFM-68K runtimes require INIT files to enable the `StdCLib` library, so you must distribute these with your application. For the PowerPC runtime, this file is called `StdCLibInit`. For the CFM-68K runtime, the `StdCLib` implementation is contained within the file `CFM-68K Runtime Enabler`. These files should be placed in the Extensions folder.

Stubs.o (Classic 68K Only)

You should use the `Stubs.o` library when building an MPW tool to reduce the size of the code. This library contains dummy library routines that override standard library routines not used by MPW tools. `Stubs.o` must be the first library in your command line.

ToolLibs

The ToolLibs libraries contain a variety of routines that you can use when building an MPW tool. They may also be useful for applications.

The ToolLibs libraries are as follows:

- `PPCToolLibs.o` for PowerPC runtime.
- `ToolLibs.o` for classic 68K runtime.
- `NuToolLibs.o` for CFM-68K runtime. This library is useful for applications only, since you cannot build a CFM-68K MPW tool.

Table 9-1 shows the ToolLibs components, the headers required, and the runtime environments that support each component.

Table 9-1 ToolLibs components

Component	Required header	PowerPC support?	Classic 68K support?	CFM-68K support?
Cursor control	`CursorCtl.h`	Yes	Yes	Yes
68K Disassembler	`DisAsmLookup.h`	No	Yes	No
Error Manager	`ErrMgr.h`	Yes	Yes	Yes
MC68000 Test	`MC68000Test.h`	No	Yes	Yes
Unmangle	`Unmangler.h`	Yes	Yes	Yes
PowerPC Disassembler	`Disassembler.h`	Yes	Yes	Yes

Other Libraries

Various Macintosh extensions and technologies require their own libraries (for example, `QuickTimeLib` and `AppleScriptLib`), which are not documented here. You should check the specific extension documentation for build, usage, distribution, and licensing information.

Switching Between Libraries

Table 9-2 lists the equivalent standard libraries for each runtime architecture. When you port to another architecture, you should use the corresponding set of libraries.

Table 9-2 Switching between MPW libraries

Library type	PowerPC	Classic 68K	CFM-68K
Toolbox, OS	`InterfaceLib`	`Interface.o`	`InterfaceLib`
C	`StdCLib`	`StdCLib.o`	`StdCLib`
C utilities, low-level I/O	`StdCLib`	`IntEnv.o`	`StdCLib`
	N.A.	`CLib881.o`	N.A.
C++	`MrCPlusLib.o`	`CPlusLib.o`	`NuCPlusLib.o`
C++ I/O streams	`MrCIOStreams.o`	`IOStreams.o`	`NuIOStreams.o`
SIOW	`PPCSIOW.o`	`SIOW.o`	`NuSIOW.o`
MPW tools	`PPCToolLibs.o`	`ToolLibs.o` and `Stubs.o`	`NuToolLibs.o`
Runtime support	`PPPCRuntime.o` and `StdCRuntime.o`	`MacRuntime.o`	`NuMacRuntime.o`
68K segment manipulation support	N.A.	`RTLib.o`	`NuRTLib.o`

continued

Table 9-2 Switching between MPW libraries (continued)

Library type	PowerPC	Classic 68K	CFM-68K
Numerics	`MathLib`	`MathLib.o`	`NuMathLib.o`
	N.A.	`MathLib881.o`	N.A.
Pascal support	N.A.	`PasLib.o`	N.A.
Object Pascal support	N.A.	`ObjLib.o`	N.A.

Note that a single library in one architecture may be split into two in another architecture. Also, some classic 68K libraries make no sense in other architectures (for example, `-model far` support and the xxx`881.o` math coprocessor libraries), so you may need to make some code changes to adapt to these differences.

Using Standard Libraries With User-Defined Main Symbols

In general, you do not need to worry about library initialization and termination procedures. However, if you override the default main symbol or provide your own shared library initialization or termination routines, you must be aware of certain constraints that arise depending on the runtime architecture you use and the libraries you link with.

Library initialization and termination procedures come in two categories: general runtime and library specific. General runtime procedures are handled by code in both the Runtime libraries and the SIOW libraries. Library-specific procedures are handled by code in each library as necessary.

Note
In the PowerPC and CFM-68K runtime environments, library-specific initialization and termination take place in the standard initialization/termination routines. (See "Initialization and Termination Routines (PowerPC and CFM-68K Only)," beginning on page 8-10.) ◆

The _RTInit routine

The `_RTInit` routine handles the general runtime initialization procedures, and it is called by either the `StdCLib` initialization routine (in PowerPC runtime) or the default main symbol (for classic 68K and CFM-68K runtime).

The `_RTInit` routine, which is described in Chapter 13, "Writing and Building MPW Tools," does the following:

- Allows access to the program parameters `argc`, `argv`, and `envp`.
- Determines whether the variable definitions (if any) pointed to by the environment pointer are set up as C or Pascal strings.
- Initializes internal library structures and global variables.

In the classic 68K runtime, `_RTInit` also performs the following:

- Calls the C++ static constructors (if they exist).
- Installs an exit procedure to ensure that the C++ static destructors (if they exist) are called on termination.
- Allocates and initializes structures used by the `IntEnv` and `StdClib` libraries.

Main Symbols in PowerPC Runtime

If you define your own main symbol in a PowerPC runtime program, you should note the following:

- If you are writing an MPW tool, you must use the default main entry point, `__start`. Also, you must not call `ExitToShell` to exit, as this causes both your tool and the MPW Shell to terminate.
- You do not need to call `_RTInit`.
- You must exit your application using the Toolbox call `ExitToShell`, either directly or through one of the Integrated Environment routines or ANSI C Library exit routines such as `exit`, `atexit`, `abort`, or `_RTExit`.

Main Symbols in Classic 68K Runtime

If you define your own main symbol in a classic 68K runtime program, you should be aware of the following:

- In the classic 68K runtime, you must call `_RTInit` explicitly in your main routine.
- You must terminate your application with a call to one of the Integrated Environment routines such as `exit`, `_exit`, or `_RTExit`. See Chapter 13, "Writing and Building MPW Tools," for more information about Integrated Environment routines. Note that use of these routines is not restricted to MPW tools.
- For optimum performance, you should make sure your stack is aligned on a longword boundary.

Main Symbols in CFM-68K Runtime

If you define your own main symbol in a CFM-68K runtime program, you should be aware of the following:

- You should call `_RTInit` if you plan to use any routines in the `IntEnv`, `StdCLib`, or `PasLib` libraries, or if you are writing an MPW tool.
- You must exit your program using the Toolbox call `ExitToShell` (same as for PowerPC runtime).

Make and Makefiles

Contents

If you are working on a large program—application, driver, or tool—you are likely to need to build the program many times during development, even though only a few of the input files might have changed at any one time. Make is a tool that you can use to minimize the time required to rebuild the program. Using information you provide in a text file called a makefile, Make determines what commands need to be executed to rebuild only those files that have changed between builds and the files that depend on those changed files.

Following an introduction that explains what makefiles are and how they are used, this chapter discusses

- the format of a makefile
- the use of Make variables
- the use of default rules in makefiles
- examples of makefiles

If you have never worked with makefiles before, you should first read the sections “Introduction to Makefiles” on page 10-4, “The Format of a Makefile” on page 10-6, and “Example 1—Creating a Makefile” on page 10-35. Then read “Default Dependency Rules” on page 10-22 and “Variables in Makefiles” on page 10-30 and see whether you can follow one of the harder examples discussed at the end of this chapter: “Example 2—A Makefile Using Modified Default Rules,” “Example 3—Make’s Makefile,” or “Example 4—Multiple Folders and Multiple Makefiles.”

If you have worked with makefiles in MPW before and consider yourself fairly adept in the art, you should read through the sections describing the features added since the release of MPW version 3.0. These are described in “Forcing a Target to Be Rebuilt” on page 10-20, “Using Make to Build Multiple Makefiles” on page 10-21, “Specifying Secondary Dependencies” on page 10-27, and “Built-in Make Variables” on page 10-31. Note also that the built-in default rules for Make now include a rule for C++ builds; this rule is shown in Listing 10-5 on page 10-23.

For help debugging your makefile, see “Debugging Makefiles,” beginning on page E-1 in Appendix E.

Introduction to Makefiles

This section defines the basic terms used in describing your program files from the Make tool's point of view: target file, prerequisite file, and root. It also discusses

- methods for creating makefiles
- naming conventions for makefiles
- Make options
- executing the output of the Make command

Basic Terms

The Make tool determines what commands need to be executed to build or rebuild a program by analyzing the information provided in a makefile; this makefile is the input file to the Make command.

A **makefile** is a text file that describes dependency information for one or more target files. A **target file** is a file that needs to be built; it depends on one or more **prerequisite files** that must exist or be brought up-to-date before the target can be built. For example, an application depends on its source file or files, a number of library files, and resource files. If any of these prerequisite files are newer than the target application file, then the application needs to be rebuilt.

The prerequisite files of a target file might themselves be targets with their own prerequisites, and so on. For example, object files depend on source files and header files; thus an object file is a prerequisite of the target program as well as the target of its prerequisite source and interface files. A target that is not a prerequisite of any other target is called a **root.** A makefile may have one or more roots.

Creating a Makefile

You can create a makefile in two ways:

- Choose the Create Build Commands item from the Build menu. The CreateMake Commando dialog box appears, and you can use it to specify the kind of program you're trying to build and the source files that your program depends on. Using this information, CreateMake creates a makefile that includes dependency information for the source files you specified and the appropriate build commands for the type of program you're building (application, SIOW, driver, stand-alone code).
- Write a makefile from scratch by using the MPW Shell editor. This file would include both the rules that define the dependencies among your source files and the build commands required to build files that are new or that have changed.

The CreateMake method of writing a makefile is easier than writing a makefile from scratch. However, because CreateMake does not know how to include dependencies on header files or library files, the method most commonly used is to have CreateMake set up the makefile and then to edit the makefile to include dependencies on header files, library files, and the makefile itself.

To edit a makefile created by CreateMake or to write a makefile from scratch, you must understand the format of the makefile and the syntax used to provide dependency information. These are described in the section "The Format of a Makefile," which begins on page 10-6.

Makefile Naming Conventions

By convention, makefiles have the suffix `.make`. CreateMake uses this convention when it creates a makefile; it assigns the name *program*`.make` to the makefile, where *program* is the name you type in the Program Name box of the CreateMake dialog box.

You specify the makefile that serves as input to Make by using the `-f` option. For example, if you enter the following command, Make uses the makefile `MyInit.make` as its input file:

```
Make MyInit -f MyInit.make
```

If you do not use the `-f` option to specify a makefile, Make looks for a file named `MakeFile` in the current directory.

The Syntax of the Make Command

The syntax of the Make command, which invokes the Make tool, is

```
Make [target1] [target2]...[-d name [=value]] [-e][-f makeFile] [-p]
        [-r [targetFile] ] [-s] [-t] [-u] [-v] [-w] [-y]
```

For complete information about the Make command, including descriptions of Make options, see the *MPW Command Reference*.

Executing Make's Output

The Make tool reads each dependency rule in your makefile and determines whether the target file exists or whether it is older than its prerequisite. If either of these conditions is met, Make outputs the command to build the target file to the active Shell window. To rebuild the program, you must execute the commands output by Make. You can automatically execute the Make command output by calling Make from a Shell script. The simplest form of such a script consists of the following two commands:

```
Make {"Parameters"} > MakeOut
MakeOut
```

The first command executes `Make`, using the parameters passed to the script. Output (that is, build commands) is redirected to the file `MakeOut`. The second line of the script executes `MakeOut`.

The Format of a Makefile

A makefile is a text file that contains the information Make requires to determine which of your program's component files have changed since the last build and how to rebuild files whose prerequisite files have changed. The main syntactic elements of a makefile are comments, variable definitions, and dependency rules. Listing 10-1 shows a very simple makefile. Makefiles used to build large programs are much more complicated than the one shown, but despite its simplicity, the following makefile uses all the elements found in more complex makefiles and is easier to take in at a glance. More realistic examples of makefiles are included at the end of this chapter.

Note
Makefile physical input lines may not exceed 255 characters when using Make versions 3.1 or earlier. Logical input lines (made up of one or more physical input lines continued with the continuation character ∂) may be of any length. ◆

Listing 10-1 `Sample.make` makefile

```
# This is a simple makefile used to build the Sample program.
# The source files and makefile for this program are found in
# the {CExamples} folder. This is a comment.

MyLibs =    "{Libraries}"MacRuntime.o ∂
            "{Libraries}"IntEnv.o ∂
            "{Libraries}"Interface.o ∂
            "{CLibraries}"StdCLib.o             #see note 1

Sample ƒƒ Sample.r                              #see note 2
    Rez Sample.r -append -o Sample              #see note 3

Sample ƒƒ Sample.c.o                            #see note 4
    ILink -t APPL -c 'MOOF' ∂
    Sample.c.o ∂
    {MyLibs} ∂
    -o Sample

Sample.c.o ƒ Sample.c                           #see note 5
    SC Sample.c                                 #see note 6
```

The following notes explain how Make interprets the contents of the `Sample.make` makefile.

1. The text on the right side of the equation replaces any occurrence of `{MyLibs}` in the makefile.
2. `Sample` depends on `Sample.r` and also depends on some other file The double-ƒ dependency signifier (ƒƒ) implies that `Sample` can be brought up to date in more than one way. A build command that is different from the `Rez` command will resolve the additional dependency.

3. If `Sample` does not exist or if `Sample.r` is newer than `Sample`, the `Rez` command needs to be output.
4. `Sample` depends on `Sample.c.o` and also depends on some other file (the double-ƒ dependency signifier (ƒƒ) implies that `Sample` can be brought up to date in more than one way). A build command that is different from the `ILink` command will resolve the additional dependency. If `Sample` does not exist or if `Sample.c.o` is newer than `Sample`, the `ILink` command needs to be output.
5. `Sample.c.o` depends on (ƒ) `Sample.c`.
6. If `Sample.c.o` does not exist or if `Sample.c` is newer than `Sample.c.o`, the `SC` compile command needs to be output.

Dependency Rules

The Make tool determines what commands need to be executed to build a program by evaluating the dependency rules included in a makefile. A **dependency rule** consists of a **dependency line,** which specifies the prerequisite file or files of a given target file, followed by a **build rule,** the command or commands needed for building the target file. The Make tool outputs this command or commands to standard output or to a specified output file in one of two cases: if the target file does not exist or if the target file is older than any one of its prerequisite files.

You can use single-ƒ dependency rules or double-ƒ dependency rules to express dependency information. Their syntax and meaning are explained in the next two sections.

Single-ƒ Dependency Rules

The syntax of a single-ƒ dependency rule is

targetFile... ƒ [*prerequisiteFile...*]
        [*ShellCommand…*]

The first line of a dependency rule is the dependency line. The following line or lines contain the build rule and must begin with a tab or a space.

The makefile shown in Listing 10-1 contains three dependency rules. The last rule is an example of a single-ƒ dependency rule:

```
Sample.c.o ƒ Sample.c
    SC Sample.c
```

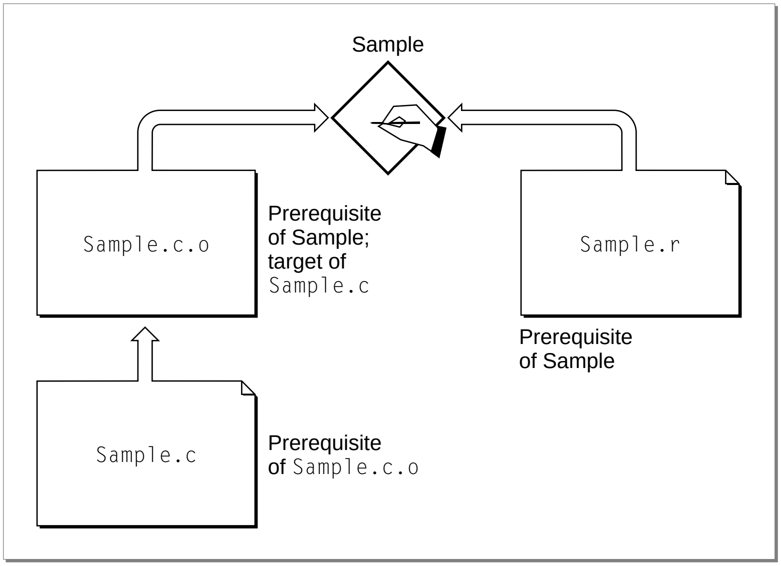
The dependency line of this rule states that the target file `Sample.c.o` depends on the prerequisite file `Sample.c`. The ƒ character (Option-F) means *depends on*. Make checks whether `Sample.c.o` exists. If it does not, Make outputs the command

```
SC Sample.c
```

If the file already exists, Make looks at the modification date of `Sample.c` (and its prerequisites, if any), and if it finds that the file is newer than `Sample.c.o`, it outputs the `SC` command.

Because a target's prerequisites can themselves be targets with their own prerequisites, Make investigates prerequisites in a recursive, bottom-up fashion. Figure 10-1 shows the dependencies among the files needed to build the `Sample` application. Make evaluates the dependency of `Sample.c.o` on `Sample.c` before evaluating the dependency of `Sample` on `Sample.c.o`.

Figure 10-1 Target files and prerequisite files



You can use the `-s` option to the Make command to display a dependency-tree graph for a specified target. Make uses indentation to indicate levels in the dependency tree. The following Make command

```
Make -s -f Sample.make
```

displays the following dependency-tree graph for the file `Sample.make`.

```
Sample
    Sample.r
    Sample.c.o
        Sample.c
```

Double-ƒ Dependency Rules

Double-ƒ dependency rules are a common source of confusion for the writers of makefiles; therefore, their definition, which is short, follows a lengthier introduction that explains why double-ƒ rules are needed.

In writing the dependency rule for the target file `Sample.c.o`, you could add more prerequisite files on the dependency line; for example, you could make `Sample.c.o` depend on the interface file `types.h`.

```
Sample.c.o ƒ Sample.c types.h
    SC Sample.c
```

If the interface file `types.h` were updated, then the target file `Sample.c.o` would need to be rebuilt. In this case, the same SC build command would be used no matter which of the prerequisite files had changed. But what about a case that is very typical for Macintosh programs, where the target file depends on two prerequisite files, each dependency requiring its own build command? For example, the `Sample` file depends on both the resource file `Sample.r` and the object file `Sample.c.o`, but a different build command is required in each case: a Rez command for the dependency on the `Sample.r` file and an ILink command for the dependency on the `Sample.c.o` file.

Conceivably, you could signal this situation to Make by writing the following dependency rules:

```
#BAD MAKEFILE!!!
MyLibs = "{Libraries}"MacRuntime.o ∂
         "{Libraries}"IntEnv.o ∂
         "{Libraries}"Interface.o ∂
         "{CLibraries}"StdCLib.o

Sample ƒ Sample.r
    Rez Sample.r -append -o Sample

Sample ƒ Sample.c.o
    ILink -t APPL -c moos ∂
    {MyLibs}       ∂
    Sample.c.o ∂
    -o Sample
```

However, when you execute the command

```
Make Sample -f Sample.make
```

to generate the required build commands, Make discovers that the makefile is giving two different build rules for building the same target file and returns the message:

```
### make - "Sample" has more than one set of build rules
File "sample.make"; Line 66
### make - Fatal error(s) in dependency file
```

A double-ƒ dependency rule tells Make that a target file *does* depend on more than one prerequisite file and that each of these dependencies requires its own build command.

The syntax of a double-ƒ dependency rule is the same as that of a single-ƒ dependency rule, except that a double ƒ is used instead of a single ƒ:

targetFile... ƒƒ [*prerequisiteFile...*]
        [*ShellCommand…*]

As in the single-ƒ dependency rule, the line or lines following the dependency line contain the build rule and must begin with a tab or a space.

Thus, the correct way to express that `Sample` depends on `Sample.r` and `Sample.c.o` is as follows:

```
MyLibs = "{Libraries}"MacRuntime.o ∂
         "{Libraries}"IntEnv.o ∂
         "{Libraries}"Interface.o ∂
         "{CLibraries}"StdCLib.o

Sample ƒƒ Sample.r
                Rez Sample.r -append -o mySample

Sample ƒƒ Sample.c.o
    ILink -t APPL -c moos ∂
    {MyLibs} ∂
    Sample.c.o ∂
    -o Sample
```

In this instance, if `Sample.r` is newer than the target file `Sample`, Make outputs the command that compiles `Sample.r` and appends the resource file to `Sample`. If `Sample.c.o` is newer than `Sample`, Make outputs the command that links `Sample.c.o` and places the `'CODE'` resources in `Sample`. If both prerequisite files are newer than `Sample`, Make outputs both the Rez and the ILink commands.

The same file can appear as a prerequisite in more than one double-ƒ rule, as in this example:

```
TargetFile  ƒƒ  A  B  D
    ShellCommands-1

TargetFile  ƒƒ  C  D
    ShellCommands-2
```

In this case, Make outputs both sets of build commands if `TargetFile` is out-of-date with respect to `A` and `C`, but only one set of build commands if `TargetFile` is out-of-date with respect to `A`. If `TargetFile` is out-of-date with respect to `D`, then both sets of build commands are output because `D` appears as a prerequisite in both dependency rules. One common example of this situation is when `D` is the name of the makefile itself. Including the makefile as a prerequisite of the program file ensures that the program is completely rebuilt when the makefile changes.

Note
You can omit the build commands from a double-ƒ rule if they are supplied by default rules. If build commands are left out of more than one double-ƒ rule for the same target, Make applies the default rules only to the first empty set. See "Default Dependency Rules" on page 10-22 for information on the Make tool's built-in dependency rules. ◆

Using Variables in Makefiles

Variables are used in makefiles to eliminate unnecessary typing and to create more general makefiles. You can use variables to specify the directory portion of filenames or to specify lists of files, such as library files, utility files, or object files.

You can define variables on the Make command line using the `-d` option or within the makefile itself. If you define a variable in the makefile and then redefine it using the `-d` option, the definition specified with the `-d` option overrides the definition in the makefile.

The syntax of a variable definition is

variableName = *stringValue*

Subsequent appearances of {*variableName*} in the makefile are replaced by *stringValue*. In the following example, the definition of `{MyLibs}` includes all library files, eliminating the need to specify each of them in the dependency line or in the `ILink` command.

```
MyLibs = "{Libraries}"MacRuntime.o ∂            # variable definition
         "{Libraries}"IntEnv.o ∂
         "{Libraries}"Interface.o ∂
         "{CLibraries}"StdCLib.o

Sample ƒƒ Sample.e
    Rez Sample.r -append -o Sample

Sample ƒƒ Sample.c.o            {MyLibs}        # use of variable
    ILink -t APPL -c moos ∂
    Sample.c.o      ∂
    {MyLibs}        ∂                           # use of variable
    -o Sample
```

```
Sample.c.o ƒ Sample.c
    SC Sample.c
```

For additional information about the use of variables in makefiles, see "Variables in Makefiles" on page 10-30.

Comments

The number sign (#) indicates a comment. Everything from the # to the end of the line is ignored. Comments always end at the next return, even if the return is preceded by the line continuation character ∂. Here is the correct way to continue a comment:

```
A ƒ B                                   #   this comment is continued
                                        #   correctly
```

The following comment is not continued correctly:

```
A ƒ B                                   #   this comment is continued ∂
                                            incorrectly
```

You can use comments in dependency lines, variable definitions, and build command lines, or on lines by themselves. Comments in build command lines are passed through to standard output, where they are processed as comments by the MPW Shell.

Using Quotation Marks in Makefiles

Make supports several of the MPW Shell's quoting conventions. Quoted items can appear in dependency lines, variable definition lines, and build command lines. The following quotation characters are used:

∂	Quotes the subsequent character; that is, the ∂ is removed and the subsequent character is taken to be a literal character (except when you type ∂ followed by Return at the end of a line to continue the line).
'...'	Quotes the enclosed string. The single quotation marks are removed.
"..."	Quotes the enclosed string, but {...} variable references are expanded, and the escape character ∂ is processed. The double quotation marks are removed.

Quotation characters are processed as follows:

- In dependency lines and in the name part of variable definitions, quotation literalizes the quoted characters (useful for expanding file or variable names).
- On the right side of variable definitions, quoted items are passed through "as is" so that the quoting takes effect when the variable is expanded.
- In build command lines, quoted items are passed through "as is" so that the quoting takes effect when the build commands are executed by the Shell.

Note
When expanding the built-in variables `{Targ}`, `{NewerDeps}`, `{Deps}`, `{TargDir}`, `{DepDir}`, and `{Default}` in build commands, Make automatically quotes their values, if necessary, because they will represent filenames or parts of pathnames. Don't quote them yourself. ◆

The Order in Which Make Builds Targets

The important orderings within a makefile are the first target mentioned (the default top-level target) and the order of prerequisite files for any given target. Otherwise, the order in which targets are mentioned is not important.

You can specify the top-level target (or targets) on the Make command line. If you do not specify a target on the command line, then Make builds the first target appearing on the left side of a dependency rule in the makefile (that is, the default top-level target).

Make builds the top-level target and its prerequisites, which may also be targets, in bottom-up order, starting with the first prerequisite in the target's prerequisite list. After the first prerequisite (and its own prerequisites) have been investigated, the target's next prerequisite is investigated. This prerequisite is the next one mentioned in the current dependency rule or in the next dependency rule that has the same left-side target.

Once a target has been investigated by Make, it is not revisited, even if it appears somewhere else among the top-level target's prerequisites. In other words, while a file can appear as a prerequisite of a number of target files, Make rebuilds it only once (if necessary) when it is first encountered in the recursive bottom-up traversal of the dependency hierarchy (prerequisite files to the top-level target).

Remember that a makefile can have one or more top-level targets (or roots); that is, it can describe how to build more than one object. You can use the Make command's `-r` option to identify all the roots of a makefile. You can use the `-s` option to display the order of all files from the top-level target to the most distant prerequisite.

Using Dependency Rules

As shown by the syntax diagrams for dependency rules, it is possible to specify multiple targets or multiple prerequisites in a dependency line and to omit build commands. This section explains how Make interprets such rules and how they might be useful to you. If you have not written makefiles before, you should skip this section and return to it after reading the sections "Variables in Makefiles," beginning on page 10-30, and "Default Dependency Rules," beginning on page 10-22.

Dependencies on Include Files, Libraries, and the Makefile

Whether you are creating a makefile from scratch or editing a makefile created by CreateMake, you should add dependencies on your include files, libraries, and on the makefile itself. Listing 10-2 shows how these might be included for the makefile shown in Listing 10-1 on page 10-7.

Listing 10-2 `Sample.make` including all dependencies

```
CObjs   = Sample.c.o ∂
        "{Libraries}"MacRuntime.o ∂
        "{Libraries}"IntEnv.o ∂
        "{Libraries}"Interface.o ∂
        "{CLibraries}"StdCLib.o

Sample      ƒƒ {CObjs} Sample.make
    ILink -o {Targ} {PObjs}
    SetFile {Targ} -t APPL -c 'MOOF' -a B
```

```
Sample      ƒƒ Sample.r Sample.make
        Rez -rd -o {Targ} Sample.r -append

Sample.c.o      ƒ Sample.make Sample.h
```

The variable `{Targ}` is assigned by Make; its assigned value is the complete filename of the target on the left side of the dependency rule whose build commands are being processed. For additional information, see "Built-in Make Variables" on page 10-31.

Omitting Build Commands

If you omit build commands for a dependency line, Make takes the build commands from one of the target's other dependency rules, or from default rules if no build rules were specified for the target. For example, if your makefile includes the following rules:

```
MyApp.c.o ƒ MyApp.c
    SC MyApp.c

MyApp.c.o ƒ MyApp.h
```

Make would use the command `SC MyApp.c` to rebuild `MyApp.c.o` if either `MyApp.c` or `MyApp.h` were newer than `MyApp.c.o`.

Several Single-ƒ Dependencies for the Same Target

If you specify more than one single-ƒ dependency line for a target, then the target depends on all the prerequisite names on all the lines. However, you must make sure that you specify the build rule only once (as indicated in the previous section).

Several Targets for a Single-ƒ Rule

More than one target filename can appear on the left side of a single-ƒ rule. Each target file on the left side depends on all of the files listed on the right side (and has the same build commands, if specified). Specifying more than one target file is exactly the same as if you had specified a separate dependency rule for each target. In this case, the built-in Make variable `{Targ}` has the value of the current target specified on the Make command line or the value of the target Make assumes by default.

In the following example, the target that is built depends on the value of the `{Targ}` variable:

```
version1.c.o version2.c.o ƒ stuff.h

version1.c.o ƒ version1.c
    SC {Targ}

version2.c.o ƒ version2.c
    SC {Targ}
```

The following Make command would display the commands required to build `version2.c.o`:

```
Make version2.c.o -f MakeFile
```

Typically, you'll have more than one target on the left side of a single-ƒ rule only when expressing dependencies, so you won't include any build rules. If you do supply build rules, you must write them in a generic fashion by using the `{Targ}` variable because each target is built independently. Contrary to what the syntax might suggest, multiple targets on the left side of a single-ƒ rule do not imply that Make builds all targets by a single application of the build rules. Therefore, you cannot use this construct to express dependencies in which a tool has more than one output file.

Abstract Targets

An **abstract target** is a target that is not actually built but represents a collection of items. A dependency rule with an abstract target has no build rules, just a dependency line; the target on the left side of the single-ƒ rule does not exist. It serves merely to trigger the dependencies for the prerequisite files on the right side of the single-ƒ rule. Thus, if your makefile includes several different roots that could be built, say, A, B, and C, the following dependency line

```
All ƒ A B C
```

would output the commands required to build `A`, `B`, and `C` when you execute the command

```
Make All -f MakeFile
```

Makefiles With Multiple Targets

Although makefiles are commonly used to build programs, a makefile can include multiple targets. Here is a trivial but clear example. Your makefile, `testme`, contains the following dependency rules:

```
beep1 ƒ
    beep

beep2 ƒ
    beep
    beep
```

If you execute the output of this command

```
Make beep1 -f testme
```

you hear one beep. If you execute the output of the command

```
Make beep2 -f testme
```

you hear two beeps.

You can write a makefile with multiple targets to build different versions of a program, each version being a target. Or you can take advantage of all the useful information often found in a makefile to define utility targets. For example, in the makefile shown in Listing 10-3, `{CObjs}` is defined as a list of the object files used by the program `Sample`.

Listing 10-3 Using utility targets

```
CObjs       = Sample.c.o                             # object files
MyLibs      =   "{Libraries}"MacRuntime.o ∂
                "{Libraries}"IntEnv.o ∂
                "{Libraries}"Interface.o ∂
                "{CLibraries}"StdCLib.o
```

```
Sample      ƒƒ {CObjs}
    ILink -o {Targ} {CObjs} {MyLibs}
    SetFile {Targ} -t APPL -c 'MOOF' -a B

Sample      ƒƒ Sample.r
    Rez -rd -o {Targ} Sample.r -append

Sample.c.o      ƒ Sample.c
    SC Sample.c
```

You can include the following dependency rule in the makefile:

```
MyObjects ƒ
    DumpObj {CObjs}
```

Then, if you execute the command

```
Make MyObjects -f sample.make
```

Make would display the command

```
DumpObj Sample.c.o
```

Forcing a Target to Be Rebuilt

Make recognizes the predefined `$OutOfDate` target as an artificial target that is always out-of-date. You can force a target to be rebuilt by making it depend on `$OutOfDate`. This procedure is illustrated by the following example.

Suppose that your makefile includes the following variable definition and dependency rule:

```
ForceRebuild =          #default variable definition (NOP)
.
.
.
SomeTarget ƒ {ForceRebuild}
    buildSomeTargetCommands
```

If you invoke Make without defining the `{ForceRebuild}` variable in the Make command line, `SomeTarget` is not rebuilt. `SomeTarget` is rebuilt if you invoke Make with the following command:

```
Make -d ForceRebuild=$OutOfDate
```

The redefinition of `{ForceRebuild}` in this Make command line overrides the definition of `{ForceRebuild}` in the makefile; making `SomeTarget` depend on `$OutOfDate` forces `SomeTarget` to be rebuilt.

Using Make to Build Multiple Makefiles

In addition to building programs, you can introduce a dependency to have a makefile generate another makefile.

Listing 10-4 shows a fragment of a makefile `Sample.make` with dependencies for building both PowerPC and classic 68K versions of `Sample`.

Listing 10-4 Using two-step makefiles

```
...
# Build the PowerPC version.
Sample.PPC ƒƒ {PPCObjects} {PPCLibs} Sample.make
    PPCLink ∂
        Sample.o ∂
        {PPCObjects} ∂
        {PPCLibs} ∂
        -fragname mooCowPPC ∂
        -o Sample.PPC

Sample.PPC  ƒƒ Sample.make Sample.r
    Rez Sample.r -append -o Sample.PPC


# Build the classic 68K version.
Sample.68k ƒƒ {68KObjects} {68KLibs} Sample.make
    ILink  ∂
        {68KObjects} ∂
        {68KLibs}∂
        -o Sample.68k
```

```
Sample.68k ƒƒ Sample.make Sample.r
    Rez Sample.r -append -o Sample.68k
...
```

You can include a new dependency rule in `Sample.make` that creates a Make command rather than a series of build commands:

```
PPCVers     ƒ $OutOfDate
        Make -f Sample.make Sample.PPC > SamplePPC.make
```

If you execute the command

```
Make -f Sample.make PPCVers
```

Make displays the command

```
Make -f Sample.make SamplePPC > SamplePPC.make
```

which, if executed, creates the file `SamplePPC.make`, which contains all the build commands needed to build the PowerPC version of `Sample`. While seemingly trivial in this case, when used with default dependency rules (described in the next section), two-step makefiles are very useful for managing large programs. See “Example 4—Multiple Folders and Multiple Makefiles,” beginning on page 10-48, for a detailed example of the two-step makefile.

Default Dependency Rules

In addition to the dependency rules specified in your makefile, Make also uses built-in default dependency rules in evaluating makefiles. The purpose of these rules is to express many specific dependencies and build commands by a single rule, thereby eliminating the need to write these out in your makefile. So, understanding what these rules are and how they work will save you time in the long run.

Default dependency rules look like regular dependency rules except that they express a dependency between two files whose names are the same but whose suffixes differ—for example:

```
SpellCheck.c.o ƒ SpellCheck.o
MyGame.c.o ƒ MyGame.o
```

Default rules are either built-in or defined by you. This section explains the format and use of built-in default dependency rules.

Built-in default dependency rules, along with related variable definitions, are stored in the Make tool's data fork and are shown in Listing 10-5.

Listing 10-5 Make default rules and variable definitions

```
# 68K build default rules
.a.o        ƒ   .a
        {Asm} {depDir}{default}.a -o {targDir}{default}.a.o {AOptions}

.c.o        ƒ   .c
        {C} {depDir}{default}.c -o {targDir}{default}.c.o {COptions}

.p.o        ƒ   .p
        {Pascal} {depDir}{default}.p -o {targDir}{default}.p.o {POptions}

.cp.o       ƒ   .cp
        {CPlus} {depDir}{default}.cp -o {targDir}{default}.cp.o {CPlusOptions}

.cpp.o      ƒ   .cp
        {CPlus} {depDir}{default}.cpp -o {targDir}{default}.cpp.o {CPlusOptions}


# PowerPC build default rules
.s.x        ƒ   .s
        {PPCAsm} {depDir}{default}.s -o {targDir}{default}.s.x {PPCAOptions}

.c.x        ƒ   .c
        {PPCC} {depDir}{default}.c -o {targDir}{default}.c.x {PPCCOptions}

.cp.x       ƒ   .cp
        {PPCCPlus} {depDir}{default}.cp -o {targDir}{default}.cp.x {PPCCPlusOptions}

.cpp.x      ƒ   .cpp
        {PPCCPlus} {depDir}{default}.cpp -o {targDir}{default}.cpp.x {PPCCPlusOptions}
```

```
Asm                         =           Asm
C                           =           SC
Pascal                      =           Pascal
CPlus                       =           SCpp
PPCAsm                      =           PPCAsm
PPCC                        =           MrC
PPCCPlus                    =           MrCpp

AOptions                    =
COptions                    =
POptions                    =
CPlusOptions                =
PPCAOptions                 =
PPCCOptions                 =
PPCCPlusOptions             =
```

Note
To make use of built-in default dependency rules, your files must use the file-naming conventions described on page 2-11 for PowerPC runtime, page 3-7 for classic 68K runtime, and page 4-12 for CFM-68K runtime. ◆

How Default Rules Work

The syntax of a default rule is

.[*suffix1*] *ƒ* .*suffix2* [*secondaryDependency*...]
ShellCommand...

A default dependency rule works like the dependency rules you have looked at so far, except that the specified dependency is between filename suffixes. The *secondaryDependency* parameter can be more filename suffixes or filenames. (For information about secondary dependencies, see “Specifying Secondary Dependencies” on page 10-27.)

The Make default dependency rules shown in Listing 10-5 on page 10-23 use this syntax to define dependencies between object files and source files written in MPW C, MPW C++, MPW Assembler, and MPW Pascal—for example:

```
.c.o    ƒ   .c
    {C} {DepDir}{Default}.c -o {TargDir}{Default}.c.o {COptions}
```

This rule states that if any file with a suffix `.c.o` depends on a file of the same name with the suffix `.c`, the subsequent build rule should be used to build the `.c.o` file. The variables used in the build rule are described in the next section. In practical terms, this means that the following dependency rule would be covered by the preceding default rule and you would not need to include it in your makefile:

```
MyFile.c.o ƒ MyFile.c
    SC MyFile.c -o MyFile.c.o
```

Your makefile might include a dependency rule like the following:

```
MyFile ƒƒ MyFile.c.o {OtherObjects}
    ILink MyFile.c.o {OtherObjects}
```

This rule would give Make sufficient information so it could apply the default rule to compile the `c.o` file. However, you would still need to include dependencies on header files—for example:

```
MyFile.c.o ƒ MyFile.h
```

You might also want to include a definition for compiler or assembler options you want to use for the build—for example:

```
COptions = -model far
```

Variables Used in Built-in Default Rules

The build commands for the default rules shown in Listing 10-5 on page 10-23 make use of certain variables. Some of these variables are already set.

```
Asm          =     Asm      # MPW Assembler
C            =     SC       # SC
CPlus        =     SCpp     # SCpp
Pascal       =     Pascal   # MPW Pascal
PPCAsm       =     PPCAsm   # PowerPC Assembler
PPCC         =     MrC      # MrC
PPCCPlus     =     MrCpp    # MrCpp
```

Thus, the variable `{PPCC}` in the default build rule is expanded to `MrC`, the command that invokes MPW's PowerPC compiler, when the makefile is executed. You can override these variables by defining them in your makefile to specify another MPW-compatible compiler. For example, you could redefine `Cplus` to refer to MrCpp, the PowerPC compiler, rather than SCpp.

The compile option variables (`{AOptions}`, `{COptions}`, and so on) are initially null; you can set these to any value or values you would want to specify as assembler or compiler options. You can then use these variables in place of the actual options (such as in default build rules).

Make expands the `{Default}` variable to the common part of filenames matched by a default rule. The variable is defined dynamically when Make applies the default rule.

Applying Default Rules Across Directories

Normally, default rules work only within a single directory because these rules specify only the suffixes of filenames. Directory dependency rules allow default rules to be applied across directories.

Just as default rules are applied to filenames that are the same except for their suffixes, directory dependencies are defined to resolve situations where source files and object files are kept in different directories—that is, the directory portion of the pathname is different for the two files. For example, if you keep all your source files in `{MPW}:Sources:` and all your object files in `{MPW}:Objects:`, you could include a directory dependency line like the following:

```
"{MPW}:Objects:" ƒ "{MPW}:Sources:"
```

Make uses this rule together with its own built-in default rules to infer that the file `{MPW}:Objects:Sample.c.o` depends on the file `{MPW}:Sources:Sample.c`.

As you might suspect, it does not make sense to include build commands for directory dependency rules.

The `{DepDir}` and `{TargDir}` variables shown in Listing 10-5 are built-in Make variables that allow default rules to work when the target and prerequisite files reside in different directories. The `{DepDir}` variable is set to the directory component of the prerequisite name. The `{TargDir}` variable is set to the directory component of the target name.

Make assigns values to these variables using the directory names you specify in directory dependency rules and uses these values in writing out the correct build command; for example:

```
SC "{MPW}:Sources:Sample.c" -o "{MPW}:Objects:Sample.c.o"
```

The `{DepDir}` and `{TargDir}` variables have values only when used in the build commands of default rules for which directory dependency rules were applied. In all other cases, these variables evaluate to the null string so that they won't interfere with the normal behavior of default rules.

Directory dependency rules have this general form:

targetDirectory: … ƒ *searchDirectory*: …

More than one directory name can appear on either side of the dependency line. You can specify the current directory by a single colon (`:`).

Directory dependency rules are applied only during the processing of default rules. If Make is applying a default rule and encounters a target name with a directory component, Make checks for a directory dependency rule for that directory. If one exists, Make tries prerequisite filenames with the directory prefixes given on the right side of the rule. The names are tried in the order they appear in the rule; thus, more than one directory name on the right side of a directory dependency rule constitutes a list of directories to search.

If default rules are meant to be applied from a directory A: to a directory B: and also within A: (that is, from A: to A:), then A: should appear on both the left and right sides of the directory dependency rule—for example,

```
A:  ƒ  A: B:
```

Specifying Secondary Dependencies

You can specify secondary dependencies in a default rule dependency line. Once again, the syntax for a default rule is

`.`[*suffix1*] ƒ `.`*suffix2* [*secondaryDependency*...]
ShellCommand...

Secondary dependencies can be more filename suffixes or fixed filenames. This default rule extension is useful for MacApp, where, typically, an object file depends on both its interface and include files. This dependency can be stated in a single rule—for example,

```
.c.o ƒ .c .c.incl
```

The filename implied by *.suffix2* is treated as the primary dependency because it triggers the subsequent build command. The filename implied by *.suffix2* must be valid for the default rule to be applied. To be valid, a filename must appear in the makefile, exist in the file system, or lead to a valid filename by further application of default rules.

Make processes secondary dependencies as follows:

- The filename implied or specified as a secondary dependency need not exist for the default rule to be applied. If the primary dependency does not exist, the secondary dependency is not processed.
- If the secondary dependency is a filename suffix or a fixed filename, a dependency is added if the expressed or implied filename is valid. Using the previous example, if `program.c` were a valid filename, Make would also compare the dates of `program.c.o` and `program.c.incl` to determine whether the target was up-to-date.
- If a filename specified by a secondary dependency is not valid, then no dependency is added and the default rule is processed as usual.

Using Built-in Default Rules

If you are planning to have an object file built by any of the default rules shown in Listing 10-5 on page 10-23, you do not need to include the build rule for that file in your makefile because the rule is already expressed by the default rule. You do need to do the following:

- Include additional dependencies, such as those on include files. You can express these dependencies by dependency lines with no build component; for example:

  ```
  MyFile.c.o ƒ Types.h
  ```

- Supply variable definitions for compiler or assembler options. For example, you might want to specify the search path for your C include files by setting the `{COptions}` variable as follows:

  ```
  COptions = -i MPW:CIncludes
  ```

- Specify directory dependencies if your source files and object files reside in different directories.
- Specify dependencies on the makefile.

Creating Your Own Default Rules

If the built-in Make default rules do not suit your purpose, you can either add a specific build rule to your makefile or add your own default rule. For example, if you are using a compiler for which no built-in default rule exists, you might add a specific build rule as in this example:

```
MyProgram.q.o ƒ MyProgram.q
    MyCompiler {MyCompilerOptions} -o MyProgram.q.o
```

Or you might add a default rule and variable definitions to eliminate the need for including the specific build rules in your makefile—for example:

```
QCompiler = MyCompiler
.q.o        ƒ        .q
    {QCompiler}{DepDir}{Default}.q ∂
        -o {TargDir}{Default}.q.o {QOptions}
```

If you add new default rules or modified versions of the built-in default rules, you should add them right after the variable definitions for your makefile.

Make applies default rules only if the file implied by the right-side suffix of the rule exists or if Make can arrive at a file that exists by further application of default rules.

If the left side of a default rule has more than one period (or component), there is the possibility that more than one default rule applies. For example, you might have a default rule for building `.o` files and another for building `.c.o` files. Because Make tries to apply default rules by matching the longest suffix first, the `.c.o` rule is tried first.

Default rules of the form

```
. ƒ .suffix
    ShellCommands...
```

specify dependencies between files with any name and files with the same name followed by the given suffix. Note, however, that default rules of this form slow down Make processing because the empty left side of the rule causes Make to check all filenames for a match.

You can use the built-in variable `{Deps}` to write a default rule for builds or links where all the dependency files will be linked together—for example:

```
. ƒ  .o
    {Link}  {LinkOptions}  {Deps}  -o {TargDir} {Default}
```

In this example, `{Deps}` represents any filename with the `.o` suffix.

Variables in Makefiles

You can use exported Shell variables, built-in Make variables, and variables you define yourself in makefiles. These are described in the following sections. You can define variables in the makefile or on the Make command line by using the `-d` option.

Shell Variables

Make automatically defines exported Shell variables before it reads the makefile, so you can use Shell variables in dependency lines and build commands.

- Shell variables in dependency lines are expanded because they are typically filenames or parts of a filename. Unidentified variables in dependency lines are reported as errors.
- Shell variables in build rules pass through unexpanded so that the MPW Shell will be able to process and expand them. If Make doesn't recognize a variable reference in a build command line, it leaves the command unchanged so that it can be processed later by the Shell.

▲ **WARNING**
Exported MPW Shell variables override Make variables with the same names. An attempt to redefine an MPW Shell variable in the makefile results in a warning message. ▲

Built-in Make Variables

Table 10-1 lists the built-in Make variables, some of which have already been mentioned in the discussion of the Make built-in default rules.

Table 10-1 A summary of built-in Make variables

Variable	Value
`AOptions`	Options for the assembler specified by `{Asm}`.
`Asm`	Set to `Asm` to specify the MPW Assembler. You can change this value to specify the MPW-compatible assembler of your choice.
`C`	Set to `SC` to specify the SC compiler. You can change this value to specify the MPW-compatible C compiler of your choice.
`COptions`	Options for the compiler specified by `{C}`.
`CPlus`	Set to `SCpp` to specify the SCpp C++ compiler. You can change this value to specify the MPW-compatible C++ compiler of your choice.
`CPlusOptions`	Options for the compiler specified by `{CPlus}`.
`Default`	Common part of filenames matched by a default rule. Make sets this value dynamically when applying the default rule.
`DepDir`	The directory portion of the prerequisite's pathname.
`Deps`	A list of the target's direct prerequisites.
`NewerDeps`	A list of names of all the target's direct prerequisites that were newer than the target. Make creates this list dynamically when it generates build commands.

continued

Table 10-1 A summary of built-in Make variables (continued)

Variable	Value
`Pascal`	Set to `Pascal` to specify the MPW Pascal compiler. You can change this value to specify the MPW-compatible Pascal compiler of your choice.
`POptions`	Options for the compiler specified by `{Pascal}`.
`PPCAOptions`	Options for the PowerPC assembler specified by `{PPCAsm}`.
`PPCAsm`	Set to `PPCAsm` to specify the PowerPC Assembler. You can change this value to indicate an MPW-compatible PowerPC assembler of your choice.
`PPCC`	Set to `MrC` to specify the MrC PowerPC compiler. You can change this value to specify your own MPW-compatible PowerPC C compiler.
`PPCCOptions`	Options for the compiler specified by `{PPCC}`.
`PPCCplus`	Set to `MrCpp` to specify the MrCpp PowerPC C++ compiler. You can change this value to specify your own MPW-compatible PowerPC C++ compiler.
`PPCCplusOptions`	Options for the compiler specified by `{PPCCplus}`.
`Targ`	The complete filename of the target on the left side of the dependency rule whose build commands are being processed.
`TargDir`	The directory portion of the target's pathname.

Note
The built-in Make variables are not case-sensitive. ◆

Because the values of the `{Targ}`, `{NewerDeps}`, and `{Deps}` built-in variables are dynamically assigned when Make generates the build commands, you cannot override these values and must only use the variables in build rules. These variables have no value when dependency lines are processed.

When default rules are applied, the `{Default}`, `{TargDir}`, and `{DepDir}` variables are also defined.

Note
When expanding the built-in variables `{Targ}`, `{NewerDeps}`, `{Deps}`, `{Default}`, `{TargDir}`, and `{DepDir}` in build commands, Make automatically quotes their values, if necessary, because they will represent filenames or parts of pathnames. Don't quote them yourself. ◆

Defining Your Own Make Variables

You can define your own variables in makefiles. One common use of variables is to provide parameters to the directory portion of filenames so that you can easily adapt a makefile to different directory setups. Another use is to create a list of filenames that will be used in more than one place.

Variable definitions take the form

variableName = *stringValue*

Subsequent appearances of {*variableName*} are replaced by *stringValue*. Any leading or trailing blanks or tabs are removed from the variable definition. Variable definitions

- can be continued across lines using the ∂ character (Option-D) so that *stringValue* can be any desired length
- can contain references to other variables
- cannot contain the ASCII characters 0 or 1 (in C, \000 or \001)

When *stringValue* is continued across lines, a single blank replaces any comments, blanks, or tabs at the end of the continued line and at the beginning of the following line. Thus, variable values can conveniently contain lists of files.

Make variables are not expanded until they are used in dependency lines or until generated in a build command. Therefore, you must

- define any variables appearing in dependency lines somewhere previously in the makefile because variables in dependency lines are expanded immediately to produce filenames
- define variables in build rules anywhere in the makefile because variables in build rules are not expanded until after the command lines are generated—that is, after the entire makefile has been read

You can define a variable on the Make command line by using the `-d` option. This option overrides any definition of the variable within the makefile, thus allowing the definition in the makefile to function as a default.

Examples of Makefiles

The following four sections offer examples of makefiles.

- The first example is included for those who have never created a makefile. It shows how to build a very simple makefile for a program with a realistic variety of components.
- The second example illustrates the use of variables in makefiles to trigger different builds depending on whether the build is done in MPW 3.3 or MPW 3.4.
- The third example shows the makefile used to build a version of the Make tool itself.
- The fourth example shows how to assign different versions of a build to different folders (in this case PowerPC, classic 68K, and a fat binary combination), and also illustrates how to include dependencies to generate multiple makefiles from one file.

If you have never written a makefile, you should read through the next section, "Example 1—Creating a Makefile." It offers a method for creating a makefile based on examining the relationship of the files that feed into your final program. If you have written a makefile before, you should look at the next three examples: "Example 2—A Makefile Using Modified Default Rules," beginning on page 10-40, "Example 3—Make's Makefile," beginning on page 10-44, and "Example 4—Multiple Folders and Multiple Makefiles," beginning on page 10-48.

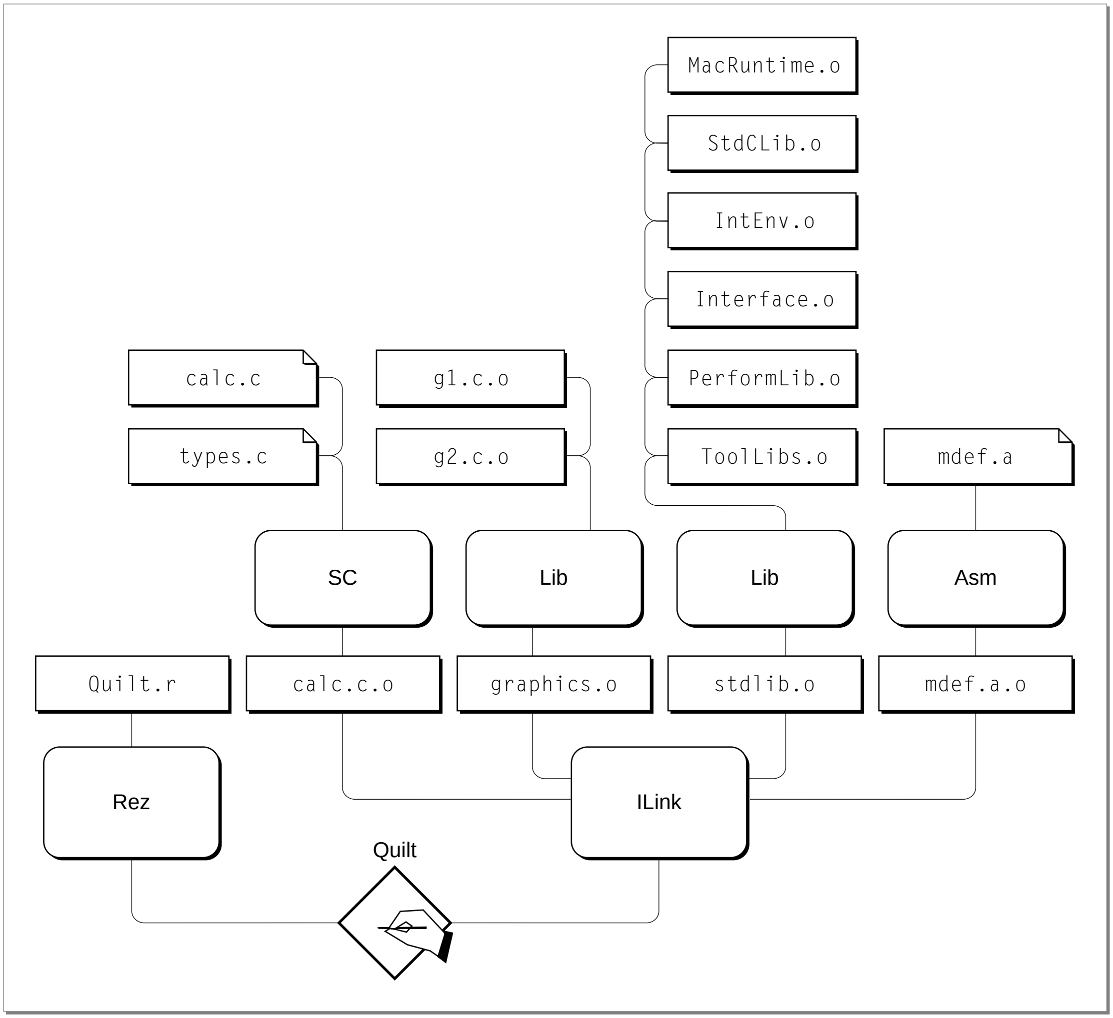
Example 1—Creating a Makefile

What your makefile contains depends on the number and kinds of source files that are needed to build your program. Figure 10-2 shows the source files and the commands used to create the executable program `Quilt`.

The source files for the `Quilt` program shown in Figure 10-2 include

- `Quilt.r`, a resource description file
- `mdef.a`, a menu definition
- `stdlib.o`, a file containing the standard system libraries required for the link
- `graphics.o`, a library containing your own graphics routines that have been completed and tested
- `calc.c.o`, an object file that is still being developed, which in turn depends on the interface file `types.c` and the implementation file `calc.c`

Figure 10-2 Source files for the `Quilt` program



One way to start writing the makefile for this build is to write out the build commands necessary to build the files shown in Figure 10-2. First, write the command that links the compiled code into the 'CODE' resources for the final program file:

```
ILink -mf -d -o Quilt ∂
        calc.c.o ∂
        graphics.o ∂
        stdlib.o
```

Next, write the ILink command that links the mdef.a.o file into the final program file:

```
ILink -o Quilt -rt MDEF=256 -m MDEF0 -sn Main=MySpecialMDEF ∂
        mdef.a.o
```

Linking the mdef.a.o file requires a separate ILink command because it must be linked as an 'MDEF' resource rather than as a 'CODE' resource.

Finally, write the Rez command that compiles the noncode resources:

```
Rez -a -o Quilt Quilt.r
```

Because the Rez command is executed after the ILink command, you must use the Rez option -a to append the compiled resources to the Quilt program file. If you were to always execute the Rez command before the ILink command, this would not be necessary. However, it's highly recommended that you always use the -a option, just for safety and peace of mind.

Before writing the dependency lines that trigger these build commands, note first that the program file Quilt is the target reached by multiple dependency paths and that each path terminates in a different build command. You must use the special symbol *ƒƒ* in the dependency line for the files involved in building the target program file; otherwise, Make will think that you are issuing different build commands for the same target file.

You can now write the dependency rules for the ILink and Rez build commands:

```
Quilt       ƒƒ      mdef.a.o
    ILink -o Quilt -rt MDEF=256 -m MDEF0 -sn Main=MySpecialMDEF ∂
        mdef.a.o
```

```
Quilt       ƒƒ      calc.c.o ∂
                    graphics.o ∂
                    stdlib.o
    ILink -mf -d -o Quilt calc.c.o graphics.o stdlib.o

Quilt       ƒƒ      Quilt.r
    Rez -a -o Quilt Quilt.r
```

Next, you must include in the makefile all the build commands for the files that are linked into the final program (`mdef.a.o`, `calc.c.o`, `stdlib.o`, and `graphics.o`) and the dependency lines that trigger these builds:

```
mdef.a.o ƒ mdef.a
    Asm mdef.a

calc.c.o ƒ calc.c
    {C} -model farcode -e "myError" calc.c

calc.c.o ƒ types.h
# No build rule is necessary here. You only need to declare the
# dependency. If types.h is newer than calc.c.o, calc.c.o is
# rebuilt.

stdlib.o ƒ "{Libraries}IntEnv.o" "{Libraries}MacRuntime.o" ∂
        "{CLibraries}StdCLib.o" "{Libraries}Toollibs.o" ∂
        "{Libraries}Interface.o" "{Libraries}PerformLib.o"
    Lib -mf -o "{Libraries}IntEnv.o" ∂
        "{Libraries}MacRuntime.o" ∂
        "{CLibraries}StdCLib.o" ∂
        "{Libraries}Toollibs.o" ∂
        "{Libraries}Interface.o" ∂
        "{Libraries}PerformLib.o"

graphics.o ƒ g1.c.o g2.c.o
    Lib -mf -o graphics.o g1.c.o g2.c.o
```

Determining the dependencies among files, writing dependency lines that express these dependencies, and writing build commands that must be executed to bring files up-to-date are the basic methods of writing a makefile.

Listing 10-6 shows the complete makefile for the `Quilt` program. Note the use of variables to simplify the makefile.

Listing 10-6 `Quilt.make` makefile

```
OBJECTS             =   calc.c.o graphics.o stdlib.o
MYLIBS =    "{Libraries}IntEnv.o""{Libraries}MacRuntime.o"∂
            "{CLibraries}StdCLib.o""{Libraries}Toollibs.o"∂
            "{Libraries}Interface.o""{Libraries}PerformLib.o"
GraphicLIBS         =   g1.c.o g2.c.o
COptions            =   -model farcode -e "myError"

Quilt       ƒƒ      mdef.a.o
    ILink -o Quilt -rt MDEF=256 -m MDEF0 -sn Main=MySpecialMDEF ∂
        mdef.a.o

Quilt       ƒƒ      {OBJECTS}
    ILink -mf -d -o Quilt {OBJECTS}

Quilt       ƒƒ      Quilt.r
    Rez -a -o Quilt Quilt.r

mdef.a.o ƒ mdef.a
    Asm mdef.a

stdlib.o ƒ {MYLIBS}
    Lib -mf -o stdlib.o  {MYLIBS}

graphics.o ƒ {GraphicLIBS}
    Lib -mf -o graphics.o {GraphicLIBS}

calc.c.o ƒ calc.c types.h
    {C} {Coptions} calc.c
```

Example 2—A Makefile Using Modified Default Rules

The makefile `SoundApp.Make`, shown in Listing 10-7, illustrates the use of variables in makefiles to trigger different builds depending on whether the build is done in MPW 3.3 or MPW 3.4.

Note
It is recommended that you not mix libraries from MPW 3.4 with headers from MPW 3.3. The focus of this example is to illustrate using MPW Shell and Make variables together to provide dynamic dependencies. ◆

Notes 1 through 10, beginning on page 10-42, describe how the makefile works; numbered comments in the makefile indicate which note explains that particular section of the makefile. Notice that the first and longest part of the makefile is dedicated to variable definitions. This simplifies the job of maintaining the makefile and also makes the subsequent dependency rules and build commands much easier to read.

Listing 10-7 `SoundApp.Make` makefile

```
# Components:

#    SoundApp.make            May 1, 1990         MPW build script
#    SoundApp.c               May 1, 1990         C source code
#    SoundApp.r               May 1, 1990         Rez source code
#    SoundAppSnds.r           May 1, 1990         Rez source code
#    SoundUnit.c              May 1, 1990         C source code

AppName             =    PSoundApp                                   # see note 1
Signature           =    'SAPP'
UtilityFolder       =    ::Utilities:        #:: means one level above current folder
SymOptions          =    -sym off            # turn this on to debug with SADE

CLibraries33        =    {MPW}Libraries:CLibraries33:                # see note 2
Libraries33         =    {MPW}Libraries:Libraries33:
```

```
IncludesFolders          =   -i {UtilityFolder}

RezOptions               =   -rd -append {IncludesFolders} ∂
                             -d Signature="{Signature}" -d AppName='{AppName}'

LinkOptions              =   {SymOptions} {SegmentMappings}
SegmentMappings          =   -sn INTENV=Main ∂
                             -sn STDCLIB=Main ∂
                             -sn MAFailureRes=Main


.c.o    ƒ   .c                                                  # see note 3
    {C} {COptions} {CAltOptions} {DepDir}{Default}.c -o {TargDir}{Default}.c.o

AppObjects      =   ∂                                           # see note 4
            "{UtilityFolder}Utilities.c.o" ∂
            SoundUnit.c.o   ∂
            SoundApp.c.o

CLibs34     =   ∂                                               # see note 5
                    "{CLibraries}StdCLib.o" ∂
                    "{Libraries}MacRuntime.o" ∂
                    "{Libraries}Interface.o" ∂
                    "{Libraries}IntEnv.o" ∂
                    "{Libraries}ToolLibs.o"

CLibs33     =   ∂
                    "{CLibraries33}StdCLib.o" ∂
                    "{Libraries33}Runtime.o" ∂
                    "{Libraries33}Interface.o" ∂
                    "{Libraries33}ToolLibs.o"

                                                                # see note 6
UtilityRezFiles             =           "{UtilityFolder}Utilities.r" ∂
                                        "{UtilityFolder}UtilitiesCommon.h"

UtilityHeaderFiles          =           "{UtilityFolder}Utilities.h" ∂
                                        "{UtilityFolder}UtilitiesCommon.h"
UtilityInterfaceFiles       =           "{UtilityFolder}Utilities.c"

{UtilityFolder}Utilities.c.o        ƒ   "{UtilityFolder}Utilities.h" ∂
                                        "{UtilityFolder}UtilitiesCommon.h"
```

```
                                                                 # see note 7
SoundApp.c.o        ƒ   {AppName}.make SoundUnit.c {UtilityInterfaceFiles}

SoundUnit.c.o       ƒ   {AppName}.make

{AppName}       ƒƒ ShellForce                                    # see note 8

ShellForce              ƒ
    BEGIN
        IF "{ShellVersion}" == ""
            ( EVALUATE "`Version`" =~ /MPW Shell≈ ([0-9]+(.[ab0-9]+)+)®1≈/ ) ∂
                ∑ Dev:Null
            SET ShellVersion "{®1}"
        END
        IF "{ShellVersion}" =~ /3.4≈/
            SET CAltOptions "-d MPW34"
            SET RezAltOptions "-d MPW34"
            SET CSysObjects "`QUOTE {CLibs34}`"
        ELSE
            SET CAltOptions "-d MPW33"
            SET RezAltOptions "-d MPW33"
            SET CSysObjects "`QUOTE {CLibs33}`"
        END
    END ∑ Dev:Null

{AppName}       ƒƒ {AppObjects}                                  # see note 9
    Link {LinkOptions} -o {Targ} {AppObjects} {CSysObjects}
    SetFile {Targ} -t APPL -c {Signature} -a B

{AppName}       ƒƒ SoundAppSnds.r {AppName}.make                 # see note 10
    Rez {RezOptions} {RezAltOptions} -o {Targ} SoundAppSnds.r

{AppName}       ƒƒ SoundApp.r {AppName}.make
    Rez {RezOptions} {RezAltOptions} -o {Targ} SoundApp.r
```

Notes

1. These variable definitions control compiler options, linker options, and search directories. Note the use of variable names on the right side of variable definitions.
2. To avoid name conflicts, MPW 3.3 libraries must be placed in separate folders from the newer MPW 3.4 libraries. Note that MPW 3.3 libraries will conflict with MPW 3.4 header files.

3. These dependency rules include modified versions of default build rules; they are used to take into account differences between MPW 3.3 and 3.4. Later in Listing 10-7, the structured commands following the `ShellForce` dependency determine which compiler options are used in the commands generated by Make.

4. These are the objects that need to be linked—the direct prerequisites of the target to be built. If any one of these changes, Make outputs the Link command.

5. Which library files are linked depends on whether MPW 3.3 or MPW 3.4 is being used. Under MPW 3.4, the `StdCLib.o` and `Runtime.o` libraries have been regrouped into `StdClib.o`, `MacRuntime.o`, and `IntEnv.o`. The appropriate choice is made dynamically before compiling and linking occur; the selection is triggered by the `ShellForce` dependency line and evaluated by the structured commands that follow that line.

6. These generic utility dependencies can be pasted in when needed. It's likely they won't all be used, but any extra ones are ignored.

7. These are dependency rules for the individual components. They are invoked by the Make built-in default rules. Note the dependency on the makefile itself.

8. This dependency rule forces the subsequent commands to execute. It must be the first rule specified for `{AppName}` so that it is executed first. Forcing the subsequent commands to execute determines the version of the MPW Shell that is currently running and allows the selection of compiler options and libraries that are appropriate for that version. You need to define the `{ShellVersion}` variable on the Make command line; for example,

   ```
   Make -f SoundApp.make -d ShellVersion=3.4
   ```

   If you don't supply a value for `{ShellVersion}`, the first `IF` clause determines the version by using the MPW `Version` command.

9. This build rule links the application. If any of the objects change or the makefile changes, the `Link` and `SetFile` commands are executed.

   Note that `{CSysObjects}` is an MPW Shell variable, not a Make variable. You can include it in the build rules but not in the dependency line because it is not yet defined when Make executes.

10. This build rule creates the application's resources and appends them to the program file.

Example 3—Make's Makefile

Listing 10-8 shows the makefile used to build an experimental version of the Make tool (represented in this makefile by the `MakeX` target). The notes, beginning on page 10-46, describe in detail a number of the Make features that were used.

Listing 10-8 Make's makefile

```
ToolDir          =   {Boot}ToolUnits:                    #   see note 1
ObjDir           =   :Obj:
MakeIncludes     =   "{ToolDir}"MemMgr.c.o ∂             #   see note 2
                     "{ToolDir}"SymMgr.c.o ∂
                     "{ToolDir}"Utilities.c.o ∂
                     "{ToolDir}"CursorCtl.c.o ∂
                     "{ToolDir}"ErrMgr.c.o ∂
                     "{CIncludes}"IntEnv.h ∂
                     "{CIncludes}"MemTypes.h ∂
                     "{CIncludes}"QuickDraw.h ∂
                     "{CIncludes}"OSIntf.h

MakeObjs         =   "{ObjDir}"Make.c.o ∂
                     "{ToolDir}"Utilities.c.o ∂
                     "{ToolDir}"MemMgr.c.o ∂
                     "{ToolDir}"SymMgr.c.o ∂
                     "{ToolDir}"CursorCtl.c.o ∂
                     "{ToolDir}"ErrMgr.c.o

# Use this one when compiling with SC.
SCLibs  =   "{Libraries}IntEnv.o" ∂
            "{Libraries}MacRuntime.o" ∂
            "{CLibraries}StdCLib.o" ∂
            "{Libraries}ToolLibs.o" ∂
            "{Libraries}Interface.o" ∂
            "{Libraries}PerformLib.o"

LinkOpts         =   -w      # no warnings
                                                         #   see note 3
```

```
SourceFiles           =   Make.c ∂
                          DefaultRules ∂
                          Makefile

COptions              =   -i {Boot}ToolUnits:              #   see note 4

"{ObjDir}"  ƒ         :                                    #   see note 5

MakeX       ƒƒ  {MakeObjs} {SCLibs}                        #   see note 6
    ILink {LinkOpts} -p -b -o MakeX ∂
    -t MPST -c "MPS " ∂
    {MakeObjs} {SCLibs} ≥LinkMsgs

MakeX       ƒƒ  defaultRules
    Duplicate -d defaultRules MakeX -y           # copy default rules into
                                                 # Make's data fork

MakeX       ƒƒ  Make.r
    Rez Make.r -o MakeX -a                       # Make's Commando resource

MakeX       ƒƒ  {MakeObjs} {Libs} defaultRules
    SetFile MakeX -m . -d .                      # set last-mod and
                                                 # creator dates

"{ObjDir}"Make.c.o      ƒƒ      Make.c                     #   see note 7
    Save Make.c ≥Dev:Null || Set Status 0        # save source before
                                                 # compile if changed

"{ObjDir}"Make.c.o      ƒƒ      {MakeIncludes}             #   see note 8

"{ToolDir}"MemMgr.c.o          ƒ   "{ToolDir}"Utilities.c.o ∂  #   see note 9
                                   "{CIncludes}"MemTypes.h

"{ToolDir}"SymMgr.c.o          ƒ   "{ToolDir}"MemMgr.c.o ∂
                                   "{CIncludes}"MemTypes.h

"{ToolDir}"Utilities.c.o       ƒ   "{CIncludes}"MemTypes.h

Backup          ƒ                                          #   see note 10
    Duplicate -y  ≈  MakeSrc:                    # backup
```

```
Restore         ƒ
    Duplicate -y  MakeSrc:≈  :                      # restore from backup

Listings        ƒ   {SourceFiles}                               # see note 11
    Print -h -r -ls .85 -s 8 -b -hf helvetica -hs 12 {NewerDeps}
    Echo "Last listings made ‘Date‘" >Listings
```

Notes

1. The exported Shell variable `{Boot}`, used in the definition of `{ToolDir}`, is defined in the MPW Shell's startup script.
2. Several variables—`{MakeIncludes}`, `{MakeObjs}`, `{SCLibs}`, and `{SourceFiles}`—are used for lists of filenames. This is a convenience because the lists are used in several places later in the makefile; it also helps to reduce errors. Note that you can temporarily remove any file from the list by placing the comment character (#) at the beginning of the line for the file.
3. The `{LinkOpts}` variable is used to specify linker options (and is used only once). This usage is handy because the definition in the makefile functions as a default that can be overridden from the command line with the `-d` option, as in

   ```
   Make -d LinkOpts='-w -l >Map'
   ```

4. The `{COptions}` definition gives a value to one of the variables used in the default rules, customizing the built-in default rules for C compiles for this particular makefile.
5. This directory dependency rule allows the MakeX tool's objects and sources to be in different directories and yet be built by the built-in default rules. In particular, `Make.c.o` will be in the `:Obj:` directory while `Make.c` is in the current directory. Note that for this device to work, `Make.c.o` must appear with the object directory prefix. Thus, it appears in the makefile as `{ObjDir}Make.c.o`.
6. There are four sets of double-ƒ dependency rules for MakeX. They use the following commands:
   - the `ILink` command, which creates the MakeX tool's code resources
   - the `Duplicate` command, which copies the default rules to the data fork of the `MakeX` file. Make reads the built-in default rules from its own data fork.

- the `Rez` command, which creates the Commando resource for the `MakeX` tool
- the `SetFile` command, which sets the creation and modification dates

The link takes place only if the MakeX objects or libraries change. The resource compiler will rebuild the `'cmdo'` resource for the Make tool only if `Make.r` has changed. The default rules are copied only if the rules have changed. And the setting of the dates will take place if either of the first two rules was activated. (Note that the fourth rule has the union of the dependency relations of the first two.)

7. The two double-ƒ dependency rules for `Make.c.o` control the compilation of the main source for Make, with some interesting side effects. The first double-ƒ rule saves the Make source before it is compiled, only if the source file has changed. The second double-ƒ rule does the actual compile. Note that this last rule has no explicit build commands, so it will be augmented by the built-in default rules, which will add a dependency relation (on the source file `Make.c`) and will supply the actual build commands for the compile.
8. The `{ObjDir}` prefix is necessary for the directory dependency rule to take effect. It allows the object and source files to be in different directories.
9. The dependency rules for `MemMgr`, `SymMgr`, and `Utilities` describe dependencies between various utility units used by Make. Several dependencies on library interface files are given. Dependencies among the utility units themselves are described by indicating a dependency on the object files of the lower-level (predecessor) units. These dependencies could have been expressed as dependencies on the source files of the lower-level units (because it is the source files that are read in a `Uses` list). However, expressing these dependencies on the object files has the nice property of ensuring that the lower-level units have been successfully compiled before the higher-level units are built.
10. The `Backup`, `Restore`, and `Listings` targets are additional roots (top-level targets) in Make's makefile, and thus represent utility targets—other things that can be built besides MakeX itself. Note that the `Backup` and `Restore` targets do not actually get built by their build rules; they are thus artificial targets and will always generate build commands if they are specified on the Make command line. Note also that they do not have any dependency relations.
11. The build rules for the `Listings` target demonstrates the use of the `{NewerDeps}` variable. The prerequisite of `Listings` is a list of the Make source files. The first build command prints the `{NewerDeps}` files. The `{NewerDeps}`

variable contains the names of the prerequisites that are newer than the target; that is, the source files that have changed since listings were last made. The last line of the build rules simply writes the current date into a file called `Listings`, which is the name of the target. This action results in a file that remembers when listings were last made. Writing the date into the file is unnecessary but convenient; the `Echo` itself is enough to change the file's last-modified date.

Note
There are several implicit builds that are generated as needed by the default rules. For example, the `{MakeObjs}` variable includes several assembly-language object files. Because `{MakeObjs}` appears as a prerequisite of the link step, these assemblies are generated, if necessary, before the link. ◆

Example 4—Multiple Folders and Multiple Makefiles

The makefile in Listing 10-9 can build PowerPC, classic 68K, and fat binary versions of the application SillyBalls found in the CExamples folder.

Executing the basic Make command generates the fat binary version of SillyBalls. You can also specify the following high-level targets:

- `make SillyBalls.PPC`, which builds the PowerPC version of SillyBalls (`SillyBalls.PPC`)
- `make SillyBalls.68K`, which builds the classic 68K version of SillyBalls (`SillyBalls.68K`)

Specifying one of the targets above generates another Make command line using this same makefile and a low-level target. This technique allows you to use the same set of low-level build rules for both the PowerPC and classic 68K versions of SillyBalls.

This example also stores the PowerPC and classic 68K object files in separate directories. The convention is to place the PowerPC objects in the directory `:PPCObjects:` and the classic 680x0 objects in `:68KObjects:`. This procedure keeps the objects distinct, and thus there is no need to apply separate naming conventions for the two runtime architectures.

For more detailed information, see the notes beginning on page 10-52.

Listing 10-9 Makefile with multiple folders and multiple target makefiles

```
# Makefile example illustrating the creation of a 68K, PPC, and fat binary application.
# The following options can be specified to Make when using this makefile:

# Options: -d Dir=<directory>       Allows you to define which directory the source is
#                                   in, thus not requiring the current MPW directory be
#                                   the same as the sources.

#          -d e=-e          Forces a full build.


# Directory information -- see note 1

Dir         =   ":"                     # all sources, etc. in this directory
Objects     =   {Dir}{Binary}Objects:   # 68K and PPC objs go in separate directories
AppName     =   SillyBalls              # name (not pathname) of this app

Makefile    =   {Dir}SillyBalls.make    # the pathname for this makefile
MakeOutput  =   "{TempFolder}MakeScript"# where to output and execute Make outputs


# Miscellaneous macro definitions.  Define any additional macros that might be useful
# for the build here.

Binary  =                               # will be "68K" or "PPC"
e       =                               # define as '-e' for full builds

Creator =   'MOOF'
Type    =   'APPL'


# Define overrides and options for the standard Make default build rules.  Everything
# that will work its way into the default build rules is defined here.  It is
# recommended you define additional options to control debugging, optimization, etc.

C           =                           # will be 68K SC or PPC MrC compiler
Opt         =   all                     # default optimization for SC
PPC_Opt     =   size                    # default optimization for MrC
Mbg         =   full                    # default MacsBug setting for SC
MacsBug     =   -mbg {Mbg}              # will be the option in COptions
```

```
ExtraOpts   =                          # for additional target-specific options

COptions    =   -opt {Opt}             # defines optimization level ∂
                {MacsBug}              # defines MacsBug symbolic information for SC ∂
                -i "{CIncludes}"       # always override SC's {SCIncludes} ∂
                {ExtraOpts}


# Define the standard libraries.  The libraries used for 68K links and PPC links are
# obviously different.

68KLibs =       "{Libraries}"Interface.o ∂
                "{Libraries}"IntEnv.o ∂
                "{Libraries}"MacRuntime.o ∂
                "{Libraries}"MathLib.o ∂
                "{CLibraries}"StdCLib.o

PPCLibs =       "{SharedLibraries}"InterfaceLib ∂
                "{SharedLibraries}"StdCLib ∂
                "{SharedLibraries}"MathLib ∂
                "{PPCLibraries}"StdCRuntime.o ∂
                "{PPCLibraries}"PPCCRuntime.o


# Define link objects and directory dependencies.  There is only one file and thus one
# object in this example.  Note that {Objects} is :68KObjects: or :PPCObjects: as
# defined above.

SillyBalls.o = {Objects}SillyBalls.c.o

{Objects}   ƒ   {Dir}


# Definitions used for high-level targets.  These are options used for the
# low-level make.  By defining these macros this way, the low-level make lines
# become almost self-documenting.

68K         =   {Dir}{AppName}.68K  -d Binary=68K# low-level targets
PPC         =   {Dir}{AppName}.PPC  -d Binary=PPC
FAT         =   {Dir}{AppName}      -d Binary=68K
```

```
Common      =   -f {Makefile} {e} ∂
                -d C='SC' ∂
                -d Opt="{Opt}"∂
                -d ExtraOpts='-b3' ∂
                -d Mbg="{Mbg}"∂
                -d Dir={Dir}


PPC_Common  =   {Common} ∂
                -d C=MrC ∂
                -d OPT="{PPC_Opt}" ∂
                -d ExtraOpts='' ∂
                -d MacsBug=


# High-level targets -- see note 2

{AppName}       ƒ   {AppName}.68K {AppName}.PPC
    Make {FAT} {Common} >{MakeOutput}
    {MakeOutput}

{AppName}.68K ƒ $OutOfDate
    Make {68K} {Common} >{MakeOutput}
    {MakeOutput}

{AppName}.PPC ƒ $OutOfDate
    Make {PPC} {PPC_Common} >{MakeOutput}
    {MakeOutput}


# Low-level targets-- see note 3

{Dir}{AppName}.68k ƒƒ{68KLibs} {SillyBalls.o}
    Link -c {Creator} -t {Type}  ∂
         {68KLibs} ∂
         {SillyBalls.o} ∂
         -o {AppName}.68k

{Dir}{AppName}.PPC  ƒƒ {68KLibs} {SillyBalls.o}
    PPCLink -c {Creator} -t {Type}∂
            {PPCLibs}       ∂
```

```
            {SillyBalls.o}∂
            -o {AppName}.PPC

{Dir}{AppName}.{Binary} ƒƒ {AppName}.r
    Rez {AppName}.r -append -o {AppName}.{Binary}

{Dir}{AppName} ƒ $OutOfDate
        Duplicate -y {Dir}{AppName}.PPC {Dir}{AppName}
        Echo "include ∂"{Dir}{AppName}.68K∂" 'CODE';" | Rez -a -o {Dir}{AppName}
```

Notes

1. Adding directory information generalizes the makefile and allows for someone else to easily port the makefile to their environment; it also removes the requirement that you must be in the same directory in order to do a build. You should define this information at the beginning of a makefile. The macro `{Binary}` is used during the low-level build to define which object's directory (and target) is being used. It is defined in the miscellaneous macro definitions section.
2. These are the targets discussed in the beginning of the section. Note that `{AppName}` is the first target. Since this is the first dependency in this makefile, explicitly specifying it is optional and it can be omitted. Note how the definitions in the previous section make defining these three variants simple and readable. These targets generate the Make command line with its output redirected to the file `{MakeOutput}` (in this example it is located in the `MPW "{TempFolder}"`, which is a convenient place for such things). The output is then executed. This initiates the low-level make to do the actual builds. `{Common}` and its super-set `{PPC_Common}` (which overrides some of the definitions in `{Common}`) appropriately parameterize the build to use the proper compiler, and so forth.
3. The high-level build lines generate the Make lines that use the low-level targets. Both the 68K and PowerPC targets have double-ƒ dependency rules since they are full builds in their own right. Both require the `.r` file, which also has a double-ƒ dependency. The Rez build rule also illustrates how the macro `{Binary}` can be conveniently used as a target that applies to both the 68K and PPC builds. The last build line is a single-ƒ dependency, which is used only to build the fat application. To build the fat application, duplicate the PowerPC version of the application (which includes its resources) and then append the `'CODE'` resources from the 68K application using Rez. The Echo command passes the required 68K files to Rez by using the `include` directive.

Writing Stand-Alone Code

Contents

Stand-alone code is commonly used to supplement the standard features provided by the Macintosh Toolbox and Operating System, to execute startup functions, or to control peripherals. It exists as a single Macintosh resource and consists almost entirely of executable object code. This chapter describes

- the characteristics of and limitations on stand-alone code
- the different kinds of stand-alone code
- how to build stand-alone code in the PowerPC and classic 68K runtime environments
- how to call stand-alone code from your C application

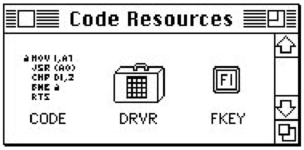
In addition, stand-alone code often poses special programming restrictions for classic 68K runtime code since it cannot rely on the services of the Segment Loader. This chapter includes information for circumventing limitations for 68K stand-alone code, including

- how to call QuickDraw routines from stand-alone code
- how to use global variables in stand-alone code
- how to design a stand-alone module that must persist across multiple invocations

Characteristics of Stand-Alone Code

Stand-alone code is program code that does not enjoy the full status of an application or shared library. A stand-alone code module exists as a single Macintosh code-type resource and consists almost entirely of microprocessor-executable object code and perhaps also some header data and other constants used by the executable portion of the module. Stand-alone code is often referred to as an **executable resource,** and a native PowerPC version is called an **accelerated resource.** Figure 11-1 shows how code-type resources are shown in the ResEdit resource picker.

Figure 11-1 Some ResEdit code-type icons



Most of these code-type resources are represented by an icon containing a stylized segment of assembly-language source code. Although `'CODE'` resources are not stand-alone code modules (they are segments of a larger application), they, too, contain executable code and so are represented by the stylized assembly-language icon. Driver resources are a special case of stand-alone code resources, and they have a different icon in the ResEdit resource picker, reminiscent of the Finder suitcase icon for a desk accessory, because the code of a desk accessory was once stored as a `'DRVR'` resource. The icon for an `'FKEY'` resource is also different, not surprisingly resembling a function key.

Types of Stand-Alone Code Resources

Table 11-1 presents an extensive (but not comprehensive) list of the currently defined code-type resources. Many are of interest primarily at a system software level. The stand-alone code resources most commonly created by application-level programmers are underlined.

Table 11-1 Stand-alone code resources

Name	Description
`'ADBS'`	Apple Desktop Bus service routine
`'adev'`	Link-Access Protocol (LAP) Manager code
`'CACH'`	RAM cache code
<u>`'CDEF'`</u>	Control definition function
<u>`'cdev'`</u>	Control panel file
`'CODE'`	Application code segment

continued

Table 11-1 Stand-alone code resources (continued)

Name	Description
'dcmd'	Extension to MacsBug command set
'DRVR'	Classic 68K device driver
'DSAT'	Startup alerts and code to display them
'FKEY'	Command-Shift-number combination keystroke
'FMTR'	3.5-inch disk formatting code
'INIT'	System extension
'itl2'	International Utilities sort hooks
'itl4'	Localizable tables and code
'LDEF'	List definition procedure
'MBDF'	Default menu definition procedure
'MDEF'	Menu definition procedure
'mntr'	Monitors control panel
'PACK'	Packages of code used as ROM extensions
'PDEF'	Code to drive printers
'ptch'	ROM patch code
'PTCH'	ROM patch code
'rdev'	Chooser code
'ROvr'	Code for overriding ROM resources
'RSSC'	ResEdit custom picker or editor
'SERD'	RAM serial driver
'snth'	Synthesizer or modifier
'WDEF'	Window definition function
'XCMD'	Extension to HyperCard command set
'XFCN'	Extension to HyperCard function set

Some of the code resource types shown in Table 11-1 supplement the standard features provided by the Mac OS. These resource types (`'WDEF'`, `'CDEF'`, `'MDEF'`, and `'LDEF'`) are used to define custom windows, controls, menus, lists, and responses to user input. In this respect, they serve particular parts of the Toolbox and very often are contained within the resource fork of an owner application.

Resources such as `'XCMD'` or `'dcmd'` are application extensions, which are loaded by the parent application (in this case, HyperCard and MacsBug, respectively).

Note
If you are using the PowerPC or CFM-68K runtime environment, you may want to use data fork–based drop-in additions in place of traditional application extensions. ◆

Resources of type `'INIT'`, `'cdev'`, and `'DRVR'` are examples of more autonomous stand-alone code. You can use them to write code that is executed automatically when the system starts up, code that adds special features to the Mac OS, or code that controls special-purpose peripherals and system functions. In addition, you are always free to define new resource types for custom stand-alone modules. The type `'CUST'` is commonly used, as shown in Listing 11-5 on page 11-13 and the examples in "Some Code Solutions for Classic 68K Stand-Alone Code," beginning on page 11-23.

Applications Versus Stand-Alone Code

During the launch process for an application, the Code Fragment Manager or (for classic 68K programs) the Segment Loader allocates space in memory to reference the application, the application's globals, and the QuickDraw globals. The method used to accomplish this depends on the runtime architecture:

- The PowerPC runtime architecture accomplishes this by setting up a TOC (Table of Contents) that is referenced by the RTOC (Table of Contents Register). Using the value in RTOC as a reference, the application can access global data and cross-fragment code. See *Inside Macintosh: PowerPC System Software* for specific information about the TOC.
- On 68K-based machines, an A5 world is set up to reference global data, jump table entries, and (in the CFM-68K runtime) cross-fragment code. See *Inside Macintosh: Memory* for more information about the A5 register and the A5 world.

PowerPC runtime stand-alone code is loaded much like any other fragment. The Code Fragment Manager loads and prepares it, so the code can access global variables normally through the TOC. In the classic 68K runtime environment, however, stand-alone code is simply loaded into memory and executed without any preparation. The Segment Loader is not used and no A5 world exists, making access to global variables problematic. See "Using Global Variables in Stand-Alone Code (Classic 68K Only)," beginning on page 11-14, for some possible solutions.

Stand-Alone Code—An Example

Listing 11-1 shows a very simple `'INIT'` code resource; it plays each of the sounds (resources of type `'snd '`) in the System file while the Macintosh computer boots. (If you want to try this out, be sure to name this file `SampleINIT.c` to work with the one of the makefiles in the next section. Make sure that all resources are unlocked and purgeable.) For the sake of simplicity, there are no references to global variables.

Listing 11-1 `SampleINIT` stand-alone code sample

```
#include <Resources.h>
#include <Sound.h>
#include <Types.h>

voidplayZoo(void);

voidplayZoo(void)
{
    short   count, index;
    Handle  sound;
    OSErr   status;

    count = CountResources('snd ');

    for (index = 1; index <= count; index++)
    {
        sound = GetIndResource('snd ', index);
        if (sound) status = SndPlay(NULL, (SndListHandle)sound, false);

    } // end for loop
}
```

The source code is very similar to that for an application. However, a stand-alone program does not contain a `main()` procedure, only a function.

Building Stand-Alone Code

The build procedure for stand-alone code is different from that for applications or shared libraries, and it also varies depending on the runtime architecture you are using. In both cases, however, the code ends up in a resource in the resource fork.

Building PowerPC Runtime Stand-Alone Code

On PowerPC-based Macintosh computers, some of the system software managers run 68K-based code in emulation rather than native PowerPC code. If you create a native PowerPC executable resource (an *accelerated resource*), it may be called by a system software manager that runs, and expects, 68K-based code. For this reason, you must attach a routine descriptor to the beginning of the resource. Note that a pointer to the routine descriptor is the form of universal procedure pointer required in the PowerPC environment. In the classic 68K environment, you could call it with a simple procedure pointer (the other form of universal procedure pointer). The routine descriptor ensures that the resource call will go through the Mixed Mode Manager (which can decide whether a mode switch is necessary). For more information about writing PowerPC-based executable resources and creating routine descriptors, see *Inside Macintosh: PowerPC System Software*.

To build an accelerated resource, you follow many of the same steps as you do when building a PowerPC runtime application. The following steps describe how to build an accelerated resource called `mooCdev`:

1. **Compile the source code.**

```
MrC mooCdev.c -o mooCdev.c.o
```

2. Link the object files.

```
PPCLink  ∂
        mooCdev.c.o ∂
        "{SharedLibraries}"InterfaceLib ∂
        -main mooProc ∂
        -o mooCdev
```

Note that you must indicate a main routine with the `-main` option.

3. Create a temporary resource.

```
Rez mooCdev.r1 -a -o mooCdev
```

This example assumes that the file `mooCdev.r1` contains the following line:

```
Read 'PWRC' (128) "mooCdev";
```

This statement defines a resource of type `'PWRC'` and assigns the data fork of `mooCdev` to be the data for the resource.

4. Attach the routine descriptor.

```
Rez mooCdev.r2 -a -o mooCdev
```

This example assumes that the contents of `mooCdev.r2` are as shown in Listing 11-2. This file redefines the resource type `'cdev'` (control panel device) to have the same syntax as the routine descriptor type `'rdes'`, as defined in `MixedMode.r`. The first field of the resource must give the value of the procedure information from the routine descriptor (type `ProcInfoType`). For more information on how to obtain this value, see *Inside Macintosh: PowerPC System Software.*

Listing 11-2 Attaching a routine descriptor to an accelerated resource

```
#include "MixedMode.r"
type 'cdev' as 'rdes';

resource 'cdev' (-4064)
{
    1, /* This must be the Mixed Mode Manager’s ProcInfo value*/
    $$ Resource ("mooCdev", 'PWRC', 128);
};
```

Listing 11-3 shows a makefile for building an accelerated resource. If you create the appropriate resource and routine descriptor files (`SampleINIT.r1` and `SampleINIT.r2`), you can use this makefile to build the `SampleINIT` example program.

Listing 11-3 Makefile for an accelerated resource

```
#VARIABLE DEFINITIONS

SrcName = sampleINIT
CdevName = {SrcName}

Objs =  {SrcName}.c.o ∂
        "{SharedLibraries}"InterfaceLib

#DEPENDENCIES

.c.o    ƒ .c
    MrC {default}.c -o {default}.c.o

{CdevName}  ƒƒ  {Objs} {Targ}.r1 {Targ}.r2 {CdevName}.make
    PPCLink {Objs} ∂
    -main playZoo ∂
    -o {Targ}
    Rez {Targ}.r1 {Targ}.r2 -a -o{Targ}
```

Building Classic 68K Stand-Alone Code

The build procedure for classic 68K stand-alone code is somewhat simpler than the PowerPC version, mostly because you can use the ILink linker to place the code in the resource instead of using Rez. Also, you do not need to worry about including a routine descriptor.

Note
Only ILink versions 2.1 or later can link stand-alone code. ◆

Note
Stand-alone code has no CFM-68K runtime equivalent, since it is more efficient to simply use a classic 68K version. Most stand-alone code resources patch or add to system-level routines that expect classic 68K conventions. Creating a CFM-68K stand-alone code fragment would therefore add the expense of having to call the Code Fragment Manager and the Mixed Mode Manager without gaining any significant benefits. ◆

The following steps describe how to build a classic 68K stand-alone code resource:

1. **Compile your source code using SC or SCpp.**

```
SC mooINIT.c -o mooINIT.o
```

2. **Link the object files with ILink.**

```
ILink mooINIT.o ∂
    -t INIT ∂
    -rt INIT=128 ∂
    -ra =resLocked ∂
    -m mooProc ∂
    -c 'MOOF' ∂
    -o mooINIT
```

The `-t` option indicates the type of resource you are building (in this case, a resource of type `'INIT'`).

The `-rt` option indicates the resource type and ID. Specifying this option also tells the ILink tool to edit `JSR`, `JMP`, `LEA`, or `PEA` instructions from A5-relative to PC-relative addressing mode.

The `-ra` option indicates the attributes you want for your resource (in this case, the resource is locked). You should always use this option to lock a resource of type `'INIT'`. An `'INIT'` resource is not automatically locked when loaded by the Operating System and could be moved during execution.

You can also use this option to load the `'INIT'` resource into the system heap. By default, an `'INIT'` resource is loaded into the application heap. This heap isn't safe for long-term storage because it is deallocated after the `'INIT'` resource is run. For complete information on setting resource attributes, see "Identifying a Resource and Setting Its Attributes," beginning on page 6-15.

You must use the `-m` option to indicate the main routine in your code.

Use the `-c` option if you want to indicate a creator.

You should also keep the following points in mind if you're writing an `'INIT'` resource:

- All resources from the `'INIT'` are disposed of when the file containing the `'INIT'` is closed. If an `'INIT'` wants to leave its resources around, they should be detached and moved to a safe place, such as the system heap, rather than into this temporary application heap.
- If an `'INIT'` resource needs large amounts of memory in the system heap, it should use a `'syzs'` resource to specify the amount of memory required.

Listing 11-4 shows a sample makefile for building a classic 68K `'INIT'` resource. You can use this makefile to build the `SampleINIT` stand-alone code example.

Listing 11-4 Makefile for a classic 68K executable resource

```
# VARIABLE DEFINITIONS

SrcName = SampleINIT
CdevName = {SrcName}

Objs = {SrcName}.c.o ∂

# DEPENDENCIES

.c.o ƒ .c
    SC {default}.c -o {default}.c.o

{CdevName} ƒƒ {CdevName}.make {Objs}
    ILink ∂
        {Objs} ∂
        -t INIT ∂
        -c '????' ∂
        -rt INIT=128 -ra =resLocked ∂
        -m PlayZoo ∂
        -o {Targ}
```

Calling Stand-Alone Code

This section explains how to call stand-alone code from a C program. The general procedure is as follows:

- Load the resource using the `GetResource` function. You might want to use the `Get1NamedResource` function instead to avoid resource numbering conflicts.
- Lock the resource. Include a procedure that calls the stand-alone code module and passes it the expected values (via parameters or a pointer to a parameter block). If you are writing PowerPC code, the dereferenced handle must be called indirectly through `CallUniversalProc`. Classic 68K code can make the call directly with a procedure pointer.
- Unlock the handle to the stand-alone code resource when you no longer need it.

Listing 11-5 shows the implemented procedure for PowerPC code.

Listing 11-5 Calling a stand-alone code module from PowerPC code

```
#include <Resources.h>
#include <Memory.h>
#include <MixedMode.h>

UniversalProcPtr myProcPtr;

void main (void)
{
        Handle      theHandle;
        struct MyParamBlock         param;
        theHandle = Get1NamedResource('CUST', "\pMySACModule");

        HLock(theHandle);

        /* Fill in the parameter block appropriately here. */
```

```
        myProcPtr = (UniversalProcPtr)*theHandle;
        CallUniversalProc(myProcPtr, kProcInfo, &param);
/* Note that the 68K call would simply be (*myProcPtr)(&param) */
        HUnlock(theHandle);
}
```

Using Global Variables in Stand-Alone Code (Classic 68K Only)

If you are writing classic 68K stand-alone code, you most likely have to address the problem of how to access global variables. References to global variables defined by a classic 68K stand-alone code module usually succeed without even a warning from the linker, but they also generally overwrite global variables defined by the current application.

References to global variables defined in the MPW libraries, like QuickDraw global variables, generate fatal link errors, as shown in the output to this ILink command:

```
ILink -t INIT -c '????' -rt INIT=128 -ra =resLocked -m PLAYZOO ∂
        SampleINIT.p.o ∂
        -o SampleINIT
# Undefined entry, name: "thePort"
# Referenced from "PLAYZOO" in file "SampleINIT.p.o"
#
ILink - Execution terminated!
```

In addition, because stand-alone code resources are not managed by the Segment Loader, they cannot be segmented into multiple resources like applications. Stand-alone code resources are restricted to 32 KB in size unless you use the `-bigseg` linker option. Keep in mind, however, that if a stand-alone code module gets much larger than 32 KB, it might be because it's trying to do too much. In general, stand-alone code should perform only simple and specific tasks.

The use of compiler and linker options to increase segments beyond their traditional 32 KB limit is helpful; it means you do not have to construct a jump table. However, the problem of using global variables and QuickDraw global variables or of calling a Toolbox routine that assumes the existence of a valid A5 world remains. The global requirements of stand-alone code vary, and there are a number of possible solutions. Some involve creating an A5 world and others do not. Independent stand-alone code such as `'INIT'` or `'DRVR'` resources requires the most work to access global variables, but other types are more forgiving.

If you are supplementing Toolbox or Mac OS routines (for example, using `'WDEF'`, `'CDEF'`, `'MDEF'`, and `'LDEF'` resources), you generally do not have to worry about sharing the calling application's global variables, since these resources are closely associated with a related manager. The call to the resource is usually implemented through a call to one of the managers, so you can pass a pointer to the resource as a parameter and let the manager worry about loading and unloading the segment. Such resource types might need to access QuickDraw global variables, however; the next section, "Referencing QuickDraw Global Variables," explains how this is done.

Stand-alone code resources used as application extensions, like HyperCard `'XCMD'` and `'XFCN'` resources or MacsBug `'dcmd'` resources, often receive support from the parent application in the form of predefined and convenient ways for defining global variables and for message passing between the application and the stand-alone code extension. The section "Extensible Applications" on page 11-17 describes how these mechanisms are implemented in ResEdit, HyperCard, and MacsBug.

Referencing QuickDraw Global Variables

Often a stand-alone code segment needs the QuickDraw global variables of the current application for which it is performing a service. For example, the drawing operations of a `'CDEF'` resource assume a properly initialized QuickDraw world, which is conveniently provided by the application. Most QuickDraw calls are supported and no special effort is required. One limitation, however, is that explicit references to QuickDraw global variables like `thePort` and `screenBits` are not allowed. The ILink tool cannot resolve the offsets to these variables because it links a `'CDEF'` resource (or any other stand-alone module) independently of a particular application. The solution, which involves allocating a record in the heap and copying the QuickDraw global variables into the record, is shown in Listing 11-6.

Listing 11-6 Making a local copy of QuickDraw global variables

```
UNIT GetQDGlobals;
INTERFACE
    USES
        Types, QuickDraw, OSUtils;
    TYPE
        QDVarRecPtr  =  ^QDVarRec;
        QDVarRec  =  RECORD
            randSeed   : Longint;
            screenBits : BitMap;
            arrow      : Cursor;
            dkGray     : Pattern;
            ltGray     : Pattern;
            gray       : Pattern;
            black      : Pattern;
            white      : Pattern;
            thePort    : GrafPtr;
        END;
    PROCEDURE GetMyQDVars (VAR qdVars: QDVarRec);
    IMPLEMENTATION
        PROCEDURE GetMyQDVars (VAR qdVars: QDVarRec);
        TYPE
            LongPtr = ^Longint;
        BEGIN
            qdVars := QDVarRecPtr(LongPtr(SetCurrentA5)^
            - (SizeOf(QDVarRec)-SizeOf(thePort)))^;
        END;
    END.
```

The unit `GetQDGlobals` allocates space for a record in the heap and then uses the following assignment to copy the QuickDraw global variables into the record.

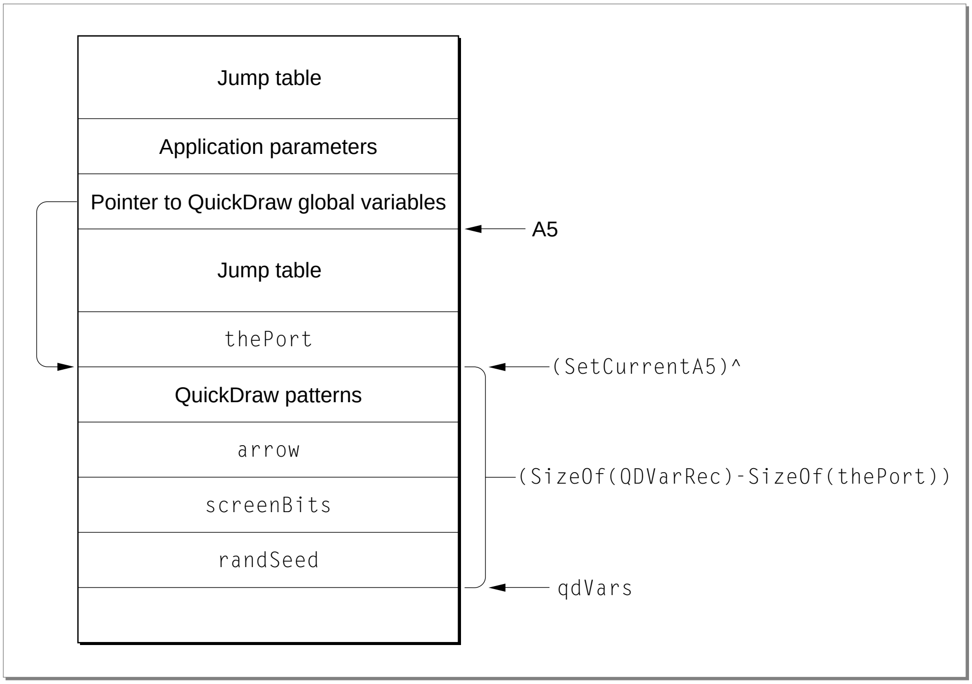
```
qdVars := QDVarRecPtr(LongPtr(SetCurrentA5)^
    - (SizeOf(QDVarRec)-SizeOf(thePort))^;
```

The calculation performed on the right side of the assignment

- returns the current value of A5 with `SetCurrentA5`
- dereferences that value to get the address of `thePort`
- performs arithmetic to determine the address of `randSeed`

Figure 11-2 illustrates this calculation.

Figure 11-2 Calculating address of `randSeed`



Extensible Applications

Some applications are intended to be extensible and provide special support for stand-alone code segments. ResEdit, for instance, uses `'RSSC'` code resources to provide support for custom resource pickers and editors. If you need a graphical editor to edit a custom resource type, such as an 8-by-64 pixel icon, you can paste separately compiled and linked extension code directly into the application's resource fork. ResEdit defines interfaces through which it communicates with these resources. In many cases, this degree of support and message passing can preempt the need to declare global variables at all. The MacsBug `'dcmd'` resource is another instance of extension code with support for global variables built in. A `'dcmd'` resource specifies in its header how much space it needs for global variables, and MacsBug makes room for them.

HyperCard is another application that provides high-level support for its `'XCMD'` and `'XFCN'` extension resources. Callback routines like `SetGlobal` and `GetGlobal` provide extension code with a convenient mechanism for defining variables that are global in scope, yet without requiring the deadly A5-relative references normally associated with global variables. The HyperCard interfaces are included in the Interfaces folder. Pascal programmers should use the `HyperXCmd.p` interface file; C programmers should use the `HyperXCmd.h` header file.

In these cases, where an application provides special support for extensions, you should take advantage of this support as much as possible. Things can get complicated quickly when no support for global variables is provided or when built-in support is not used. You should not use the techniques used to build an A5 world, described in the following section, unless absolutely necessary. Also, when writing an application, you may want to consider supporting extension modules. With the move toward object-oriented programming and reusable code, demand for extension module support is growing. Support for extension modules can rarely be tacked on as an afterthought, and it is worth looking at how ResEdit, HyperCard, and Apple File Exchange support modular code when considering similar features.

Building an A5 World

There are cases where building an A5 world is unavoidable. Consider the following examples:

- A stand-alone module consists of two functions. There is one main entry point, and one function calls another function in the process of calculating its final result. Instead of passing a formal parameter to the subordinate function, the programmer chooses to use a global variable.
- A stand-alone module consists of one function. The module is loaded into memory once and invoked multiple times by the host application. The module requires its own private storage to persist across multiple invocations.
- A complex `'INIT'` resource uses QuickDraw, or a `'cdev'` resource is complex enough to require an application-like set of global variables to accomplish its self-contained task. A module might need to access data in a Toolbox callback (like a dialog hook) where the interface is fixed, for instance.

The following sections describe the routines you need to call to build an A5 world and explain how you call a stand-alone code module from an application. You need to understand this material before moving on to the sections that discuss specific solutions to the problems described in the preceding bulleted list.

The SAGlobals Unit

Building an A5 world would seem to be fairly complicated, but most of the necessary code is already written. Much of it is in the MPW `MacRuntime.o` library. What's not in the MPW library is the initial allocation of space for an A5 world. For an application, this is done by the Segment Loader. A stand-alone module can emulate the entire process by using glue code around calls to the appropriate routines in `MacRuntime.o`. This glue code is furnished by the `SAGlobals` unit shown in Listing 11-7 on page 11-21. `SAGlobals` makes it very easy to use global variables in stand-alone code because it automates the process of allocating space for global variables and initializes them the same way an application would. describes the routines included in `SAGlobals`. Stand-alone code modules that need to use global variables can include the interfaces in this unit. You must link these code modules with `MacRuntime.o` and `SAGlobals.o`.

Table 11-2 `SAGlobals` routines

Routine	Description and declaration
`MakeA5World`	Allocates space for an A5 world based on the size of the global variables defined by the module and its units. The procedure returns a handle to the A5 world in the `A5Ref VAR` parameter. If sufficient space is not available, `A5Ref` is set to `NIL` and further initialization is aborted. `PROCEDURE MakeA5World (VAR A5Ref: A5RefType);`
`SetA5World`	Locks down the handle allocated with `MakeA5World` and sets the A5 register appropriately. The function return value is the old value of A5, which should be saved for use by `RestoreA5World`. `FUNCTION SetA5World (A5Ref: A5RefType) : LongInt;`

continued

Table 11-2 `SAGlobals` routines (continued)

Routine	Description and declaration
`RestoreA5World`	Restores A5 to its original value (which should have been saved) and unlocks the A5 world to avoid heap fragmentation in case the A5 world is used again. `PROCEDURE RestoreA5World (oldA5: LongInt; A5Ref: A5RefType);`
`DisposeA5World`	Disposes of the A5 world handle. `PROCEDURE DisposeA5World (A5Ref: A5RefType);`
`OpenA5World`	Combines the action of `MakeA5World` and `SetA5World` for those cases where the routines are called consecutively. For A5 worlds that must persist across different invocations, use `MakeA5World` once and invoke it each time with `SetA5World`. Or call `OpenA5World` at the beginning and `CloseA5World` at the end. `FUNCTION OpenA5World (VAR A5Ref: A5RefType) : LongInt;`
`CloseA5World`	Corresponds to `OpenA5World`; it combines the action of `RestoreA5World` and `DisposeA5World`. In some cases, it is necessary to call these two explicitly. `PROCEDURE CloseA5World (oldA5: LongInt; A5Ref: A5RefType);`

How SAGlobals Does Its Work

The `SAGlobals` unit, shown in Listing 11-7, uses two MPW library routines to set up and initialize an A5 world.

- `A5Size` determines how much memory is required for the A5 world. This memory consists of two parts: memory for global variables and memory for application parameters.
- `A5Init` takes a pointer to the A5 global variables and initializes them to the appropriate values.

Listing 11-7 The `SAGlobals` unit

```
UNIT SAGlobals;
INTERFACE
    USES
        Types, Memory, OSUtils;
    TYPE
        A5RefType = Handle;

    PROCEDURE MakeA5World (VAR A5Ref: A5RefType);
    FUNCTION SetA5World (A5Ref: A5RefType) : LongInt;
    PROCEDURE RestoreA5World (oldA5: LongInt; A5Ref: A5RefType);
    PROCEDURE DisposeA5World (A5Ref: A5RefType);
    FUNCTION OpenA5World (VAR A5Ref: A5RefType) : LongInt;
    PROCEDURE CloseA5World (oldA5: LongInt; A5Ref: A5RefType);

IMPLEMENTATION

    CONST
        kAppParmsSize = 32;

    FUNCTION A5Size : Longint;
        C;  EXTERNAL;    {in Runtime.o}

    PROCEDURE A5Init (myA5: Ptr);
        C;  EXTERNAL;    {in Runtime.o}

    PROCEDURE MakeA5World (VAR A5Ref: A5RefType);
    BEGIN
        A5Ref := NewHandle(A5Size);
        {The calling routine must check A5Ref for NIL!}
        IF A5Ref <> NIL THEN
            BEGIN
                HLock(A5Ref);
                A5Init(Ptr(Longint(A5Ref^) + A5Size - kAppParmsSize));
                HUnlock(A5Ref);
            END;
        END;
```

```
    FUNCTION SetA5World (A5Ref: A5RefType) : LongInt;
        BEGIN
            HLock(A5Ref);
            SetA5World := SetA5(LongInt(A5Ref^) + A5Size - 
kAppParmsSize);
        END;

    PROCEDURE RestoreA5World (oldA5: LongInt; A5Ref: A5RefType);
        BEGIN
        IF Boolean (SetA5(oldA5)) THEN;    {side effect only}
        HUnlock(A5Ref);
    END;

    PROCEDURE DisposeA5World (A5Ref: A5RefType);
        BEGIN
            DisposHandle(A5Ref);
        END;

    FUNCTION OpenA5World (VAR A5Ref: A5RefType) : LongInt;
        BEGIN
            MakeA5World(A5Ref);
            IF A5Ref <> NIL THEN
                OpenA5World := SetA5World(A5Ref)
            ELSE
                OpenA5World := 0;
        END;

    PROCEDURE CloseA5World (oldA5: LongInt; A5Ref: A5RefType);
        BEGIN
            RestoreA5World(oldA5, A5Ref);
            DisposeA5World(A5Ref);
    END;
END.
```

When MPW links an application together, it has to describe what the global variable area should look like. At the very least, it needs to keep track of how large the global variable section should be. In addition, it might need to specify what values to put into the global variable area. Normally, this means setting everything to 0, but some languages like C allow specification of preinitialized global variables. The linker normally creates a packed data block that describes all of this and places it into a segment called `%A5Init`. Also included in this segment are the routines called by the MPW runtime initialization package to act upon this data. `A5Size` and `A5Init` are two such routines. `A5Size` looks at the field that holds the unpacked size of the data and returns it to the caller. `A5Init` is responsible for unpacking the data into the global variable section. In the case of a stand-alone module, all code and data needs to be packed into a single segment or resource, so `%A5Init` is not used. The `-sg` option to the ILink command is used to make sure that everything is in the same resource. The Commando interface to the `CreateMake` command is very good about specifying this automatically when you select the Code Resource button for program type, but you must remember to specify this option if you create your own makefiles.

The rest of the `SAGlobals` unit is mostly self-explanatory. The calls to the Memory Manager allocate the amount of space indicated by `A5Size` and lock the handle down when in use by the module. The calculation performed by `MakeA5World` and `SetA5World` is required to set A5 to point to the boundary between the global variables and the application parameters. Since the application parameters, including the pointer to QuickDraw global variables, are 32 bytes long, the calculation is

address-stored-in-A5 = *starting-address* + *block-length* – 32

As demonstrated in the examples in the next section, a module can simply call `MakeA5World` to begin building its own A5 world, and it can call `SetA5World` to invoke it and make it active. In addition, the module should check `A5Ref` to see if it is `NIL`. If so, there is not enough space to allocate the A5 world, and the module needs to abort gracefully or find another way of getting its job done. Also, the programmer should be aware that `A5Ref` is *not* an actual A5 value. As its name implies, it is a reference to an A5 world.The actual value of A5 is calculated whenever that world is invoked, as described in the preceding paragraph.

Some Code Solutions for Classic 68K Stand-Alone Code

The following three sections demonstrate solutions for those situations where you might want to build an A5 world when writing classic 68K stand-alone code. For each example, the source code for the stand-alone module, for the calling program, and for the makefiles is shown.

Example 1—Using Global Variables in Stand-Alone Code

`LazyPass` is a stand-alone module that implements the function of determining a circle's area from its circumference. The unit consists of two functions. The `CircleArea` function is the main entry point; it calls the function `RadiusSquared` in the process of calculating its final result. Instead of passing a formal parameter to the subordinate function, `CircleArea` uses the global variable `radius`.

When `LazyPass` is executed, it creates an A5 world, does its job, and then disposes of the A5 world, making sure to restore the host application's world. Listing 11-8 shows the source code for `LazyPass`. Pay special attention to the underlined items. These demonstrate the use of routines that build an A5 world and use global variables in stand-alone code.

Listing 11-8 Example module `LazyPass.p`

```
UNIT LazyPass;

INTERFACE
    USES
        ypes, SAGlobals;

    FUNCTION CircleArea (circumference: Real) : Real;

IMPLEMENTATION

{Define a variable global to all routines in this unit.}
    VAR radius : Real;

    FUNCTION RadiusSquared : Real;
        FORWARD;
{Define CircleArea first to place entry point}
{at the beginning of the module.  }
    FUNCTION CircleArea (circumference: Real) : Real;
```

```
VAR
    A5Ref: A5RefType;
    oldA5: Longint;
BEGIN
    oldA5 := OpenA5World(A5Ref);
        radius := circumference / (2.0 * Pi);
        CircleArea := Pi * RadiusSquared;
    CloseA5World(oldA5, A5Ref);
END;

FUNCTION RadiusSquared : Real;
    BEGIN
        RadiusSquared := radius * radius;
    END;
END.
```

Listing 11-9 shows the makefile for the `LazyPass` module.

Listing 11-9 Makefile for `LazyPass`

```
    #   File:       LazyPass.make
    #   Target:     LazyPass
    #   Sources:    LazyPass.p
    OBJECTS = LazyPass.p.o

LazyPass ƒƒ LazyPass.make {OBJECTS}
    ILink -w -t '????' -c '????' -rt CUST=128 ∂
-m CIRCLEAREA -sg LazyPass ∂
    {OBJECTS} ∂
    "{Libraries}"MacRuntime.o ∂
    "{Libraries}"Interface.o ∂
    "{PLibraries}"SANELib.o ∂
    "{PLibraries}"PasLib.o ∂
    "{MyLibraries}"SAGlobals.o ∂
    -o LazyPass
LazyPass.p.o ƒ LazyPass.make LazyPass.p
    Pascal -i "{MyInterfaces}" LazyPass.p
```

The `LazyTest` file, shown in Listing 11-10, is a very simple program that shows how to load and call the `LazyPass` stand-alone module. Things to watch out for are standard I/O (`ReadLn` and `WriteLn`) and error checking (or lack thereof).

Listing 11-10 Example testing program `LazyTest.p`

```
PROGRAM LazyTest;
USES
    Types, Resources, Memory, OSUtils;

VAR
    a, c: Real;
    h1: Handle;

FUNCTION CallModule (parm: Real; modHandle: Handle) : Real;
    INLINE $205F,  {pop handle off stack}
        $2050,  {dereference to get address of 'CUST' 128}
        $4E90;  {call LazyPass, leaving variable on stack}

BEGIN
    Write('Circumference:');
    ReadLn(c);

    h1 := GetResource('CUST',128);                      {load resource}
        HLock(h1);                                      {lock resource}
        a := CallModule(c,h1);                          {Call LazyPass}
        HUnlock(h1);                                    {unlock resource}

    WriteLn('Area: ',a);
END.
```

Listing 11-11 shows the makefile for `LazyTest`. Note the directive used to include the `LazyPass` module in the final application. This avoids the need to paste `LazyPass` into the application manually with a resource editor. It is also an example of a very powerful feature of the MPW scripting language, which allows the output of one command to be piped into the input of another.

Listing 11-11 Makefile for `LazyTest`

```
    #   File:       LazyTest.make
    #   Target:     LazyTest
    #   Sources:    LazyTest.p

             OBJECTS = LazyTest.p.o

    LazyTest ƒƒ LazyTest.make LazyPass
        Echo 'Include "LazyPass";' | Rez -o LazyTest -a

LazyTest ƒƒ LazyTest.make {OBJECTS}
    ILink -w -t APPL -c '????' ∂
        {OBJECTS} ∂
        "{Libraries}"MacRuntime.o ∂
        "{Libraries}"Interface.o ∂
        "{PLibraries}"SANELib.o ∂
        "{PLibraries}"PasLib.o ∂
        -o LazyTest
    LazyTest.p.o ƒ LazyTest.make LazyTest.p
        Pascal  LazyTest.p
```

Example 2—Stand-Alone Code That Maintains Its State Across Multiple Invocations

In the example shown in Listing 11-12, the stand-alone module `Persist` consists of one function. The function maintains a running total of the squares of the parameters it receives from the host application. The module is loaded into memory once and invoked multiple times by the host application, `PersistTest`.

The `Persist` module requires its own private storage to persist across multiple invocations by `PersistTest`. The module must also pass a reference to its global variable storage (A5 world) back to the application so that it can be easily restored the next time the module is invoked. `Persist` uses the `message` parameter to receive messages from the host application.

Underlining indicates those statements that are required to resolve the construction of an A5 world, global declarations, and message passing.

Listing 11-12 Example module `Persist.p`

```
UNIT Persist;

INTERFACE

    USES
        Types, SAGlobals;

    CONST
        kAccumulate = 0;  {These are the control messages.}
        kFirstTime = 1;
        kLastTime = 2;

    FUNCTION AccSquares (parm: LongInt; message: Integer;
        VAR A5Ref: A5RefType) : LongInt;

IMPLEMENTATION
    {Accumulation global used to retain a running}
    {total over multiple calls to the module.}
    VAR accumulation : LongInt;

        FUNCTION AccSquares (parm: LongInt; message: Integer;
            VAR A5Ref: A5RefType) : LongInt;
    VAR
        oldA5: LongInt;
    BEGIN
        IF message = kFirstTime THEN MakeA5World(A5Ref);
        oldA5 := SetA5World(A5Ref);
            IF message = kFirstTime THEN accumulation := 0;
            accumulation := accumulation + (parm * parm);
            AccSquares := accumulation;
        RestoreA5World(oldA5, A5Ref);
        IF message = kLastTime THEN DisposeA5World(A5Ref);
    END;
END.
```

Listing 11-13 shows the makefile for the `Persist` module.

Listing 11-13 Makefile for `Persist`

```
        #   File:       Persist.make
        #   Target:     Persist
        #   Sources:    Persist.p

    OBJECTS = Persist.p.o

    Persist ƒƒ Persist.make {OBJECTS}
        ILink -w -t '????' -c '????' -rt CUST=129 -m ACCSQUARES ∂
            -sg Persist ∂
        {OBJECTS} ∂
        "{Libraries}"MacRuntime.o ∂
        "{Libraries}"Interface.o ∂
        "{PLibraries}"SANELib.o ∂
        "{PLibraries}"PasLib.o ∂
        "{MyLibraries}"SAGlobals.o ∂
        -o Persist
    Persist.p.o ƒ Persist.make Persist.p
        Pascal -i "{MyInterfaces}" Persist.p
```

`PersistTest`, shown in Listing 11-14, is a bare-bones application that demonstrates how the host application calls the `Persist` module. `PersistTest` uses the `message` parameter to tell the module when to initialize and when to end. It also maintains a handle to the module's A5 world between invocations.

Listing 11-14 Example test program `PersistTest.p`

```
PROGRAM PersistTest;
USES
    Types, Resources, Memory, OSUtils;

CONST
    N = 5;
    kAccumulate = 0;    {these are the control messages}
    kFirstTime = 1;
    kLastTime = 2;
```

```
VAR
    i : Integer;
    acc : LongInt;
    h1, otherA5: Handle;

    FUNCTION CallModule (parm: LongInt; message: Integer; VAR otherA5:
            Handle; modHandle: Handle) : LongInt;
        INLINE $205F,    {pop handle off stack}
               $2050,    {dereference to get address of loaded resource}
               $4E90;    {call Persist, leaving variables on the stack}

BEGIN
    h1 := GetResource('CUST',129);
    MoveHHi(h1);
    HLock(h1);

    FOR i := 1 TO N DO
        BEGIN
            CASE i OF
                1: acc := CallModule(i,kFirstTime,otherA5,h1);
                N: acc := CallModule(i,kLastTime,otherA5,h1);
            OTHERWISE
                acc := CallModule(i,kAccumulate,otherA5,h1);
            END;
            WriteLn('SumSquares after ',i,' = ',acc);
        END;
    HUnlock(h1);
END.
```

Listing 11-15 shows the makefile for `PersistTest`.

Listing 11-15 Makefile for `PersistTest`

```
    #   File:       PersistTest.make
    #   Target:     PersistTest

    #   Sources:    PersistTest.p

OBJECTS = PersistTest.p.o
```

```
PersistTest ƒƒ PersistTest.make Persist
    Echo 'Include "Persist";' | Rez -o PersistTest -a

PersistTest ƒƒ PersistTest.make {OBJECTS}
    ILink -w -t APPL -c '????' ∂
        {OBJECTS} ∂
        "{Libraries}"MacRuntime.o ∂
        "{Libraries}"Interface.o ∂
        "{PLibraries}"SANELib.o ∂
        "{PLibraries}"PasLib.o ∂
        -o PersistTest
PersistTest.p.o ƒ PersistTest.make PersistTest.p
    Pascal  PersistTest.p
```

Example 3—Stand-Alone Code That Calls Toolbox Managers

The next stand-alone code example is a complex `'INIT'` resource that uses arbitrary Toolbox managers to present a user interface. It is also the first example in which a stand-alone code resource uses other resources.

The `StopBoot` module, shown in Listing 11-16, might look a bit familiar because it performs the same function as the sample `'INIT'` resource shown in Listing 11-1 on page 11-7. However, it has the added feature of providing a dialog box during the startup sequence.

In general, an `'INIT'` resource can simply call `OpenA5World` on entry and `CloseA5World` before exiting. Everything between can then be just like an application: `InitGraf`, `InitWindows`, and so on. An `'INIT'` resource should be careful, though, to restore the graphics port (`GrafPort`) to its initial value before exiting.

Listing 11-16 Example module `StopBoot.p`

```
UNIT StopBoot;

INTERFACE

    USES
        Types, SAGlobals, OSUtils,
        QuickDraw, Fonts, Windows, Menus, TextEdit, Dialogs,
        Resources, Sound, ToolUtils;
```

```
    PROCEDURE BeAPest;

IMPLEMENTATION

    PROCEDURE BeAPest;
        CONST
            kStopBootDLOG = 128;
        VAR
            A5Ref: A5RefType;
            oldA5: LongInt;
            numSnds, i, itemHit: Integer;
            theSnd: Handle;
            playStatus: OSErr;
            orwell: DialogPtr;

{PROCEDURE ClearDeskHook;                                 If the code is to run
INLINE $42B8, $0A6C                                       on a Mac Plus, SE, or
                                                          Classic®, uncomment this code
PROCEDURE ClearDragHook                                   to include these declarations
INLINE $42B8, $09F6                                       and avoid a crash}

    BEGIN
        IF NOT Button THEN BEGIN
            oldA5 := OpenA5World(A5Ref);
                IF A5Ref <> NIL THEN BEGIN
                InitGraf(@thePort);
                InitFonts;
                InitWindows;
                InitMenus;
                TEInit;
                InitDialogs(NIL);
                {ClearDeskHook;                           Uncomment this code if running
                ClearDragHook;                            on Mac Plus, SE, or Classic}
                InitCursor;
                orwell := GetNewDialog(kStopBootDLOG, NIL, WindowPtr(-1));
                numSnds := CountResources('snd ');

                FOR i := 1 TO numSnds DO BEGIN
                    theSnd := GetIndResource('snd ',i);
                    IF theSnd <> NIL THEN
                        playStatus := SndPlay(NIL,theSnd,FALSE);
```

```
            END;
         REPEAT
            ModalDialog(NIL, itemHit);
         UNTIL itemHit = 1;
         DisposDialog(orwell);
      CloseA5World(oldA5, A5Ref);
   END;
  END;
 END;
END.
```

Listing 11-17 shows the Rez input file used to create the `'DLOG'` and `'DITL'` resources used by the `StopBoot` module.

Listing 11-17 Resource description file that creates a dialog box

```
resource 'DLOG' (128) {
    {84, 124, 192, 388},
    dBoxProc,
    visible,
    noGoAway,
    0x0,
    128,
    ""
};

resource 'DITL' (128) {
    {  /* array DITLarray: 2 elements */
        /* [1] */
        {72, 55, 93, 207},
        Button {
            enabled,
            "Continue Booting"
        },
    /* [2] */
        {13, 30, 63, 237},
        StaticText {
            disabled,
            "This is an exaggerated case of the type "
            "of INIT that bothers me more than anything else."
```

```
        }
    }
};
```

The makefile for `StopBoot`, shown in Listing 11-18, includes a Rez command that is used to include the dialog box resources. The makefile uses two MPW Shell variables, `{MyInterfaces}` and `{MyLibraries}`, that represent the directories containing the `SAGlobals` headers and library, respectively. If you are following along with these examples, you would need to define these Shell variables, possibly in the `UserStartup` file, or to replace the occurrences with the name of whatever directory actually contains the necessary `SAGlobals` files.

Listing 11-18 Makefile for `StopBoot`

```
OBJECTS = StopBoot.p.o
StopBoot ƒƒ StopBoot.make StopBoot.r
    Rez -o StopBoot "{RIncludes}"Types.r StopBoot.r

StopBoot ƒƒ StopBoot.make {OBJECTS}
    ILink -w -t INIT -c '????' -rt INIT=128 -ra =resLocked ∂
              -m BEAPEST -sg StopBoot ∂
        {OBJECTS} ∂
        "{Libraries}"MacRuntime.o ∂
        "{Libraries}"Interface.o ∂
        "{PLibraries}"SANELib.o ∂
        "{PLibraries}"PasLib.o ∂
        "{MyLibraries}"SAGlobals.o ∂
        -o StopBoot
StopBoot.p.o ƒ StopBoot.make StopBoot.p
    Pascal -i "{MyInterfaces}" StopBoot.p
```

Building PCI Device Drivers

Contents

This chapter describes how to build PCI (Peripheral Component Interconnect) device drivers for PowerPC-based Macintosh computers. PCI drivers require a native PowerPC version of the Device Manager, which allows much more flexibility when writing drivers.

Currently there are five device driver families:

- block (file type `'blok'`)
- display/video (file type `'disp'`)
- generic native (file type `'ndrv'`; most drivers fall into this category)
- Open Transport (file type `'otan'`)
- SCSI (file type `'scsi'`)

More device driver families may be defined in the future. Each driver family requires that you link with a slightly different set of object files, but the general build procedure is the same.

All device drivers may contain three levels of routines that are more or less independent of each other:

- The hardware interrupt level. Code execution at this level includes installable interrupt handlers for PCI and interrupt handlers provided by Apple. Hardware interrupt-level execution happens as a direct result of an interrupt request.
- The secondary interrupt level. This level is analogous to the deferred task execution level. The secondary interrupt queue is filled with requests to execute subroutines that are posted for execution by the hardware interrupt handlers. The handlers need to perform certain actions but choose to defer the execution of those actions to minimize interrupt-level execution. Unlike hardware-interrupt handlers, which can nest, secondary interrupt handlers always run serially.
- The task level. This is the noninterrupt level that connects the device driver to the Mac OS.

The architecture for PCI drivers keeps these components separate, making it easier to write the actual driver. You can access the driver only through a small collection of driver library routines, and driver code cannot access traditional Toolbox routines.

Note
Older drivers still function under the native Device Manager. However, they cannot take full advantage of the PowerPC native code. ◆

Note that this chapter does not describe how to write PCI device drivers. For that information, you should read *Designing PCI Cards and Drivers for Power Macintosh Computers*. You may also want to review *Inside Macintosh: Devices* for general information about building device drivers.

Building a PCI Driver

One advantage of the PowerPC runtime environment is that the Code Fragment Manager handles all the code regardless of its type. PCI drivers are built as shared libraries, so they have free access to global variables and may be called from other fragments. The following steps describe how to build a generic PCI driver of type `'ndrv'`.

1. **Compile the code using MrC or MrCpp.**

```
MrC mooDriv.c -o mooDriv.c.o
```

   You should make sure that that proper PCI include files are listed in your code. Otherwise you can use the `-i` option to include the proper headers when compiling.

2. **Link the object files and libraries using PPCLink.**

```
PPCLink ∂
        mooDriv.c.o ∂
        "{PPCLibraries}"PPCCRuntime.o ∂
        "{DriverLibraries}"DriverServicesLib ∂
        "{DriverLibraries}"DriverLoaderLib ∂
        "{DriverLibraries}"NameRegistryLib ∂
        "{DriverLibraries}"PCILib ∂
        -export TheDriverDescription ∂
        -export DoDriverIO ∂
        -xm s ∂
        -main DoDriverIO ∂
        -fragname mooDriver ∂
        -t 'ndrv' ∂
        -c 'MOOF' ∂
        -o mooDriv
```

Note the absence of `InterfaceLib`. PCI drivers do not access the Toolbox. They interact with the operating system only through the driver libraries.

The four driver shared libraries (`DriverServicesLib`, `DriverLoaderLib`, `NameRegistryLib`, and `PCILib`) are required when building a PCI driver. Depending on the type of driver you are building, you may need to add other libraries (for example, `OpenTptModuleLib` for Open Transport drivers or `VideoServicesLib` for video drivers).

All drivers must export the symbol `TheDriverDescription`. Depending on the type of driver you are writing, you may need to export other symbols as well. (In this case, all `'ndrv'` driver types must export `DoDriverIO`.)

The `-xm s` option specifies that the driver takes the form of a shared library.

In `'ndrv'` driver types, `DoDriverIO` is the main entry point, which is indicated by the `-main` option.

The `-fragname` option specifies the name of the driver fragment.

The `-t` option specifies the driver type.

The `-c` option indicates the creator.

The `-o` option specifies the name of the output file.

3. **Append resources to the driver file using Rez.**

```
Rez mooDriv.r -append -o mooDriv
```

The completed driver can now be loaded onto the expansion ROM of a PCI card or else stored as a file in the Extensions folder.

Listing 12-1 shows a sample makefile for building a PCI driver that you can use as a starting template for your own makefile.

Listing 12-1 Makefile for a PCI driver

```
# Makefile to build mooDriv from    mooDriv.c
                                    mooDriv.h
                                    mooDriv.r

# VARIABLE DEFINITIONS

# Define object files that PPCLink will combine.
Objects = mooDriv.c.o
```

```
# Define the exported symbols for PPCLink.
Exports =   TheDriverDescription ∂
            DoDriverIO

# Define the standard libraries to link with.
PPCLibs =   "{PPCLibraries}"PPCCRuntime.o ∂
            "{DriverLibraries}"DriverServicesLib ∂
            "{DriverLibraries}"DriverLoaderLib ∂
            "{DriverLibraries}"NameRegistryLib ∂
            "{DriverLibraries}"PCILib


# DEFAULT BUILD RULE

all ƒ mooDriv

# COMPILE DEPENDENCIES

mooDriv.c.o ƒ mooDriv.c mooDriv.h
     MrC mooDriv.c -o mooDriv.c.o

# TARGET DEPENDENCIES
# Note that the driver is being built as a shared library.

mooDriv ƒƒ mooDriv.make  {Objects} {PPCLibs}
    PPCLink -o mooDriv ∂
        {Objects} ∂
        {PPCLibs} ∂
        -export {Exports} ∂
        -xm s ∂
        -main DoDriverIO
        -fragname mooDriver ∂
        -t 'ndrv' ∂
        -c 'MOOF'

mooDriv ƒƒ mooDriv.make mooDriv.r
    Rez mooDriv.r -append -o mooDriv
```

Writing and Building MPW Tools

Contents

A tool is a program that runs in the MPW Shell environment and has access to the facilities provided by the MPW Shell. The compilers, linkers, Make, and so on are all tools provided with the MPW development system. You can write your own tools to extend the functionality of MPW or to test numeric or file-oriented algorithms that you can then integrate into an event-driven application.

This chapter provides information about writing an integrated MPW tool. It describes

- the characteristics of programs that run as tools
- the conventions for the behavior of MPW tools
- the runtime environment of MPW tools
- the utility, alias-resolution, signal-handling, cursor-control, and error-message routines used by tools and how you access these from C and assembly language

This chapter assumes you are familiar with the compile and link process for the runtime architecture you intend to build for. In addition, you should read Chapter 14, "Creating Commando Dialog Boxes for Tools and Scripts," for information on how to write a `'cmdo'` resource, which is used to display a Commando dialog box for a tool or script.

Overview

This section describes how you link a tool, what are the characteristics and expected behavior of tools that run in the MPW Shell, and how tools share MPW's runtime environment.

From a programming viewpoint, tools resemble applications in many aspects of their behavior. Like applications, tools can have global variables, and you link tools just as you link applications. The major differences between tools and applications are that tools do not have to initialize their environment (except for QuickDraw, if used) and that tools have access to any of the MPW Shell's open windows.

Linking a Tool

Linking your MPW tool is the same as linking an application, except that you must use the linker's `-t` option to set the file type to `'MPST'` and the linker's `-c` option to set the creator to `'MPS '`. For example,

```
PPCLink -t MPST  -c 'MPS ' ...
```

Listing 13-1 shows the commands required to build the sample tool Count in the PowerPC runtime environment. Source files for this tool are in the CExamples folder.

Listing 13-1 Building the Count tool

```
MrC Count.c -o Count.c.o
PPCLink Count.c.o ∂
    "{SharedLibraries}"StdCLib ∂
    "{SharedLibraries}"InterfaceLib ∂
    "{PPCLibraries}"StdCRuntime.o ∂
    "{PPCLibraries}"PPCRuntime.o ∂
    -c 'MPS ' ∂
    -t MPST ∂
    -o Count
Rez Count.r -o Count -append
```

In general, tools do not need to be linked with any special libraries. However, if you are building a classic 68K tool, you should link with `Stubs.o`; this file contains dummy library routines used to override standard library routines that are not used by MPW tools, thus reducing the tool's code size. `Stubs.o` must be the first library you link with.

If you use the signal-handling routines, cursor-control routines, or the MPW error-management routines described at the end of this chapter, you must link with the appropriate ToolLibs library (`PPCToolLibs.o` or `ToolLibs.o`).

The sections "Accessing the MPW Shell—C" on page 13-15 and "Accessing the MPW Shell—PowerPC and 68K Assembly Language" on page 13-16 provide additional information on the specific library files you need to include in your link.

Conventions for the Behavior of MPW Tools

MPW tools observe certain conventions that allow them to work well together in an integrated fashion. The tool you design should adhere to the following guidelines to avoid confusing the user and to take advantage of the MPW Shell's command-line processing features.

- Your tool should take its inputs as command-line parameters, rather than prompt for input. This allows the tool to be included in a script and to take advantage of the MPW Shell's command-line processing features such as variable substitution and filename generation.
- Your tool should have command options for specifying deviations from its standard behavior. The order of these options should not be significant so that the user can specify them anywhere on the command line. Command options should not be case sensitive.
- Your tool should be able to operate on a list of filename parameters, not just one, so it can take advantage of the MPW Shell's filename generation feature.
- Your tool should take its input from standard input and write its output to standard output when the user does not specify a file parameter. The use of standard I/O allows the piping of the output of one program into the input of another. For example, the following command sends the output of the `Files` command into the input of the `Count` command, yielding the number of files and directories in the current directory:

  ```
  Files | Count -l
  ```

  If you do not use standard input or standard output, you should provide a usage message or a list of options.
- Your tool should spin the cursor to allow switching to different applications during tool execution. (The cursor is spun at regular intervals for cooperative multitasking; the spin routine includes an event loop and observes mouse-down events.)
- Your tool should operate silently as it processes its input. The spinning cursor provides visual feedback. If you think the user might want detailed feedback, you should provide a `-p` (progress) option to send status and summary information to diagnostic output.

- Error messages for your tool should be in the form of MPW Shell comments or should be "executable" so that the user can easily locate the error. For example, the language translators report errors in the form

  ```
  File "Test.c" ; line 25   ### expected: ';' got: name
  ```

  The user can execute this message to open the file and select the offending line. The error message should also include the name of the tool. For example, you can add the line "*ToolName* `- terminated!`" after the actual error message.

- You should provide a Commando interface for your tool. See Chapter 14, "Creating Commando Dialog Boxes for Tools and Scripts," for more information.

- Your tool should use temporary memory. See *Inside Macintosh: Memory* for more information.

- Upon completion, your tool should free all memory it had allocated.

See the *MPW Command Reference* for information about the syntax conventions of MPW commands.

Status Code Conventions

When your tool terminates, it should return a status code to the MPW Shell. If the status code is nonzero and if the MPW Shell variable `{Exit}` is nonzero (the default), the MPW Shell terminates the execution of the current command file. The MPW Shell also converts the status code to string form and creates an MPW Shell variable called `{Status}` with that value. You can test this variable and take appropriate action.

Table 13-1 describes the conventions used for status codes. Note that only the bottom 24 bits of a tool's status code are returned to the MPW Shell. All negative numbers, except for –9, are reserved for use by the MPW Shell.

Table 13-1 Status code conventions

Code	Meaning
0	Success
1	Command syntax error
2	Some error in processing
3	System error or insufficient resources
–9	User abort

If you want your tool to return error codes other than these, you should carefully document the numbers and their meanings.

The returned status code is undefined if you do not explicitly return a value by using the method recommended for your language. These methods are as follows:

- In C, result codes are passed as the return value from your main function or as the parameter to the C library `exit` function. Note that 0 is returned as the value of the main function in C if there is no explicit return on exit call.
- In PowerPC and 68K assembly language, the Integrated Environment routine `_RTExit` is available. `_RTExit` takes the status code as a parameter.

A Tool's Runtime Environment

Because your tool is executed within the MPW Shell environment, you need to know how tools coexist with MPW as a host application. This information is essential for memory management and debugging.

Using Initialization Routines

Because tools run in the MPW Shell, your tool should not include or call most Macintosh Toolbox initialization routines. In particular, you must not call the following routines:

`ExitToShell`	`MaxApplZone`
`InitDialogs`	`RsrcZoneInit`
`InitFonts`	`SetApplLimit`
`InitMenus`	`SetGrowZone`
`InitResources`	`TEInit`
`InitWindows`	

However, if your tool uses QuickDraw or calls any routine that uses QuickDraw, be sure to call the `InitGraf` procedure. Calling `InitGraf` is necessary because QuickDraw uses TOC-relative or A5-relative global variables, and tools have their own private global area. Even a simple call to the QuickDraw function `Random` does not work properly unless you first call `InitGraf`.

If your tool calls `InitGraf` and writes to `stdout` or `stderr` (including error messages), then you should call the `SetFScaleDisable` procedure with a parameter value of `true` after you call `InitGraf`. Otherwise, your text output might be improperly scaled.

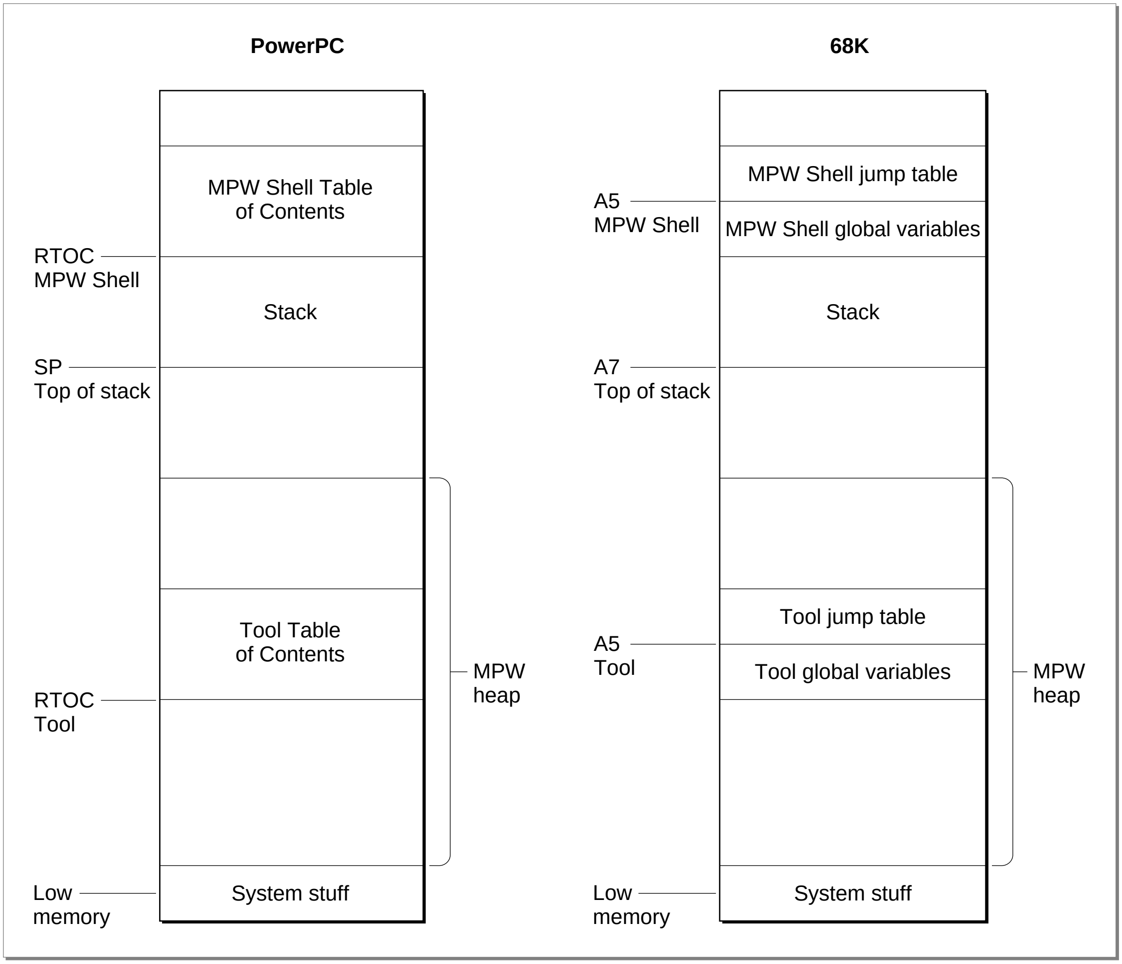
If your tool opens any windows, make sure that it closes or disposes of those windows before it terminates.

Note
If you are writing a PowerPC MPW tool, you may include user-defined initialization or termination routines. ◆

Allocating Memory Space for a Tool

The MPW Shell and tools execute out of the same heap and share the same stack. Figure 13-1 shows the allocation of memory for a tool running in the MPW Shell for the PowerPC and 68K runtime environments.

Figure 13-1 Memory maps for tools running in the MPW Shell



Before the MPW Shell launches a tool, it allocates a nonrelocatable block in the heap whose contents vary depending on the runtime environment:

- If both the tool and the MPW Shell are written in PowerPC code, the MPW Shell sets up a Table of Contents (TOC) containing the tool's global variables and transition vectors. The RTOC is adjusted to point to the beginning of the tool's TOC.

- If both the tool and the MPW Shell are written in classic 68K code, the MPW Shell sets up an A5 world containing the tool's global variables and jump table and adjusts the A5 register to point there. As with applications, jump-table entries are expressed as positive offsets from A5; global variables are expressed as negative offsets from A5.
- If a PowerPC version of the MPW Shell is launching a 68K tool, the MPW Shell sets up a pseudo-A5 world and proceeds under emulation.

Dynamic stack space required by the tool is allocated on the same stack as used by the MPW Shell. Heap objects created by the tool are stored in the MPW Shell's heap.

When a tool terminates, the MPW Shell restores the registers to their previous values and deallocates the tool's global area and any other pointers and handles in the heap that might not have been allocated. The tool's resources are not unloaded immediately; they are unlocked and made purgeable so that the space can be used if needed. This allows the MPW Shell to restart the tool quickly if it is still in memory.

▲ **WARNING**

Although the MPW Shell releases memory that the tool has allocated, sometimes the MPW Shell has insufficient information to determine the owner of a master pointer. When a master pointer is `nil`, the MPW Shell cannot release it and cannot reuse it. `nil` master pointers are produced as a result of calls to the `EmptyHandle` procedure and by a number of Resource Manager actions. For example, calling the `GetResource` function with `ResLoad` set to `false` creates a `nil` master pointer. If this is followed by a `DetachResource` or `RmveResource` procedure, the handle remains as a `nil` pointer. It is always good programming practice to clean up handles after they have become obsolete. Use the `DisposHandle` procedure to get rid of obsolete handles. ▲

Sharing and Expanding the Heap

Because the MPW Shell and tools share the same heap, some cooperation is necessary to ensure efficient use of the heap. Before the MPW Shell launches a tool, it makes many of its own heap objects unlocked and purgeable. The MPW Shell's memory-resident code is kept as low in the heap as possible. The tool's code should be moved as high in the heap as possible. This is done automatically if the locked bit is not set on the tool's code resources (the default from the linker). When allocating heap space, tools should attempt to allocate no more space than is needed so that the MPW Shell does not needlessly purge its own objects from the heap.

When there is insufficient memory to run a tool, you can make more memory available in the following ways.

To obtain more memory while running MPW:

- Close all MPW Shell windows. (A block is allocated on the heap for each open window.)
- Pipe tool output to a file, rather than to a window.
- See whether your tool can borrow memory from the Process Manager's temporary heap.

To obtain more memory by relaunching MPW:

- Specify a larger partition size for the MPW Shell by using the MPW Shell's Get Info window.
- Use the SetShellSize tool to allocate more heap space. See the next section, "Sharing the Stack," for additional information.

To obtain more memory by rebooting the Macintosh system:

- Turn off or reduce the size of the disk cache, then reboot.
- Move any debuggers from the System Folder. For example, you can free up about 90 KB by running without the MacsBug debugger; to do this, hold down the mouse button while booting.

Sharing the Stack

When the MPW Shell starts up, it immediately grows the heap to its maximum size based on the maximum stack size.

- The default maximum stack size is 10 KB when less than 480 KB is available for the application heap.
- The default maximum stack size is 20 KB when more than 480 KB is available.

You can use the SetShellSize tool to adjust the stack size and the partition size. The following statement sets the stack size to be 32 KB and the partition size to be 8192 KB:

```
SetShellSize -s 32k -p 8192K
```

For more information about the SetShellSize tool, see the *MPW Command Reference*.

Because the stack is shared by the MPW Shell and the tool, executing tools from within nested scripts results in less stack space for the tool. The MPW Shell uses about 200 bytes of stack per nesting level.

▲ **WARNING**
The 68K MPW Shell segments might not be able to load into memory if your tool calls the `MaxMem` function, allocates all available memory, and then calls any MPW Shell services, such as writing to an open window. ▲

Tool Utility Routines

The rest of this chapter presents detailed information about writing an MPW tool and about the routines included in the Runtime libraries (`MacRuntime.o` and `PPCRuntime.o`) and ToolLibs libraries (`PPCToolLibs.o`, `ToolLibs.o`, and `NuToolLibs.o`) that are used by tools. These routines allow tools to make use of many facilities, which include passing parameters, accessing MPW Shell variables, using preopened files for text-oriented input and output, handling I/O to windows and selections within windows, Finder alias resolution, signal handling, exit processing, controlling the cursor, and handling error messages. Examples of each of these routines are provided for both C and for PowerPC and 68K assembly language.

The MPW libraries contain five groups of routines you can call from your tool:

- MPW Shell environment routines and associated data structures. You use these to access MPW command-line parameters, MPW Shell variables, and the standard input, output, and diagnostic files. These routines are described in "MPW Shell Utility Routines" on page 13-29. The I/O routines described here may also be used with files in both tools and applications.

- Alias-resolution routines. You can use these routines from your tool or application to resolve aliases in a full or partial pathname. These routines are described in "Alias-Resolution Routines," beginning on page 13-37.
- MPW Shell signal-handling routines. You use these to access MPW software interrupts. These routines are described in "Signal-Handling Routines" on page 13-43.
- MPW Shell cursor-control routines. You use these to control the form and action of the cursor. The routines are described in the section "Animated Cursor-Control Routines" on page 13-47.
- Error-message routines. You use these to access messages in the Mac OS system error message file. These routines are described in "Retrieving Error Text" on page 13-52.

Programming for the MPW Shell

This section describes what your tool needs to do in order to

- access the MPW Shell
- process command-line parameters after the MPW Shell has completed processing the command line for filename generation or variable substitution
- access the MPW Shell's standard I/O files
- handle potential input and output buffering problems
- handle input and output to windows and to selections in windows
- process error information

Accessing the MPW Shell—C

To access the MPW Shell environment from C, do the following:

1. Include the necessary header files.
2. Link your program with the usual Runtime and InterfaceLib libraries. You must link with a ToolLibs library if you are using the cursor-control or error-message routines described later in this chapter. You might also need to link with `StdClib` or `StdCLib.o`. if you want to access the Standard C Library routines.

The Standard C Library interface files, in the CIncludes folder, contain most of the interfaces needed for writing tools. In addition to the Standard C Library functions, MPW's implementation of C includes the following files:

- `Signal.h`, containing the declarations of routines that give you access to MPW software interrupts. The implementation of these routines is in the appropriate Runtime library (`PPCRuntime.o`, `MacRuntime.o`, or `NuMacRuntime.o`).
- `CursorCtl.h`, containing the declarations of routines used to control the form and action of the cursor. The implementation of these routines is in the appropriate ToolLibs library (`PPCToolLibs.o` ,`ToolLibs.o`, or `NuToolLibs.o`).
- `ErrMgr.h`, containing the declaration of routines used to access messages in the Macintosh OS error-message file. The implementation of these routines is in the appropriate ToolLibs library (`PPCToolLibs.o`, `ToolLibs.o`, or `NuToolLibs.o`).

The CExamples folder contains source files for the sample tool Count.

Accessing the MPW Shell—PowerPC and 68K Assembly Language

To access the MPW Shell environment from assembly language, do the following:

1. Import the names of the routines you are using. See "Importing the Routines" on page 13-17 for additional information.
2. Use the correct calling conventions. See "Assembly-Language Calling Conventions" on page 13-17 for additional information.

   Call the `_RTInit` function early in your program. See "The _RTInit Function" on page 13-18 for additional information.

   If you want to use `exit`, `abort`, or `_RTExit` in PowerPC assembly language, your main entry point must initialize `__target_for_exit` by calling `setjmp`. Otherwise the main entry point should just do a return to the caller. Classic 68K tools can use `exit` or `abort` at the end of the program. Do not use `ExitToShell`, as this will cause the MPW Shell to quit.
3. Link your assembled object files with the library or libraries that contain the routines' code.

   The code for the MPW Shell environment and signal-handling routines are stored in the Runtime libraries (`PPCRuntime.o`, `MacRuntime.o`, or

`NuMacRuntime.o`). The code for the cursor-control and error-message routines is in the appropriate ToolLibs library. You must link the appropriate file or files to your object files if you use any of these routines.

The AExamples folder contains source files for the sample tool Count.

Importing the Routines

You can import the names of the routines described in this chapter by using `IMPORT` directives. For the MPW Shell environment and signal-handling routines, you can simply include the files `IntEnv.a` and `Signal.a`, respectively; they contain the required directives. For the cursor-control and error-message routines, you must write your own `IMPORT` directives in your source text.

The MPW Shell environment and signal-handling routines are mostly C routines; hence their names are case sensitive. The cursor-control and error-message routines are all Pascal routines. Their names are not case sensitive unless `CASE OBJ` or `CASE ON` is in effect, in which case you must import their names in capital letters.

Assembly-Language Calling Conventions

If you are writing PowerPC assembly-language code, you must follow the standard PowerPC calling conventions. Parameters are processed from left to right and are placed into the general-purpose registers GPR3 through GPR10 and (if necessary) floating-point registers FPR1 through FPR13. Any additional parameters are pushed onto the stack (which allocates enough space to hold all the parameters, whether they are passed in registers or not). Return values are placed in GPR3 or FPR1, or else indicated by passing a pointer to a structure as the implicit leftmost parameter. For more information, see *Inside Macintosh: PowerPC System Software*.

For 68K assembly-language code, you can have either C or Pascal calling conventions. The routine you are calling determines the calling convention. (Of course, if you are writing your own routines, you can choose which convention to use.)

- If the calling convention is C, then push the parameters on the stack from right to left. When the function returns, its arguments will still be on the stack and its return value will be in register D0. Note that the SC/SCpp compilers treat an `int` as 4 bytes long.

- If the calling convention is Pascal, you must reserve space on the stack for the return value, if any. Then push the arguments from left to right. When the routine returns, the arguments will no longer be on the stack; the return value (if the routine was a function) will be on top of the stack.

The _RTInit Function

The `_RTInit` function performs the following general runtime initializations:

- Allows access to the program parameters `argc`, `argv`, and `envp`.
- Determines whether the variable definitions (if any) pointed to by the environment pointer are set up as C or Pascal strings.
- Initializes internal library structures and global variables. (In the classic 68K runtime, `_RTInit` allocates approximately 500 bytes of nonrelocatable space in the heap and calls `_DataInit`, the routine that initializes global data.)

In the classic 68K runtime, `_RTInit` also performs the following:

- Calls the C++ static constructors (if they exist).
- Installs an exit procedure to ensure that the C++ static destructors (if they exist) are called on termination.
- Allocates and initializes structures used by the `IntEnv` and `StdClib` libraries.

For PowerPC tools, you do not need to call `_RTInit`, as its functions are handled by the `StdCRuntime.o` library.

For classic 68K tools, you must call `_RTInit` before any of the other routines described in this section, and you should call it before other code segments have been loaded. The `_RTInit` function should be one of the first calls in your program; the last call should be to the `exit` or `_exit` procedure, which calls the `_RTExit` procedure. The `exit` procedure is described in "exit—Terminate the Current Application" on page 13-32.

The syntax of the `_RTInit` function is

```
int _RTInit(ProcPtr retPC, int *pArgC, char ***pArgV,
            char ***pEnvP, int forPascal)
```

The function `_RTInit` uses C calling conventions; its parameters are described in Table 13-2. The `_RTInit` function returns a value of 1 if your program is being launched by the Macintosh Finder and a value of 0 if it is being launched by the MPW Shell. This is the value placed in the `StandAlone` variable, described in "StandAlone—Check If Running in the MPW Shell" on page 13-30.

Table 13-2 `_RTInit` parameters

Parameter	Value
`retPC`	The address to which program control should pass upon execution of `_RTExit`, as described in "MPW Shell Utility Routines" on page 13-29.
`pArgC`	Pointer to a long integer that `_RTInit` will set to the number of command-line parameters. For additional information, see the next section, "Accessing Command-Line Parameters."
`pArgV`	Pointer to a pointer variable that `_RTInit` will set to point to a list of parameters. For additional information, see the next section, "Accessing Command-Line Parameters."
`pEnvP`	Pointer to a pointer variable that `_RTInit` will set to the vector of exported MPW Shell variables. For additional information, see "Accessing Exported MPW Shell Variables" on page 13-22.
`forPascal`	A numeric value passed to `_RTInit`. Its value should be 0 if you want the strings pointed to by `envp` and `argv` to be in C format (terminated by a zero character), and 1 if you want them to be in Pascal format (preceded by a length byte).

For an example of the use of the `_RTInit` function in the code of an MPW tool, see `Count.a` in the AExamples folder. The routine `Init` shows how to call `_RTInit`. The exiting routine is called `Stop`; it shows how to call the last call, `exit`.

Accessing Command-Line Parameters

The MPW Shell passes command-line parameters to tools, and tools written in C and assembly language must access these parameters.

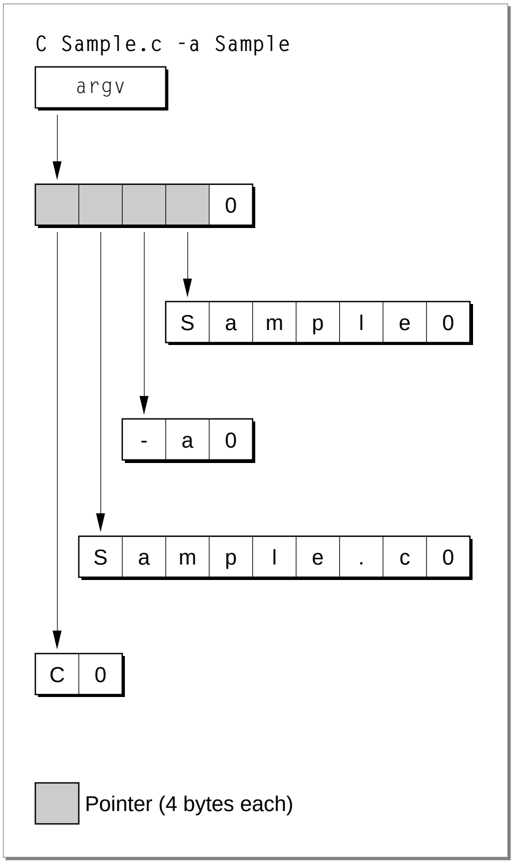
For example, the MPW Shell analyzes the following command line for any special processing, such as filename generation or variable substitution:

```
MyTool filename -optionA -optionB
```

Then the MPW Shell splits up the resulting text into individual words and uses two variables to communicate information about the number and value of the specified parameters:

- The `argv` parameter is an array of pointers to strings containing the text of the specified parameters; `argv[0]` always points to the command name. For C programs, the string is a C string. For assembly-language programs, the string type is determined by the value of the `forPascal` parameter to your `_RTInit` call. Figure 13-2 illustrates the `argv` structure.
- The `argc` parameter is the argument count, which is always at least 1, the name of the tool being the first argument.

Figure 13-2 Parameters in C



The tool can determine what action to take after accessing the information pointed to by `argv`. Every tool is passed at least one parameter: the name of the tool itself. This parameter is always the first parameter (technically, parameter 0) and is useful for error messages or other special actions.

The argument count, `argc`, contains the number of parameters including parameter 0, the command name. For example, the variable `argc` for the following command would have the value 4:

```
C Sample.a -a Sample
```

Element 0 of `argv` is always the command name, as supplied by the user. When a user is running an MPW Shell script, it's important that error messages include the name of the particular MPW program that generated the error. You can include the program name in the error message with code such as this in MPW Pascal:

```
progName := argv^[0]^;          {Store program name in temp variable.}
...
IF IOResult <> 0 THEN
    Writeln(diagnostic, progName, '-cannot open file', fileName);
```

Accessing Command-Line Parameters—C

C uses standard argument-passing mechanisms as defined in ANSI C, with the addition of a third parameter allowing access to MPW Shell variables. The main program is passed three parameters: `argc`, the argument count; `argv`, the argument vector; and, optionally, `envp`, the environmental pointer.

- The value of `argc` includes the command name (parameter 0) and is thus always one more than the number of parameters to the command.
- The `argv` parameter is a pointer to a zero-terminated array of pointers to the parameters, each of which is in C string (zero-terminated) format as shown in Figure 13-2 on page 13-20. The last element of the array is a `NULL` pointer.

The third parameter, `envp`, is described in "Accessing Exported MPW Shell Variables" on page 13-22.

Accessing Command-Line Parameters—PowerPC and 68K Assembly Language

In PowerPC or 68K assembly language, you can use the Integrated Environment routine `_RTInit` to access the command parameters. The addresses of the variables `argv` and `argc` are passed to `_RTInit`, which initializes them.

The `argv` variable, set by `_RTInit`, is a pointer to an array of strings, dynamically allocated and initialized by the MPW Shell when a tool begins execution. Each command-line parameter to the tool is stored as a Pascal-formatted or C-formatted string (depending on the value of the `forPascal` parameter passed to `_RTInit`), pointed to by a pointer in the array.

Accessing Exported MPW Shell Variables

The MPW Shell maintains a set of variables that can be made available to tools with the `Export` command. Whenever you run a tool, the MPW Shell makes a copy of the names and string values of all exported variables and passes this list to the program. The tool can then determine the value of a variable by one of two methods:

- Using the `getenv` function, which is described in “getenv—Access Exported MPW Shell Variables” on page 13-31. This is the preferred method.
- Doing a linear search of the list of variables until the desired variable name is found; the following sections explain how you access the items in this list from C, Pascal, and assembly language.

Because only a copy of the variable is passed, a tool cannot alter the value of an MPW Shell variable.

Accessing Exported MPW Shell Variables—C

You can access the list of exported MPW Shell variables from C by means of a parameter to the C main-entry-point function `main` if the main procedure is declared as

```
main(int argc, char *argv[], char *envp[])
```

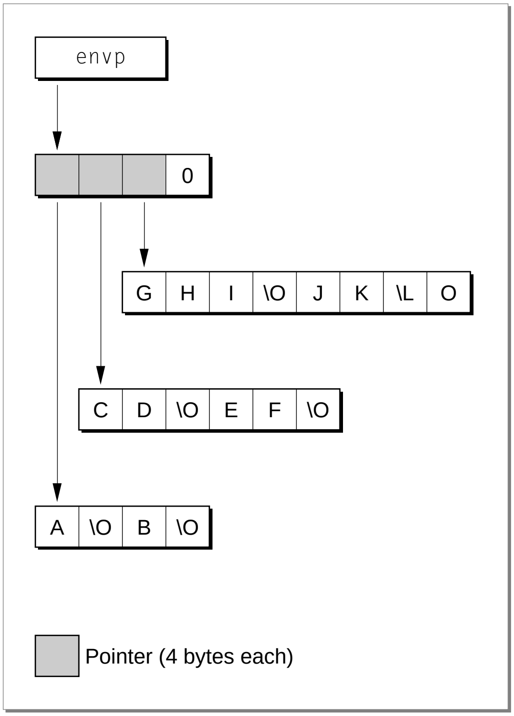
The `envp` parameter is the environment pointer, a null-terminated array of pointers. This array represents the set of MPW Shell variables that have been made available to tools by means of the MPW `Export` command. The *n*th `envp` entry has the form

```
envp[n] = "varName\0varValue\0";
```

The last `envp` entry is a null pointer. Figure 13-3 shows the format of the `envp` array.

Note
If you use `envp` to search the environment, be sure to use case-insensitive string comparisons. ◆

Figure 13-3 Format of `envp` array for CI



Accessing Exported MPW Shell Variables—PowerPC and 68K Assembly Language

The Integrated Environment routine `_RTInit` can also be used to access MPW Shell variables in assembly language. The address of `envp`, a pointer variable, is passed to `_RTInit`, which initializes it. You can choose Pascal or C strings by setting the `forPascal` parameter to the appropriate value in the call to `_RTInit`. See Table 13-2 on page 13-19 for additional information about `_RTInit` parameters.

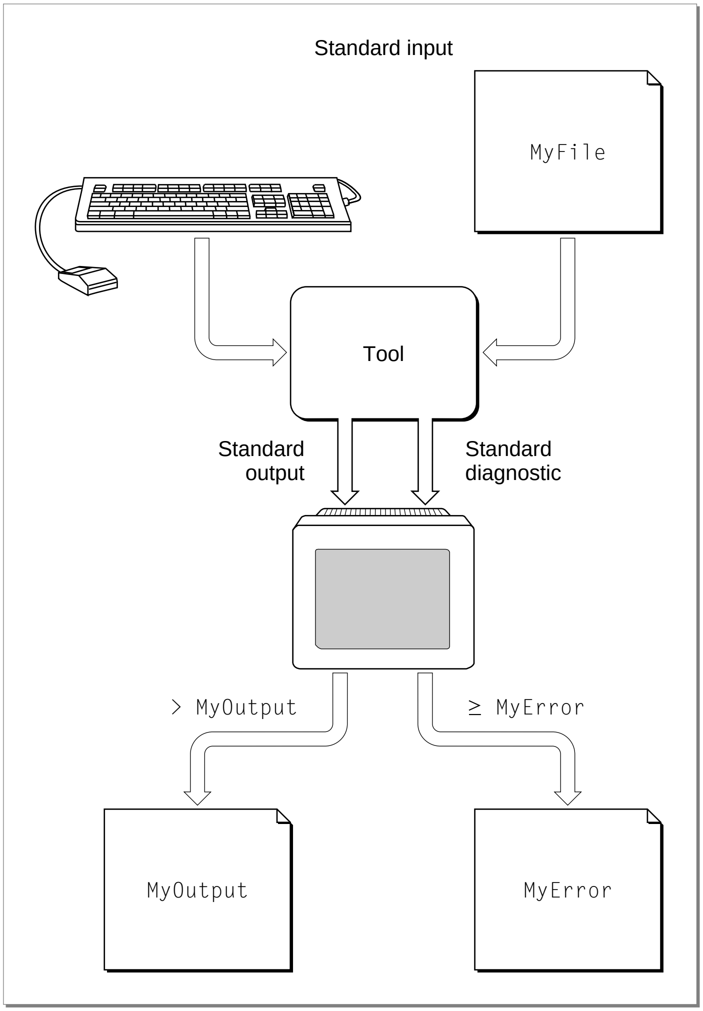
Standard Input and Output Channels

Before starting a tool, the MPW Shell sets up three text I/O channels that the tool can use to communicate with the outside world. These are

- standard input
- standard output
- diagnostic output (standard error)

By default, these channels are connected to the console (that is, the frontmost, active window). The user can type and enter (or select) program input in any window; the MPW Shell displays program output immediately after the command in the same window. The user can take this input from or direct this output to other files by specifying I/O redirection (using the operators <, >, >>, ≥, ≥≥, Σ, or ΣΣ) on the command line. When the MPW Shell encounters the I/O redirection operators, it opens or creates the necessary files, removes the redirection notation from the command line so that it doesn't appear in the program's parameter list, and then arranges for the open files to be passed to the program. When the tool finishes, the MPW Shell flushes any buffered output and closes the files. Figure 13-4 shows the use of the standard I/O channels and of the redirection operators.

Figure 13-4 The standard I/O channels and redirection



I/O Buffering

When using I/O routines provided by the language libraries, varying degrees of buffering normally occur on the standard I/O channels.

- Input from the console is buffered until the Enter key is pressed. If there is a selection when Enter is pressed, the selected text is used to satisfy the console read request; otherwise, the entire line that contains the insertion point is given to the reader.

  The MPW method of reading input creates a difficulty for interactive tools that write prompting text and pause to read a response entered on the same line. The tool receives the prompt back as part of the line read, unless the response has been selected when Enter is pressed.

- When input is taken from a file, the I/O package will, by default, read the data from the disk in 1 KB blocks.
- Text written to standard output is also buffered 1 KB at a time before being sent to a file or to the console. (As a convenience, when a read request is issued to the console, all interactive output buffers are flushed so that any prompting text will appear before the program pauses, waiting for input.)
- Text written to the diagnostic channel is buffered one line at a time so that error messages and progress information appear in a timely manner while the program is executing.

Note that this buffering can cause apparently anomalous behavior. In particular, if both standard output and diagnostic output are directed to the console, the order of the output on the screen might not match the order in which the data was written. This change in order might result because the separate buffers are flushed at different times. You can circumvent this problem by flushing standard output before writing to diagnostic output.

Assembly-language programmers must do their own buffering or call C buffered I/O routines.

In C, the standard I/O files are available for reading or writing via the file descriptors 0, 1, and 2 or via the `StdIO` stream descriptors `stdin`, `stdout`, and `stderr`. These descriptors are fully documented in the ***MPW Standard C Library Reference***.

I/O to Windows and Selections in Windows

The MPW Shell also provides tools with the ability to read and write to windows or to selections within windows. No special programming is required to use this feature. The MPW Shell monitors file system calls and intercepts those that refer to a file that is currently open as a window. These calls are redirected automatically to the window rather than the file. Thus, any modifications to the file do not become permanent until the window is saved.

Accessing selections within windows is almost as transparent to programs. All that is required is that the filename contain the `§` selection suffix (Option-6) as shown in this example:

volumeName`:MPW:Worksheet.§`

Reading from a selection is the same as reading from a file, and the beginning and end of the selection are treated as the bounds of the file. However, writing to a selection *replaces* the selection and has the useful property that the data written is inserted into the file, rather than overwriting the data that follows it in the file.

Because the MPW Shell handles window and selection I/O automatically, tools should simply assume that they are always dealing with files.

Error Information

All MPW Shell I/O routines report errors by setting the value of the integer variable `errno`. Possible error values are shown in Table 13-3. In addition, the routines `open`, `close`, `read`, `write`, and `ioctl` set the variable `MacOSErr`.

Depending on the language you use, `errno` and `MacOSErr` are handled as follows:

C	The variables `errno` and `MacOSErr` are global variables.
PowerPC and 68K assembly language	Import the variables `errno` (a long) and `MacOSErr` (a word).You can import these variables with the IntEnv.a interface file.

The variable `errno` is an integer. Its behavior is described in the *MPW Standard C Library Reference*. The values of `errno` are typically small positive integers. Zero means that there is no error. However, libraries do *not* set `errno` to zero on successful calls.

`MacOSErr` is a short (16-bit value) that holds the result codes from Macintosh Toolbox calls made by the libraries (such as the result of a file system call made by the `ioctl` function). `MacOSErr` holds 0 if there is no error; if it holds a negative number, that means there is an error. See *Inside Macintosh* for details on result codes.

Table 13-3 MPW Shell I/O errors

Value	Identifier	Message	Explanation
2	`ENOENT`	No such file or directory	This error occurs when a file whose filename is specified does not exist or when one of the directories in a pathname does not exist.
3	`ENORSRC`	Resource not found	A required resource was not found. This error applies to `faccess` calls that return tab, font, or print record information. It is also returned by file open calls when attempting to open a normal file as an alias file.
5	`EIO`	I/O error	Some physical I/O error has occurred. This error may in some cases be signaled on a call following the one to which it actually applies.
6	`ENXIO`	No such device or address	I/O on a special file refers to a subdevice that does not exist, or the I/O is beyond the limits of the device. This error may also occur when, for example, no disk is present in a drive.
7	`E2BIG`	Insufficient space for return argument	The data to be returned is too large for the space allocated to receive it.
9	`EBADF`	Bad file number	Either a file descriptor does not refer to an open file, or a read (or write) request is made to a file that is open only for writing (or reading).
12	`ENOMEM`	Not enough space	The Mac OS ran out of memory while the library call was executing.
13	`EACCES`	Permission denied	An attempt was made to access a file in a way forbidden by the protection system.
17	`EEXIST`	File exists	An existing file was mentioned in an inappropriate context.

continued

Table 13-3 MPW Shell I/O errors (continued)

Value	Identifier	Message	Explanation
19	ENODEV	No such device	An attempt was made to apply an inappropriate Mac OS call to a device; for example, to read a write-only device.
20	ENOTDIR	Not a directory	An object that is not a directory was specified where a directory is required; for example, in a path prefix.
21	EISDIR	Is a directory	An attempt was made to write on a directory, or to open a directory as a file.
22	EINVAL	Invalid parameter	Some invalid parameter was provided to a library function.
23	ENFILE	File table overflow	The table of open files is full, so temporarily a call to `open` cannot be accepted.
24	EMFILE	Too many open files	The Mac OS cannot allocate memory to record another open file.
28	ENOSPC	No space left on device	During a write to an ordinary file, there is no free space left on the device.
29	ESPIPE	Illegal seek	An `lseek` was issued incorrectly.
30	EROFS	Read-only file system	An attempt to modify a file or directory was made on a device mounted for read-only access.
31	EMLINK	Too many links	An attempt to delete an open file was made.

MPW Shell Utility Routines

MPW Shell utility routines provide methods to

- determine whether a program is running under the MPW Shell (`StandAlone`)
- access the values of MPW Shell variables (`getenv`)
- specify exit handlers (`atexit`)
- terminate the current application (`exit`, `abort`, `_RTExit`)

- access information about MPW Shell documents (`faccess`)
- determine if a particular trap is available on the current system

Table 13-4 provides summary information for the routines described in this section.

Table 13-4 MPW Shell utility routines

C	Assembly language	Action
`StandAlone`*	`StandAlone`*	Determines if a program is running in the MPW shell
`getenv`	`getenv`	Accesses the value of MPW Shell variables
`atexit`	`atexit`	Executes a routine before normal termination
`exit abort`	`_RTExit`	Closes all files opened with the standard I/O routines and terminates the program
`faccess`	`facess`	Provides access to control and status information for the specified files
`TrapAvailable`		Determines if a particular trap is available on the current system

* This is a global variable.

StandAlone—Check If Running in the MPW Shell

The standard libraries provide a method to determine if a program is running in the MPW Shell.

C

The global variable `StandAlone` is of type `int`. If `StandAlone` is 0, the program is running in the MPW Shell.

PowerPC and 68K Assembly Language

Import the `longint` variable `StandAlone` (in the interface file `IntEnv.a`). If `StandAlone` is 0, the program is running in the MPW Shell.

getenv—Access Exported MPW Shell Variables

The `getenv` function is used to access the value of MPW Shell variables.

C

```
char *getenv(char *varname)
```

The function `getenv` returns a pointer to a string containing the value of the variable whose name is specified by `varname`. If `varname` is not found or if `getenv` is called from an application, `getenv` will return `NULL`. The variable-name search is not case sensitive.

It is also possible to access the value of an MPW Shell variable by using the `envp` parameter to the C main-entry-point function. For additional information, see “Accessing Exported MPW Shell Variables—C” on page 13-22.

PowerPC and 68K Assembly Language

Use the C `getenv` function.

▲ **WARNING**

The functions `getenv` and `IEGetEnv` return a pointer to the memory location where a copy of the MPW Shell variable resides. Do not modify the value of the copy in such a way as to increase its length. ▲

atexit—Install a Function to Be Executed at Program Termination

The `atexit` function allows you to execute a routine before normal termination. The function is declared as follows:

```
int atexit(void (*func)(void))
```

Normal program termination closes and flushes open files and releases program memory. If you want additional exit processing, you can use `atexit` to insert a routine that is executed just before normal termination. The parameter `func` is a pointer to such a routine. Up to 32 exit procedures are permitted (not including the one used by the standard I/O routines to flush all the buffers). The routines specified are executed in the reverse order of their installation. The routines are called with no parameters.

▲ **WARNING**
If a function was installed more than once, it is executed as many times as it was installed. ▲

C

```
int atexit(void (*func)(void))
```

The routine `atexit` returns a value of 0 if the installation succeeds.

PowerPC and 68K Assembly Language

Use the C `atexit` routine.

exit—Terminate the Current Application

The functions `exit` and `abort` close open file descriptors and terminate the application or tool. The function `exit` takes a value that will be returned to the caller; `abort` does not. The functions are declared as follows:

```
void exit(int status)
void abort()
```

The `exit` function performs its duties in the following order:

1. It executes all exit procedures in reverse order of their installation by `atexit`, followed by the standard exit procedures if standard I/O routines were used. All buffered files are flushed and closed.
2. It closes all open files that were opened with `open`.

3. If the program is a tool running in the MPW Shell, `exit` places the lower 3 bytes of `status` into the MPW Shell's `{Status}` variable and returns control to the MPW Shell.
4. If the program is an application, `exit` terminates the application.

There is no return from `exit` or `abort`.

The functions `exit` and `abort` do not close files your tool opened with calls to the I/O routines documented in *Inside Macintosh*. However, the MPW Shell closes them after the tool returns.

The `{Status}` variable should be 0 for normal execution or a small positive value for errors. For additional information, see the section "Status Code Conventions" on page 13-8.

C

```
void exit(int status)
void abort()
```

In C, the `main` program is a function that returns an integer. The return value of `main` is interpreted by the MPW Shell as the program status. If a `main` program returns to the MPW Shell without setting `status` to an integer value, it will return a random status.

PowerPC and 68K Assembly Language

Use the C `exit` or `abort` routine. Both these routines terminate a program running in the MPW Shell by calling `_RTExit`, which is declared as follows:

```
_RTExit(longint status);
```

The `_RTExit` procedure must be the last executed routine in a tool running in the MPW Shell. It calls any routines installed by the `atexit` function (described on page 13-31) and then returns control to the address specified by the `retPC` parameter in the original `_RTInit` call.

Programs normally call the `exit` or `abort` routine.

faccess—Named File Access and Control

The function `faccess` provides access to control and status information for named files.

```
int faccess(char *filename, unsigned int cmd, long *arg)
```

The parameter `cmd` must be set to one of the constants in Table 13-5 to indicate what operation is to be performed on the file. As noted in the table, some calls to `faccess` also require the `arg` parameter, usually as a long integer or as a pointer to a long integer. All commands can be used on open or closed files.

Note
The commands shown in Table 13-5 are available to all programs running in the MPW Shell except for `F_DELETE` and `F_RENAME`, which are available to all programs. ◆

If `faccess` is successful, it returns a nonnegative value, usually 0. If the file cannot be accessed, `faccess` returns –1. If the requested resource for `F_GTABINFO`, `F_GFONTINFO`, or `F_GPRINTREC` does not exist for the named file, default values are stored, and the function returns a value greater than 0.

Table 13-5 Commands of the function `faccess`

Command	Action
`F_DELETE`	Deletes the named file, or returns an error if the file is open or in a window. The `arg` parameter is ignored. You can use this command from applications.
`F_GFONTINFO`	Returns the font and font size of an MPW text file specified by `filename`. The `arg` parameter is a pointer to a long integer. The font number is stored in the upper word of the long integer; the font size is stored in the lower word.
`F_GPRINTREC`	Gets a print record `TPrint` for the MPW text file `filename`. The `arg` parameter is a handle to the print record. Before calling `faccess` with this `cmd` value, the Macintosh Printing Manager must be initialized, and the print record handle `THPrint` must be allocated.

continued

Table 13-5 Commands of the function `faccess` (continued)

Command	Action

`F_GSELINFO`	Gets the selection information for the MPW text file specified by `filename`. The `arg` parameter is a pointer to a selection record. A selection record is a C structure (or Pascal record) in this form:
	`struct SelectionRecord {`
	`        long startingPos;`
	`        long endingPos;`
	`        long displayTop`
	`        };`
	The `startingPos` field specifies the starting position of the selection, the `endingPos` field specifies the ending position of the selection, and the `displayTop` field specifies the position of the first character at the top of the window. All three positions are offsets from the beginning of the file, with the first position in the file being 0.

`F_GTABINFO`	Returns the tab setting for an MPW text file specified by `filename`. The `arg` parameter is a pointer to a long integer. The long integer's value is the tab setting expressed as the number of spaces in the text file's font.
`F_GWININFO`	Gets the current window position. The `arg` parameter is a pointer to a rectangle (of type `Rect`) to store the information. The rectangle is in global coordinates.
`F_OPEN`	Reserved for Mac OS use.
`F_RENAME`	Renames the named file. The `arg` parameter is a pointer to a string containing the new name. You can use this command from applications.
`F_SFONTINFO`	Sets the font and font size of an MPW text file named by `filename`. The `arg` parameter is a long integer. The font number is read from the upper word of the long integer; the font size is read from the lower word.
`F_SPRINTREC`	Sets a print record for the MPW text file `filename`. The `arg` parameter is a handle to the print record. Before calling `faccess` with this `cmd` value, the Macintosh Printing Manager must be initialized, and the print record handle `THPrint` must be allocated.

continued

Table 13-5 Commands of the function `faccess` (continued)

Command	Action
`F_SSELINFO`	Sets the selection information for the MPW text file `filename`. The `arg` parameter is a pointer to a selection record (see the description of the `F_GSELINFO` command). The display starts on the line that contains the first character at the top of the window (its location is specified by the `displayTop` variable). The window does not automatically scroll horizontally to display the actual character specified. It is invalid to set `startingPos` to a negative value or to a value greater than that specified by the `endingPos` variable or to a value that is greater than the length of the file. It is also invalid to set `displayTop` to a value greater than the length of the file. If the value of the `displayTop` variable is negative, it is ignored, and only `startingPos` and `endingPos` are used. (This is useful if you want the MPW Shell to provide for scrolling only when necessary. If `displayTop` is greater than 0, scrolling is done on each `faccess` call.)
`F_STABINFO`	Sets the tab setting for an MPW text file specified by `filename`. The `arg` parameter is a long integer representing the tab setting expressed as the number of spaces in its font.
`F_SWININFO`	Sets the current window position. The `arg` parameter is a pointer to a rectangle (of type `Rect`) specifying the new size and position. If the window size is invalid or the rectangle is completely off the screen, `faccess` returns `-1`.

C

```
int faccess(char *filename, unsigned int cmd, long *arg)
```

The `cmd` constants are declared in the file `FCntl.h`. If `faccess` returns with an error, it also sets the value of `errno`.

PowerPC and 68K Assembly Language

Use the C function `faccess`. All strings are C strings. The `cmd` constants are declared in the file `IntEnv.a`. If `faccess` returns with an error, it also sets the value of `errno`.

TrapAvailable—Determine Whether Trap Is Available

The routine `TrapAvailable` is described in the Processes chapter of *Inside Macintosh: Overview* and is included in the Runtime libraries. You can call this function to determine if a particular trap is available on the current system. `TrapAvailable` is only available in C.

```
Boolean TrapAvailable (short TrapNumber);
```

Alias-Resolution Routines

If you need to access a file programmatically from your tool or application—a Preferences file, for example—you cannot use the Toolbox `ResolveAliasFile` routine because this function requires an FSSpec record that cannot be created using the `FSMakeFSSpec` routine if the original path for the file contains an embedded alias. To resolve this problem, the Runtime libraries include five routines that return the FSSpec record or resolved pathname of a pathname containing embedded and leaf aliases. You can use these routines in conjunction with Toolbox calls that require an FSSpec record or pathname.

Two of the routines return the FSSpec record that results when all aliases in the specified path have been resolved. The other three routines return pathnames instead of FSSpec records.

Note

All the routines that resolve aliases require System 7 and the Alias Manager; it is the responsibility of the caller to check for their presence. While the FSSpec record returned can be used directly in the new FSSpec-type calls for opening files, you can also extract the information for use with old-type file system calls. See the File Manager chapter of *Inside Macintosh: Files* for a description of FSSpec records and the routines that use them. ◆

Table 13-6 lists the alias-resolution routines described in this section.

Table 13-6 Alias-resolution routines

Routine name	Action
MakeResolvedFSSpec	Creates an FSSpec record with all aliases resolved.
ResolveFolderAliases	Resolves embedded folder aliases in a path to a file without resolving aliases for the leaf name in the path.
ResolvePath	Returns a path that does not contain aliases; the return value is a C string.
IEResolvePath	Returns a path that does not contain aliases; the return value is a Pascal string.
MakeResolvedPath	Returns a resolved path; a flag allows you to suppress the resolution of leaf aliases.

MakeResolvedFSSpec—Resolve Aliases and Create an FSSpec Record

The function `MakeResolvedFSSpec` creates an FSSpec record with all aliases resolved. This function can handle paths containing no aliases or just a leaf alias more quickly than the `ResolveFolderAliases` function, described next.The structure of `MakeResolvedFSSpec` (available only in C) is as follows:

```
OSErr MakeResolvedFSSpec (short volume, long directory,
            Str255 path,FSSpec *theSpec, Boolean *isFolder,
            Boolean *hadAlias, Boolean *leafIsAlias);
```

The parameters of the `MakeResolvedFSSpec` function are described below. The combinations of values for the `volume`, `directory`, and `path` parameters may be any combinations described in the File Manager chapter of *Inside Macintosh: Files*.

Parameter	Effect
volume	Specifies the volume ID (or working directory) for the file.
directory	Specifies the directory ID for the file.
path	Specifies the partial or full pathname for the file.

continued

Parameter	Effect
theSpec	Is the FSSpec record created for the file that the function returns to the caller.
isFolder	Is true if the volume, directory, and path specified a folder rather than a document.
hadAlias	Is true if the specified pathname contained an alias anywhere in the path.
leafIsAlias	Is true if the file specified by the path was an alias file.

ResolveFolderAliases—Resolve Folder Aliases

The function ResolveFolderAliases creates an FSSpec record for the specified file by stepping through all folders specified as part of the path parameter. If an alias file is encountered in place of a folder name, the alias file is resolved. If the embedded alias file points to a document instead of a folder, an error is returned. The declaration of ResolveFolderAliases (available only in C) is as follows:

```
OSErr ResolveFolderAliases (short volume, long directory,
            Str255 path,Boolean resolveLeafName, FSSpec *theSpec,
            Boolean *isFolder, Boolean *hadAlias, Boolean
            *leafIsAlias);
```

You can use this function to resolve embedded folder aliases in a path to a file without resolving aliases for the leaf name in the path. This is useful if you want to open the alias file itself, rather than the file or folder it points to.

Note that this function always parses the specified path. If you are calling this routine to resolve folder aliases embedded in the path without resolving the terminal leaf name in the path, it is more efficient to call the Toolbox routine FSMakeFSSpec first, and call ResolveFolderAliases only if the call to FSMakeFSSpec fails. If the FSMakeFSSpec call succeeds, your path contained no embedded aliases. It may, however, still contain a leaf alias, which you can resolve using the Toolbox routine ResolveAliasFile.

The parameters of the `ResolveFolderAliases` function are described below. The combinations of values for the `volume`, `directory`, and `path` parameters may be any combinations described in the File Manager chapter of *Inside Macintosh: Files*.

Parameter	Effect
`volume`	Specifies the volume ID (or working directory) for the file.
`directory`	Specifies the directory ID for the file.
`path`	Specifies the partial or full pathname for the file.
`resolveLeafName`	Determines whether the terminal leaf name specified in the `path` parameter should be resolved. If you specify `true`, the leaf name will be fully resolved and the FSSpec record returned will point to an actual file. If you specify `false`, the FSSpec record returned will point to the original leaf file (whether or not it is an alias file). The function sets the `isFolder` and `leafIsAlias` flags according to the type of file found. No error is returned if the parameter `resolveLeafName` is `false` and the file specified is not an alias file.
`theSpec`	Is the FSSpec record created for the file that the function returns to the caller.
`isFolder`	Is `true` if the volume, directory, and path specified a folder rather than a document. The value of this flag is correct regardless of the setting of `resolveLeafName`.
`hadAlias`	Is `true` if the specified pathname contained an alias anywhere in the path. The value of this flag is correct regardless of the setting of `resolveLeafName`.
`leafIsAlias`	Is `true` if the file specified by the path was an alias file. The value of this flag is correct regardless of the setting of `resolveLeafName`.

ResolvePath—Return a Resolved Path as a C String

The `ResolvePath` function accepts a path that may contain aliases and returns a path that does not contain any aliases. The declaration is as follows:

```
OSErr ResolvePath (char *rawPath, char *resolvedPath,
            Boolean *isFolder, Boolean *hadAlias);
```

You specify the path as a C string, and the function returns a C string for the resolved path. The `IEResolvePath` function, described next, is exactly the same except that it takes and returns the path as a Pascal string.

The `ResolvePath` function and the `IEResolvePath` function take a path that may contain aliases and return a path that does not contain any aliases. If the original path did not contain aliases, it is returned unchanged as the resolved path. If aliases are found, a full, resolved pathname to the target of the alias path is returned. If the resolved pathname is too long to fit in a Pascal string (and thus too long to pass to the Toolbox), both routines return the error code `badNamErr`. The routines might also return other errors if a problem is encountered while resolving aliases or constructing the resulting path. The routines return two Boolean flags that you can use to check whether the path references a file or folder and whether the original path contained aliases. The functions resolve partial pathnames relative to the current working directory.

The parameters of the `ResolvePath` and `IEResolvePath` functions are described below.

Parameter	**Effect**
`rawPath`	Specifies the path that may contain aliases. The `ResolvePath` function expects a C string; `IEResolvePath` expects a Pascal string. Do not use code-relative constant data for this parameter when calling `ResolvePath` because an in-place C-to-Pascal string conversion is done on the parameter.
`resolvedPath`	Is the pathname returned by the function. The `ResolvePath` function returns a C string; `IEResolvePath` returns a Pascal string. You must allocate at least 256 bytes of storage for this buffer.
`isFolder`	Is `true` if the volume, directory, and path specified a folder rather than a document.
`hadAlias`	Is `true` if the specified pathname contained an alias anywhere in the path.

IEResolvePath—Return a Resolved Path as a Pascal String

This function is identical to the `ResolvePath` function just described except that it accepts and returns a path as a Pascal string rather than as a C string.

```
Pascal OSErr IEResolvePath (char *rawPath, char *resolvedPath,
            Boolean *isFolder, Boolean *hadAlias);
```

MakeResolvedPath—Return a Resolved Path as a Pascal String

The function `MakeResolvedPath` returns a resolved path like the `ResolvePath` and `IEResolvePath` routines just described, but gives you more flexibility by allowing you to specify a partial pathname and also allows you to specify a volume and directory in addition to a pathname. If you specify a partial pathname, the function evaluates the path relative to the volume and directory you specify rather than relative to the current working directory, as is the case with the `ResolvePath` and `IEResolvePath` functions. The C declaration of `MakeResolvedPath` is as follows:

```
OSErr MakeResolvedPath (short volume,long directory, Str255 path,
            Boolean resolveLeafAlias, char *buffer,Boolean
            *isFolder, Boolean *hadAlias, Boolean *leafIsAlias);
```

The function `MakeResolvedPath` also includes a flag that allows you to suppress the resolution of leaf aliases. This is useful when you want to act on an alias file directly.

The parameters of the `MakeResolvedPath` function are described below.

Parameter	**Effect**
`volume`	Specifies the volume ID (or working directory) for the file.
`directory`	Specifies the directory ID for the file.
`path`	Is a Pascal string that specifies the partial or full pathname for the file.

continued

Parameter	Effect
`resolveLeafAlias`	Determines whether the terminal leaf name specified in the `path` parameter should be resolved. If you specify `true`, the leaf name will be fully resolved and the path returned will point to an actual file. If you specify `false`, the path returned will point to the original leaf file (whether or not it is an alias file). The function sets the `isFolder` and `leafIsAlias` flags according to the type of file found. No error is returned if the `resolveLeafAlias` parameter is `false` and the file specified is not an alias file.
`buffer`	Returns the resolved pathname as a Pascal string.
`isFolder`	Is `true` if the volume, directory, and path specified a folder rather than a document. The value of this flag is correct regardless of the setting of `resolveLeafAlias`.
`hadAlias`	Is `true` if the specified pathname contained an alias anywhere in the path. The value of this flag is correct regardless of the setting of `resolveLeafAlias`.
`leafIsAlias`	Is `true` if the file specified by the path was an alias file. The value of this flag is correct regardless of the setting of `resolveLeafAlias`.

Signal-Handling Routines

MPW provides a set of routines to handle signals. Signal handling is available only for tools that run in the MPW Shell; it is not available for applications that run in the Macintosh Finder.

A **signal** is similar to a hardware interrupt in that its invocation can cause program control to be temporarily diverted from its normal execution sequence; the difference is that the events that raise a signal reflect a change in program state rather than hardware state. Examples of signal events are stack overflow, heap overflow, software floating-point exceptions, and Command-period interrupts.

A program running in the MPW Shell can detect two software interrupts. One is the Command-period, represented by the value `SIGINT`. The other is abnormal termination by the `Abort` function, represented by the value `SIGABRT`.

As additional software interrupts are added, new values will be added to represent them. The signal-handling routines will then accept these new values.

The default action of any signal is to close all open files, execute any exit procedures (described in “exit—Terminate the Current Application” on page 13-32), and terminate the program. If your tool requires special handling of a signal or chooses to ignore the signal, you can use the routine `signal` to replace the default signal-handling routine with your own routine.

Table 13-7 lists the signal-handling routines described in this section.

Table 13-7 Signal-handling routines

C	Assembly language	Action
`signal`	`signal`	Replaces the current signal handler with a user-supplied signal handler
`raise`	`raise`	Allows signals to be raised under program control

Signal Handling—C

To access the signal handler in C, do the following:

- Include the file `Signal.h` in your source text.
- Link your program with the file `MacRuntime.o`.

The type definition `SignalHandler`, used later in this section, is *not* included in the file `Signal.h`. It might be typed as follows:

```
Typedef void (*SignalHandler) (int);
```

Signal Handling—PowerPC and 68K Assembly Language

To access the signal handler in assembly language, do the following:

- Include the file `Signal.a` in your source file.
- Link your program with the file `StdClib` (for the PowerPC Assembler) or `IntEnv.o` (for the 68K MPW Assembler).

signal—Specify a Signal Handler

The function `signal` replaces the current signal handler (the routine to be executed upon receipt of the signal specified by the `signum` parameter) with a user-supplied signal handler. The function is declared as follows:

```
void (*signal (int signum, void (*newHandler)(int)))(int);
```

You can set or restore the default signal handler by specifying `SIG_DFL` as the current signal handler.

You can specify one of two predefined signal handlers in `newHandler` parameter:

- The function `SIG_IGN` does nothing. You can specify it to ignore the signal.
- The function `SIG_DFL` is the default signal handler. It calls the program's `exit` procedure.

The `newHandler` function that is passed to `signal` takes one parameter (a long integer). The parameter is the number of the signal that is currently being handled. Writing a signal handler is described in "Writing a Signal Handler" on page 13-46.

The function `signal` returns the previous `SignalHandler` pointer. If this pointer must be restored in another part of the program, save the return value and restore it with another call to `signal`.

C

```
void (*signal (int signum, void (*newHandler)(int)))(int);
```

Alternatively, you can use the equivalent:

```
Typedef void (*SignalHandler)(int);
SignalHandler *signal(int signum, SignalHandler *newHandler)
```

PowerPC and 68K Assembly Language

Use the C function `signal`.

raise—Raise a Signal

The function `raise` allows signals to be raised under program control. It sends the signal `signum` to the program. The function (usable in both C and assembly language) is declared as follows:

```
int raise(int signum)
```

The function returns 0 if successful, nonzero otherwise. Notice that `raise` might not return, depending on the signal handler installed.

Writing a Signal Handler

The declaration of the `signalHandler` routine is as follows:

```
void signalHandler(int signum)
```

When a signal is raised, a call is made to the handler specified as the parameter `newHandler` in a call to `signal`. One parameter is passed to the signal handler. This parameter, `signum`, is the signal number currently being handled.

When the tool starts, all signal handlers are set to `SIG_DFL`; this procedure disables all signals and calls the routine `exit`. To specify your own signal handler routine, call `signal` with your routine as the `newHandler` parameter. When the signal is raised, your routine is called. Before your routine is called, the `SIG_DFL` routine is reinstalled as the handler for that signal. Therefore, if you want to continue handling the signal, your routine must reinstall itself with another call to `signal` at the end of your signal handler.

▲ **WARNING**
Because `SIG_DFL` is reinstalled as part of the signal-handling process, your tool could be interrupted by a second signal that would then call `SIG_DFL`. It is safest to disable further signals by calling `signal(SIG_IGN)` at the beginning of your handler. Then reinstall the appropriate handler at the end. ▲

You can think of signals as operating at the interrupt level. Therefore, the safest signal handler would set a global flag, reinstall itself, and return. Then in the main body of your code, you could check for the flag and take appropriate action.

If you want to terminate program execution because of a signal, do the following:

1. In your signal handler, disable that signal (using `SIG_IGN`) and set a flag.
2. In the main body of your code, you can do some cleanup procedures and call `exit`.

If you install a signal handler for Command-period, you should return an exit code of –9 to the MPW Shell. For information on returning exit codes, see “exit—Terminate the Current Application” on page 13-32.

Signals cannot be raised while executing in ROM or in the MPW Shell. If a signal event occurs while executing outside the tool, the signal state is set, and the signal handler is executed as soon as program control returns to the tool. Because a signal can interrupt the tool at any point, there is no protection against heap corruption if a signal handler executes calls that modify the state of the heap. Because most buffered I/O potentially modifies the heap, writing to standard output or standard error is *not recommended* in signal handlers.

If you must perform I/O or other operations as a result of a signal, set a flag and check the flag during your own processing loop.

Animated Cursor-Control Routines

Five routines in the MPW ToolLibs libraries let you control the appearance and action of the MPW cursor, which is used to tell users that their commands are being processed. In addition, when used in MPW tools, spinning the cursor allows your tool to operate in the background in a multi-application environment.

These routines all use Pascal calling conventions and Pascal-style strings. Table 13-8 lists the cursor-control routines described in this section.

Table 13-8 Cursor-control routines

Name	Action
InitCursorCtl	Initializes the cursor-control (CursorCtl) unit
Show_Cursor	Increments the cursor level, which might have been decremented by Hide_Cursor
Hide_Cursor	Calls the QuickDraw HideCursor procedure
RotateCursor	Rotates the cursor image by one frame
SpinCursor	Rotates the cursor image but maintains own internal counter

Accessing Cursor-Control Routines—C

The C header file CursorCtl.h provides interfaces to procedures in the MPW ToolLibs libraries that let you control the appearance and action of the cursor. Link this file with the appropriate file (PPCToolLibs.o, ToolLibs.o, or NuToolLibs.o).

To access the cursor-control unit in C, include this statement in your source file:

```
#include <CursorCtl.h>
```

InitCursorCtl—Initializing Cursor Control

The InitCursorCtl procedure initializes the CursorCtl unit. Call this procedure once, prior to calling the RotateCursor procedure (described on page 13-51) or the SpinCursor procedure (described on page 13-52). You do not need to call InitCursorCtl if you use only Hide_Cursor and Show_Cursor. The C declaration is as follows:

```
pascal void InitCursorCtl(acurHandle newCursors);
```

If the parameter newCursors is nil, InitCursorCtl loads an 'acur' resource and the 'CURS' resources specified by the 'acur' resource ID. If any of the resources cannot be loaded, the cursor is not changed. The 'acur' resource is assumed to be either in the currently running tool or application, or in the MPW Shell for a tool, or in the System file. The 'acur' resource ID must be 0 for a tool or application, 1 for the Shell, and 2 for the System file (assuming that cursors are in the System file).

If newCursors is not nil, it is assumed to be a handle to an 'acur'-formatted resource designated by the caller, and this resource is used instead of executing the GetResource procedure on 'acur'.

Note

If you call RotateCursor or SpinCursor without first calling InitCursorCtl, then RotateCursor and SpinCursor do the work of InitCursor the first time you make the call. However, calling InitCursorCtl yourself avoids possible memory fragmentation. ◆

CursorCtl declares acurHandle as a handle to 'acur' resources of type RECORD as follows:

```
TYPE
    acurHandle = ^acurPtr;                  {Handles to 'acur' resources}
    acurPtr    = ^acur;                     {Pointers to 'acur' resources}

    acur  =
    RECORD                                  {Layout of an 'acur' resource}
        N:          Integer;                {Number of cursors (“frames of film”)}
        Index:      Integer;                {Next frame to show <for internal use>}
        Frame1:     Integer;                {'CURS' resource ID - frame #1}
        fill1:      Integer;                {<for internal use>}
        Frame2:     Integer;                {'CURS' resource ID - frame #2}
        fill2:      Integer;                {<for internal use>}
{- - -- - - - - -- - - - - -- - - - - -- - - - -}
        FrameN:     Integer;                {'CURS' resource ID - frame #2}
        fillN:      Integer;                {{<for internal use>}
    END;
```

See “RotateCursor—Spin Cursor Using External Counter” on page 13-51 for a description of how the 'acur' frames are used to animate the cursor.

▲ **WARNING**
`InitCursorCtl` modifies the `'acur'` resource in memory. Specifically, it changes each `FrameN/fillN` integer pair to a handle to the corresponding `'CURS'` resource, also in memory. Thus, if `newCursors` is not `nil` when `InitCursorCtl` is called, you must guarantee that `newCursors` always points to a "fresh" copy of an `'acur'` resource. This need concern you only if you want to repeatedly use multiple `'acur'` resources during the execution of your tools. ▲

Show_Cursor—Increment Cursor Level

The `Show_Cursor` procedure increments the cursor level (which may have been decremented by `Hide_Cursor`). If the level is 0, it displays the cursor. The cursor level never increments above 0. The C declaration is as follows:

```
pascal void Show_Cursor(Cursors cursorKind);
```

The parameter `cursorKind` lets you select the form of the cursor. The file `CursorCtl.h` declares the type `Cursors` as well as the possible values shown in Table 13-9.

Table 13-9 Cursor kinds

Value	Cursor
0	`HIDDEN_CURSOR`
1	`I_BEAM_CURSOR`
2	`CROSS_CURSOR`
3	`PLUS_CURSOR`
4	`WATCH_CURSOR`
5	`ARROW_CURSOR`

Except for `HIDDEN_CURSOR`, the QuickDraw `SetCursor` procedure is done for the specified cursor prior to calling `ShowCursor`. `HIDDEN_CURSOR` simply causes a `ShowCursor` call.

Note
You must call `InitGraf` before the `ShowCursor` call when selecting the `ARROW_CURSOR` type because it is one of the QuickDraw global variables set up by `InitGraf`. ◆

Hide_Cursor—Decrement Cursor Level

The `Hide_Cursor` procedure calls the QuickDraw `HideCursor` procedure. (Thus, the Macintosh cursor level is decremented by 1 when this routine is called.) If the cursor was visible, it is then hidden. For further information, see details about QuickDraw in *Inside Macintosh: Imaging With QuickDraw.*

```
pascal void  Hide_Cursor(void);
```

RotateCursor—Spin Cursor Using External Counter

The `RotateCursor` procedure rotates the cursor image by one frame whenever the value of `counter` is a multiple of 32. The C declaration is as follows:

```
pascal void RotateCursor(long counter);
```

To use `RotateCursor`, your program must set up and increment (or decrement) a suitable counter. If the value of `counter` is positive, the cursor rotates clockwise (that is, sequencing is forward through the `'acur'` cursor frames); if the value of the counter is negative, the cursor rotates counterclockwise (that is, sequencing is backward through the `'acur'` resource frames).

Note
`RotateCursor` invokes a QuickDraw `SetCursor` call for the proper cursor picture. It assumes that the cursor is visible as the result of a prior `Show_Cursor` call. ◆

SpinCursor—Spin Cursor Using Internal Counter

The `SpinCursor` procedure performs the same actions as `RotateCursor`, but maintains its own internal counter rather than passing a counter. It is provided for those who do not have a convenient counter handy but still want to use the spinning cursor or any sequence of cursors specified by `InitCursorCtl`. The C declaration is as follows:

```
pascal void SpinCursor(short increment);
```

Your program specifies the `increment` to be counted (either positive or negative), and `SpinCursor` adds it to its counter. It is the sign of the increment, not the sign of the accumulated value of the `SpinCursor` counter, that determines the cursor's direction of spin.

- A positive increment spins the cursor clockwise (that is, sequencing is forward through the `'acur'` cursor frames).
- A negative increment spins it counterclockwise (that is, sequencing is backward through the `'acur'` resource frames).
- An `increment` value of 0 resets the counter to 0.

Retrieving Error Text

Six routines in the MPW ToolsLibs libraries let you retrieve and modify the text of error messages in the Macintosh OS error message file or in an error file private to a tool (created with the MakeErrorFile tool).

Table 13-10 provides summary information for the routines described in this section.

Table 13-10 Routines used to retrieve error message text

Name	Action
`InitErrMgr`	Allows the MPW Error Manager to access the file `SysErrs.Err` or a custom error file
`GetSysErrText`	Retrieves the message text that corresponds to the system error number
`GetToolErrText`	Retrieves the message text that corresponds to the error number in a tool's error message file
`AddErrInsert`	Adds another insert to an error message string
`addInserts`	Adds a set of inserts to an error message string
`CloseErrMgr`	Closes all files opened by the MPW Error Manager

Error Manager—C

To use the Error Manager in C, do the following:

- Include the header file `ErrMgr.h` (which includes the file `Types.h`) as follows:

```
#include ErrMgr.h
```

- Link your file with the appropriate ToolLibs library (`PPCToolLibs.o`, `ToolLibs.o`, or `NuToolLibs.o`).

InitErrMgr—Accessing Error Messages

You must call the `InitErrMgr` routine before any of the other Error Manager routines.

```
InitErrMgr(Str255 toolErrFilename,Str255
     sysErrFilename, Boolean showToolErrNbrs);
```

To access Mac OS error messages, use the Pascal call

```
InitErrMgr('', '', false);
```

This call causes the Error Manager to access the file `SysErrs.err`. Table 13-11 shows the order that folders are searched when trying to open `SysErrs.err`. The search order is from top to bottom, and certain folders are searched only for MPW tools.

Table 13-11 `SysErrs.Err` search path

Folder	Searches for application?	Searches for tool?
Current working directory	Yes	Yes
`{ShellDirectory}`	No	Yes
System Folder	Yes	Yes
System Folder:Preferences	Yes*	Yes*
`{PrefsFolder}`	No	Yes*

* This folder is searched only under System 7.0 or later.

The routine `InitErrMgr` opens the first occurrence of the file `SysErrs.err` it finds when searching the above paths.

Note
If a system message filename was not specified to `InitErrMgr`, then the Error Manager uses the message file contained in the file `SysErrs.err`. This file is first accessed as `{ShellDirectory}SysErrs.err` on the assumption that `SysErrs.err` is kept in the same directory as the MPW Shell. If the file cannot be opened, then the Error Manager attempts to open `SysErrs.err` in the System Folder. ◆

If `InitErrMgr` is not explicitly called, then `GetSysErrText` or `GetToolErrText` calls `InitErrMgr` (`'', '', TRUE`) the first time it is called.

If you wish to access a tool-specific error file, supply the name of the error file as the first parameter to `InitErrMgr`. If the tool is an MPW tool with the error file copied into the tool's data fork, the first parameter may be the null string, and the `InitErrMgr` routine opens the appropriate file. This occurs only if a Runtime library or `PasLib.o` is linked with the program.

The setting of the `showToolErrNbrs` parameter determines whether error numbers are displayed. Specify `true` if you want all messages to end with the error number, as in

msgtxt `([OS]Error` ***n***`)`

If the Error Manager cannot find the message text, it displays the message

`([OS]Error` ***n***`)`

The `toolErrFilename` parameter specifies the name of the tool-specific error file and should be the null string if not used (or if the tool's data fork is to be used as the error file). Use `sysErrFilename` to specify the name of the system error file. This should normally be the null string, which causes the Error Manager to look in the MPW Shell directory or in the System Folder for `SysErrs.err`. Specifying names for the error files avoids `IntEnv` calls that look up the values of Shell variables.

Note
The assembly-language caller must define and export the variable `_EnvP` with a null value if `MacRuntime.o` is not linked with the tool. For example, outside all modules (`procs`) place the following:

```
            EXPORT_         EnvP
EnvP        DC.L            Ø
```

GetSysErrText—Fetch Error Message

The `GetSysErrText` procedure gets the message text that corresponds to the system error number specified in the `msgNbr` parameter. This information is maintained in the system error-message file (`SysErrs.err` in `{ShellDirectory}`).

The `errMsg` parameter is a pointer to a string of type `Str255`, in which the error message text will be placed. The maximum length of the message is limited to 254 characters. The C declaration is as follows:

```
void GetSysErrText(short msgNbr,char *errMsg);
```

If `GetSysErrText` is successful (and if `ShowToolErrNbrs` is `true` on the initialization call), the form of the error message returned is

errorText (*OSErrorNumber*)

If `GetSysErrText` is unsuccessful, the form of the error message returned is

OSErrorNumber (*reasonMessageNotFound*)

The `GetSysErrText` routine might fail if the file `SysErrs.Err` is not found or if it contains no message text corresponding to `msgNbr`.

GetToolErrText—Fetch Text From Specified File

The `GetToolErrText` routine gets the message text that corresponds to error number `msgNbr` in the tool error message file. The name of this file is specified in the `InitErrMgr` call. The text message is returned in `errMsg`. The C declaration is as follows:

```
void GetToolErrText(short msgNbr,
                char *errInsert,char *errMsg);
```

Inserts are indicated in error messages by specifying the ^ character to indicate where the insert is to be placed. Any message to be inserted should be contained in `errInsert`. Otherwise, `errInsert` should be `NULL`. The error insert placed in the text of the error message replaces the first instance of the character ^ in the message; if no ^ is present, the error insert is appended to the end of the text of the message following an intervening blank.

Note

If a tool message filename was not specified to `InitErrMgr`, then the Error Manager assumes the message file is in the data fork of the tool calling the Error Manager. The name of the tool is stored in the Shell variable `{Command}`, and the value of that variable is used to open the error message file. ◆

AddErrInsert—Add Another Insert to Message

The `AddErrInsert` routine adds another insert to an error message string. Use this call when more than one insert is needed in a message (because it contains more than one ^ character). The insert is handled in the same fashion as in the `GetToolErrText` procedure. The C declaration is as follows:

```
void AddErrInsert(unsigned char *insert,
    unsigned char *msgString);
```

addInserts—Add Inserts to Message

The `addInserts` routine is available only in C. It is declared as follows:

```
extern unsigned char *addInserts(unsigned char *msgString,
    unsigned char *insert, ...);
```

The `addInserts` routine adds a set of inserts to an error message string. `AddErrInsert` is called for each insert parameter specified. Note that the last parameter must be a null string pointer. If not, memory may be corrupted.

CloseErrMgr—Close Error Files

The `CloseErrMgr` routine closes all files opened by the Error Manager. If you omit this call, which is not recommended, normal program termination closes the files. The C declaration is as follows:

```
void CloseErrMgr(void);
```

Creating Commando Dialog Boxes for Tools and Scripts

Contents

You can create a Commando dialog box for tools you build yourself by creating a `'cmdo'` resource and appending the compiled resource to the resource fork of your tool. This chapter includes detailed instructions on how to write a `'cmdo'` resource. If you have created Commando dialog boxes using earlier versions of MPW, you do not have to read this chapter.

Providing a Commando dialog box for a tool makes it easier for the user

- to learn about a tool because the dialog box displays available command options along with help text explaining each option
- to run the tool because the Commando program constructs a syntactically correct command line based on the choices the user makes when clicking Commando dialog box controls

Introduction to MPW describes the parts of the Commando dialog box and how Commando constructs the command line based on user selections. You must read this information if you have never used a Commando dialog box. If you are not familiar with the process of creating and building resources, you must also read Chapter 6, "Creating Noncode Resources and Manipulating Resources."

This chapter begins with four sections that provide information you must know in order to create Commando dialog boxes.

- "About the Commando Program" describes how the Commando program works and summarizes the steps required to create a Commando dialog box.
- "Creating a 'cmdo' Resource" offers basic information about identifying a `'cmdo'` resource and about the recommended sizes for Commando dialog boxes and controls.
- "Editing Commando Dialog Boxes" explains the use of the Commando editor to fine-tune the positioning of controls in the dialog box.
- "The Structure of a 'cmdo' Resource" provides a summary of `'cmdo'` controls and explains how you specify dependencies between controls.

The section "Commando Control Reference" describes the types of controls used in a Commando dialog box, the type declaration used to specify each control, and a sample data definition for each. Within these sections, you only have to read those subsections that describe the type of control you are interested in including in your dialog box.

If you want to nest Commando dialog boxes, see "Using Nested Dialog Boxes." For information on how to redirect your input or output, see the section "Redirection."

The section "A Commando Example" shows a sample resource description file for creating a Commando dialog box.

Note
Because the process of writing a `'cmdo'` resource is exactly the same for a tool or a script and because the steps required to append it to the resource fork of a tool or script are also the same, this chapter makes no distinction between the two and, for the sake of brevity, refers generically to tools and scripts as *tools*. ◆

About the Commando Program

This section explains how Commando uses the information provided by the `'cmdo'` resource to build a dialog box, describes the MPW Shell variables used by Commando, and summarizes the steps required to create a Commando dialog box.

When a user invokes Commando, the program looks in the resource fork of a tool or script for a resource of the type `'cmdo'`. It then loads the resource, builds a dialog box list, handles events, and passes the command line back to the MPW Shell for execution. The `'cmdo'` resource describes the dialog box (and its controls) used by the tool. A dialog box can include other nested dialog boxes.

The user can invoke Commando interactively or by executing a script that contains a command preceded by the word `Commando` (or the command name followed by an ellipsis).

Note
Of the two methods for invoking Commando from a script, one allows alias substitution and the other does not. If the user includes the line

```
commando NewTool
```

in a script, Commando cannot find the command `NewTool` if it has been aliased to another name. If the user includes the following line in a script

```
NewTool…
```

to invoke Commando from a script, the Commando user interface is invoked after the MPW Shell has carried out all alias and variable substitutions. ◆

MPW Shell Variables Used by Commando

Commando uses three MPW Shell variables:

- `{Aliases}`. This variable lists all defined aliases, with each name separated by a comma. The list contains only the names, not the definitions. Commando uses `{Aliases}` with the built-in command `Alias`. Without this variable, Commando would have no way of knowing the names of existing aliases. The variable `{Aliases}` is exported by the `Startup` script.
- `{Commando}`. This variable tells the MPW Shell which tool to execute when the ellipsis character is present in a command line. The value of the `{Commando}` variable is set to `Commando` in the `Startup` file. If you want to substitute another Commando-type tool, redefine the `{Commando}` variable to the name of the tool. If the variable does not exist, the MPW Shell removes the ellipsis from the command line and executes the command.
- `{Windows}`. This variable lists the current windows, with each name separated by a comma. Commando uses this information to redirect input to or output from existing windows. The variable `{Windows}` is exported by the `Startup` script.

Creating Commando Dialog Boxes

If you have never created a Commando dialog box, you should start by examining the definition of the sample Commando resource `Count.r` in the CExamples folder or `ResEqual.r` in the PExamples folder.

You can use the following steps to create a Commando dialog box for a tool or script:

1. **Create a resource description file for the `'cmdo'` type resource.**

   The type declaration for the `'cmdo'` resource is found in the `Cmdo.r` file in the RIncludes folder. Throughout this chapter, each type of Commando control is illustrated by a figure showing the control and the type declarations and data definitions that produce that control.

If an existing tool has a Commando control that you want to use, decompile the tool's `'cmdo'` resource by using the resource decompiler DeRez, then cut and paste the relevant fields into your resource description file. For example, to examine the Pascal compiler's `'cmdo'` resource, you would use the following command:

```
DeRez {MPW}Tools:Pascal -only cmdo cmdo.r
```

2. **Compile the `'cmdo'` resource and append it to your tool or script using a command like the following:**

```
Rez  MyCommando.r  -o  "{MPW}MyTool"  -a
```

3. **Display the Commando dialog boxes for your tool or script. If you want to make additional changes, use the Commando editor to adjust the coordinates of windows and move or resize the controls. If you are modifying an existing resource, you'll need to edit the help messages as well.**

   The use of Commando's editor is described in "Editing Commando Dialog Boxes" on page 14-9.

4. **When you are satisfied with your final result, decompile the `'cmdo'` resource to retain an archival copy, and add comments for clarity.**

   You should also decompose any radio button dependencies (that is, separate the part number from the button number) so that they are readable. See "Radio Buttons," beginning on page 14-26, for more information.

Creating a 'cmdo' Resource

The type declaration file for Commando, `Cmdo.r`, is located in the RIncludes folder. To create a `'cmdo'` resource from scratch, you must create a file containing `Resource` statements that correspond to the `Type` statements included in the `Cmdo.r` file.

Following a brief discussion of `'cmdo'` resource IDs and names, and a summary of the recommended sizes for a Commando dialog box and its controls, the section "The Structure of a 'cmdo' Resource" on page 14-14 describes the fields of the `'cmdo'` resource, what aspects of a Commando dialog box these fields control, and explains how you must specify values for these fields (in corresponding `Resource` statements) in order to create a Commando dialog box.

Resource ID and Name

You can use any valid resource ID for a `'cmdo'` resource.

Specifying a name for the `'cmdo'` resource affects the name displayed by Commando as a label for the Do It button. The Do It button is the dialog box control the user selects to execute the tool or script.

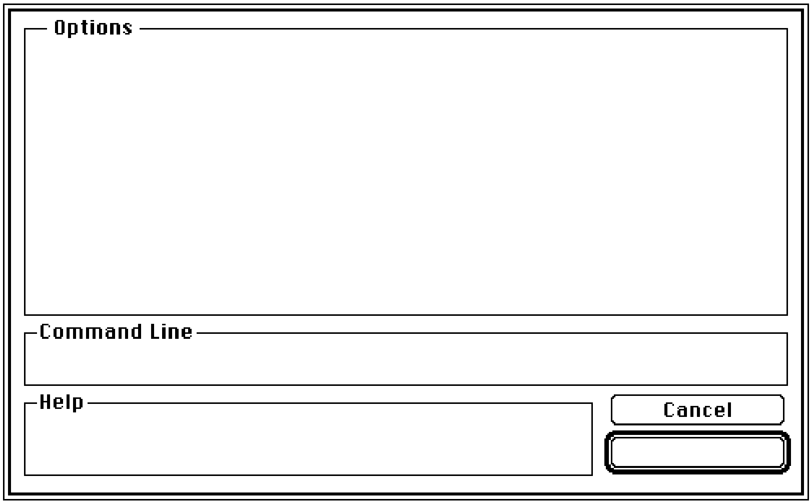
- If you do not specify a resource name, Commando uses the name of the tool or script passed from the MPW Shell. Commando capitalizes the first character and forces the rest of the characters to lowercase. For example, “StackSNiffER” becomes “Stacksniffer.”
- If you do specify a resource name, Commando uses that name as the label for the outlined Do It button. You can use this method to override the capitalization scheme used by Commando. If you do, you should remember to change the resource name if you rename your tool.

You might want to specify a resource name that is different from the name of the tool or script to help the user. For example, you might want the Do It button for the C tool to say “Compile” rather than “C.”

Size of the Dialog Box and Controls

The width of Commando dialog boxes is fixed at 480 pixels. You are free to set the height to accommodate the controls in your tool’s dialog box. The number specifying the height shouldn’t exceed 295 to be compatible with the smaller Macintosh screens. Specifying 295 pixels for the height in the `'cmdo'` resource results in the layout shown in Figure 14-1.

Figure 14-1 The basic template for a Commando dialog box



The part of the Commando dialog box labeled *Options* in Figure 14-1 represents the user control area. When you are using Commando in edit mode, you cannot select or move controls outside of this area, although you can edit the text shown in the Help box.

At the bottom of the Commando dialog box is a three-line Help box. The text in this box should be a brief, concise description of the tool, stating what it does. The Help box is not scrollable, so you need to limit your text to the confines of the box.

Table 14-1 gives dimensions for elements that can be used in a Commando dialog box. These dimensions are recommended but not required. The sizes of the text-editing fields are important if you want to avoid text that shifts up and down slightly when it is selected.

Table 14-1 Recommended sizes for Commando dialog box elements

Screen element	Recommended size
Regular entries	16 pixels high
Multiple regular entries	16 pixels per line
Checkboxes	16 pixels high
Radio buttons	16 pixels high
Pop-up menus	19 pixels high
Pop-up menu titles	16 pixels high (Top of title starts 1 pixel below the top of the pop-up menu; that is, top of title = top of pop-up menu + 1 pixel.)
Editable pop-up menus	20 pixels high
Editable pop-up titles	16 pixels high (Top of title starts 3 pixels below the top of the editable pop-up menu; that is, top of title = top of editable pop-up menu + 3 pixels.)
Icons	32 pixels high, 32 pixels wide
Pictures	Same relative bounds as the rectangle stored in the `'PICT'` resource

Editing Commando Dialog Boxes

You can use the built-in Commando editor to fine-tune the placement and size of the controls in a Commando dialog box and to edit text labels. You use the editor after you have built your resource and appended it to the tool's resource fork.

Starting with version 3.0 of MPW, Commando offers a built-in editor that lets you edit text labels and help messages and graphically move and size the controls within a Commando dialog box. This feature makes designing, redesigning, and fine-tuning Commando dialog boxes much easier.

Although you can use the Commando editor to move and size controls, you cannot use it to create, duplicate, or delete controls. This means that you still have to manually create the Commando resource, but that you don't have to be concerned about the coordinates and sizes of the controls. Once you've created the `'cmdo'` resource, you can simply bring up the Commando dialog box in edit mode, arrange all the controls to your liking, and then use DeRez to decompile the `'cmdo'` resource.

To enable the Commando editor, you hold down the Command key immediately after launching Commando until the wristwatch cursor appears. You can also invoke the editor by specifying the `-modify` option on the command line that you use to invoke Commando, as in these examples:

`Commando` ***toolName*** `-modify`

or

toolName… `-modify`

After you launch Commando with the built-in editor enabled, Commando can run in one of two modes:

- Edit mode. You use this mode to relocate and resize controls and edit text labels and help text.
- Normal mode. You use this mode to compose commands and pass them to the MPW Shell.

Resizing a Commando Dialog Box

Once you have enabled the Commando editor, you can resize any Commando dialog box by holding down the Option key while dragging the dialog box's lower-right corner (where you would ordinarily find a size box in a standard Macintosh window). You can also resize nested dialog boxes. However, you cannot resize dialog boxes to be larger than the size of the original Macintosh screen.

Commando Controls

Controls displayed in Commando dialog boxes include checkboxes, radio buttons, boxes where you enter text, three-state controls, and others. The section "Commando Control Reference" on page 14-22 describes each of these controls in detail. This section explains how you select and modify controls that are directly editable.

Keep the following points in mind when editing a Commando dialog box:

- Lines and boxes surrounding other controls must be declared later in the resource than the controls they surround. You might encounter situations in which you have to move a control out of the way to select a control underneath.
- Controls sized or moved in nested dialog boxes do not go back to their original size or position when you click the nested Cancel button.
- When the Commando editor is enabled, any text you enter in a field that is reserved for user entry is saved as the default text. The text you enter is displayed the next time Commando is invoked.

See “Regular Entry” on page 14-22 and “Multiple Regular Entry” on page 14-23 for additional information.

To edit a control you must do the following:

1. Select the control.
2. Move or resize the control or, in the case of a text label or help message, edit the text.
3. Save the modified Commando dialog box when you are finished.

These steps are described in the following sections.

Selecting Controls

To select a control, hold down the Option key and click the control. A control that has been selected is outlined by a rectangle containing a small gray box in the lower-right corner. The small gray box is called the control’s *grow handle.*

To select multiple controls, hold down the Option and Shift keys together and click each control to be selected. You can also select a group of controls by holding down the Option and Shift keys, positioning the cursor at the corner of the group, and dragging diagonally. As you drag, a marquee (a moving dashed rectangle) appears. When you release the mouse button, the controls inside the marquee are selected.

To deselect a control, hold the Option key down and click the control.

Basically, selecting controls works exactly like selecting icons in the Finder, except that you hold down the Option key with the Shift key to make multiple selections. Because the Commando editor does not allow you to select controls outside the user control area, the coordinates you give when manually creating the Commando resource should fall within the user area.

Moving Controls

To move a control, hold down the Option key as you drag the control or a selected group of controls. The Commando editor does not allow controls to be dragged outside the user control area or closer than two pixels from the boundary of the Commando dialog box.

You can move selected controls one pixel at a time by holding down the Option key and pressing the appropriate arrow key.

You can align the top-left corner of the control to a four-pixel grid by holding down the Command key while dragging. If you hold down the Command key while dragging a selected group of controls, the top-left corners of *each* of the selected controls are aligned to the grid.

Sizing Controls

To size controls, hold down the Option key and drag the small gray rectangle in the selection box's lower-right corner (the grow handle).

To size the control's height to the recommended Commando height while sizing the control, hold down the Option and Command keys and drag the selected control's grow handle. In this case, the right edge is aligned to a four-pixel grid. List and Multiple Regular Entry controls are sized to the nearest whole line. (Table 14-1 on page 14-9 lists recommended heights for each control.)

Some controls, such as redirection controls, cannot be resized and have no grow handles.

Editing Labels and Help Messages

To edit a text label, hold down the Option key and click the text. You can change the text in the same way you change the text for an icon in the Finder. Once you have selected the text, don't hold down the Option key to change it. Text title labels are the only labels that can be edited.

To edit a help message, first select the control for which the help message is displayed; this locks its help message in the Help box. Release the Option key and click inside the Help box to edit the help message. The insertion point is placed at the location of the click. To delete the existing help text, select the help message and press Delete. The help message stays locked until you select another control or until all the controls are deselected.

Strings and MPW Shell Variables

You can dynamically change strings in Commando dialog boxes by using MPW Shell variables in place of strings. Any string in the `'cmdo'` resource, including option strings, help strings, titles, and so on, can be an MPW Shell variable. The syntax of such an entry is

"{*MPWShellVariable*}"

Observe the following when specifying an MPW Shell variable in place of a string:

- The string must begin with a left brace (`{`) and end with a right brace (`}`).
- No leading or trailing spaces are allowed.
- The MPW Shell variable must be an exported variable. If the variable is undefined at the time the Commando dialog box is invoked, the variable name with braces is displayed.
- Variables cannot be embedded within strings.

When Commando is invoked with its built-in editor, MPW Shell variable strings are not expanded to the MPW Shell variable values. This is done so that the strings can be edited and then saved as MPW Shell variables rather than as the values of MPW Shell variables.

This feature has been used in some of Projector's Commando dialog boxes to display the current user, as shown in Figure 14-2.

Figure 14-2 Using string variables in Commando resources


```
Or  {{-1}}, RegularEntry {
    "User",
    {81, 78, 96, 113},
    {81, 120, 97, 294},
    "{User}",
    ignoreCase,
    "-u",
    "Enter the name of the current user.  If"
    "no name is entered, the name in {User} is used."
{,
```

Saving the Modified Commando Dialog Box

Once you've modified a Commando dialog box and clicked the main Cancel or Do It button, Commando prompts you with a Save dialog box that gives you the options Save, Don't save, and Cancel.

When Commando saves the resource, it replaces the original resource in the resource fork of the tool or script with the modified version. The next time you run Commando, it will use the changed resource; the control positions and sizes will be where you last left them. After saving the resource, you can use DeRez to decompile the `'cmdo'` resource. This last step is not required, but is a good precaution. If the tool is damaged or lost, you can recompile the resource description file rather than having to write it from scratch.

The Structure of a 'cmdo' Resource

The `'cmdo'` resource is composed of a series of case statements that you can use to specify each kind of control you want to include in a Commando dialog box. Table 14-2 on page 14-15 presents a summary of these controls and of the case statement that governs their creation and management. The section "Commando Control Reference" on page 14-22 describes each type of control individually with sample case statements for each.

At the least, you need to specify for each control

- its location in the dialog box
- a string specifying the option or parameter that Commando is to print in the command line when the user selects that control
- a string used by Commando to display help text for the dialog box or for any command option or parameter the user can select from the dialog box

  The Help box is a three-line box. The text you decide to display in the Help box should be a concise description of what the tool, option, or parameter does. The Help box is not scrollable, so you must limit your text to the confines of the box.

Table 14-2 Controls and the `'cmdo'` resource

Control name	Case name	Description
Regular entry	`RegularEntry`	Used for editable text fields and special options that have no specialized control
Multiple regular entry	`MultiRegularEntry`	Used for editable text field values that can be entered more than once, like `-define` options
Checkbox	`CheckOption`	Used for options that have multiple settings, like `-print` options
Radio button	`RadioButtons`	Used for several mutually exclusive options
Box	`Box`	Used to draw boxes around controls or to draw lines
Text box	`TextBox`	Used to draw a box with embedded title
Text title	`TextTitle`	Used to draw text in any font
Pop-up menu	`PopUp`	Used to display lists of windows, aliases, fonts, or MPW Shell variables from which the user can choose one item
Editable pop-up menu	`EditPopUp`	Allows editing of pop-up menus associated with a text-edit box
List	`List`	Used to enable users to make multiple selections from a list of volumes, MPW Shell variables, windows, or aliases
Three-state button	`TriStateButtons`	Used to handle Set, Clear, and Don’t Touch options
Icon and picture	`PictOrIcon`	Used to place icons and pictures in Commando windows
Files	`Files`	Used to select a file or directory for input or output

continued

Table 14-2 Controls and the `'cmdo'` resource (continued)

Control name	Case name	Description
Multiple files	`MultiFiles`	Used to select multiple files and directories for input and output
Version	`Versiondialog box`	Used to place a version string in the Commando dialog box
Redirection	`Redirection`	Used to redirect standard output, diagnostic output, or standard input

In addition to the case statements describing various kinds of controls, the `'cmdo'` resource also includes three types of cases that define the dependence of one control on another. Thus, every case statement defining a Commando control must be preceded by a case statement specifying the relation of that control to one or more controls. A control can be defined to be

- independent of the setting of any other control
- dependent on the setting of one or more controls
- inversely dependent on the setting of one or more controls

The following sections explain how these kinds of dependencies affect the behavior of Commando dialog boxes and offer several examples of how to define them. You can find additional information about defining dependencies on radio buttons in the section “Radio Buttons,” beginning on page 14-26.

Parent and Dependent Controls

Sometimes one control is dependent on the value of another control. For example, a font size control might be dependent on a font selection control. In this case, the font size control is termed the **dependent control** and the font selection control is called the **parent control.**

Commando numbers each item sequentially in the order of its appearance in the resource description file. The dependent-parent relationship in a Commando dialog box is controlled by the sequential order of items entered into a `'cmdo'` resource. These numbers do not appear in the resource code; you must count them manually. You specify the dependency of one control on a parent control by specifying the number of the parent control for the dependent control.

Listing 14-1 shows the part of the 'cmdo' type declaration that defines dependencies among controls.

Listing 14-1 Dependency declarations

```
switch {
    case NotDependent:
        key int = 0;

    case Or:
        key byte = 1;
        byte = $$CountOf(OrArray);
        wide array OrArray {
            int;                        /* item number dependent upon */
        };

    case And:
        key byte = 2;
        byte = $$CountOf(AndArray);
        wide array AndArray {
            int;                        /* item number dependent upon */
        };
};
```

In the corresponding Resource statement, you would need to precede each case statement for a dependent control with a case statement describing the type of dependency.

For example, if a checkbox is not dependent on any other item, it is declared as follows; the text in boldface specifies the dependency information:

```
NotDependent { }, CheckOption {
        NotSet,
        {20, 20, 40, 200},
        "Check me",
        "-c",
        "Help us to help you.",
},
```

In the next example, the text in boldface specifies that this checkbox is directly dependent on the control defined by the second item that appears in the resource description file. If that control is enabled, the checkbox is also enabled.

```
Or { {2} }, CheckOption {
    NotSet
    {20,10,40,350},
    "Append resources to resource file",
    "-a",
    "Use this option to append the specified resource."
},
```

An item may be dependent only on other items within the same dialog box. In the case of nested dialog boxes, the items in the second and succeeding dialog boxes must be renumbered, starting from 1.

A Commando control can be dependent on more than one control. For example, a control might be enabled only when two other controls are enabled. Such situations are considered multiple dependencies.

Multiple dependencies may be of two types: OR and AND.

- In an OR dependency, a dependent control is enabled if any of its parents is enabled.
- In an AND dependency, the dependent control is enabled only if all its parents are enabled.

It is possible to mix AND dependencies and OR dependencies. For example, include an item within an AND or OR list that is dependent on a dummy control (case `Dummy`)—and make the dummy control dependent on another list of controls. An example appears in the section "Radio Buttons," beginning on page 14-26.

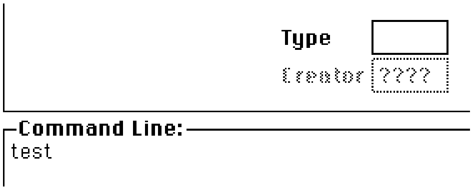
Direct Dependency

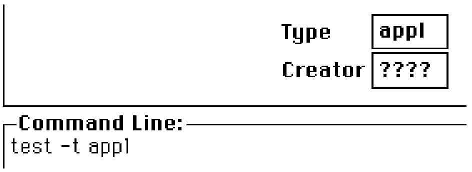
If a control is directly dependent on its parent, the dependent control is disabled if the parent control is disabled or has no value.

Figure 14-3 shows two states of a directly dependent control. In the first case, nothing has been entered in the Type field, so the dependent Creator field is disabled and appears dimmed in the dialog box. In the second case, the Creator field is enabled because something has been typed in the Type field.

Figure 14-3 also illustrates how the `ignoreCase/keepCase` flag works. Because the flag is `keepCase` and `'appl'` is not equal to `'APPL'` (the default value in this case), the option is displayed in the Command Line box.

Figure 14-3 A direct dependency



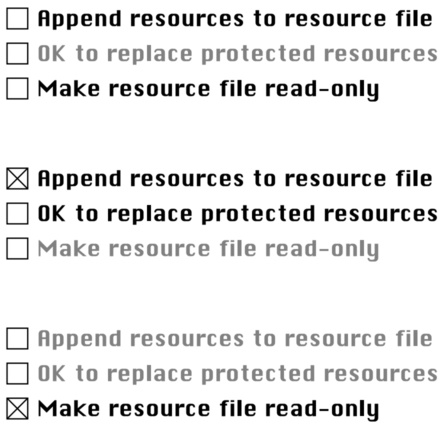


Inverse Dependency

A control can be inversely dependent on another control. In such a case, if the parent is disabled, then the dependent is enabled. Or if the parent is enabled, then the dependent is disabled. To make a control inversely dependent on another control, you make the value of the parent negative.

It is also possible for two controls to be inversely dependent on each other. This means that both controls are enabled until one is selected; then the other is disabled. For example, there are two types of dependencies illustrated in Figure 14-4. The user can select either the top checkbox or the bottom one, but not both; that is, the user is allowed to append resources to a resource file *or* to make the resource map read-only. The middle checkbox is enabled only when the top checkbox is checked because it makes sense to replace protected resources only when appending to a source file.

Figure 14-4 Inverse dependencies



Here is the resource description of the three checkboxes shown in Figure 14-4. Note the negative value of the parent control, which defines the inverse dependency.

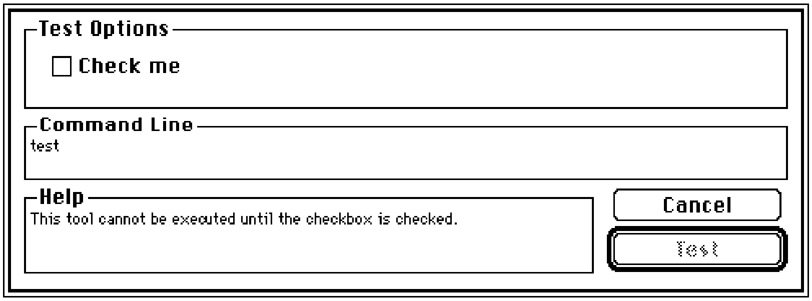
```
Or { {-3} }, CheckOption {
        NotSet,
        {20, 10, 40, 350},
        "Append resources to resource file",
        "-a",
        "some help text..."
},
Or { {1} }, CheckOption {
        NotSet,
        {40, 10, 60, 350},
        "OK to replace protected resources",
        "-ov",
        "some help text..."
},


Or { {-1} }, CheckOption {
        NotSet,
        {60, 10, 80, 350},
```

```
        "Make resources file read-only",
        "-ro",
        "some help text…"
},
```

Dependency on the Do It Button

To make the Do It button dependent on something, you must use the special `DoItButton` case in the resource definition. This case can be specified only once per resource and can be specified only for the top-level dialog box. In the example shown in Figure 14-5, the Do It button labeled Test is dependent on the checkbox.

Figure 14-5 A dependent Do It button



In the following case statements defining the dependency of the Do It button shown in Figure 14-5, the `CheckOption` case, which defines the “Check me” checkbox, is the first item in the resource; the dependent Do It button specifies the item number of the parent before the case statement for the Do It button.

```
NotDependent { }, CheckOption {
        NotSet,
        {20, 20, 40, 200},
        "Check me",
        "-c",
        "Help us to help you.",
},
Or { {1} }, DoItButton {
}
```

Commando Control Reference

The following sections describe the types of controls used in a Commando dialog box and the case statements that produce them. The controls are listed in the order shown in Table 14-2 on page 14-15. Additional information about nesting dialog boxes and redirection is presented in “Using Nested Dialog Boxes” on page 14-58 and “Redirection” on page 14-60.

Regular Entry

The regular entry control is the most generic control available. The control behaves exactly like the text-editing fields in standard Macintosh dialog boxes. In addition to its use for entering strings and numbers, you can use the regular entry control for special options that have no specified standard control.

Here is the case declaration for regular entry controls:

```
case RegularEntry:
        key byte = RegularEntryID;
        cstring;                            // title
        align word;
        rect;                               // bounds of title
        rect;                               // bounds of input box
        cstring;                            // default value
        byte ignoreCase  keepCase;          // the default value is never
                                            //  displayed in the Command window. If
                                            //  user entry matches the default value,
                                            //  then that value isn't displayed. This
                                            //  flag tells Commando whether to ignore
                                            //  case when comparing the contents of
                                            //  the text edit window with the default
                                            //  value
        cstring;                            // option returned
        cstring;                            // help text for entry
```

Multiple Regular Entry

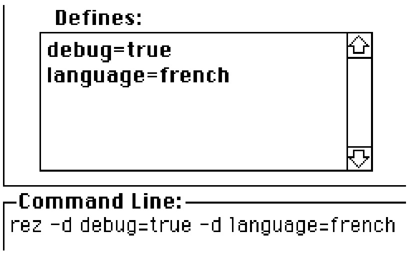
The multiple regular entry control is similar to the regular entry control, except that the multiple regular entry control accepts values that can be entered more than once. For example, most compilers accept some type of `-define` option that can be specified more than once. An example of a command line that accepts multiple defines is shown in Figure 14-6.

Here is the case declaration for the multiple regular entry control. Note that the `cstring` field for default values is the only control that passes its default values to the command line. This is an exception to the rule.

```
case MultiRegularEntry:                       // scrollable lists of an option
        key byte = MultiRegularEntryID;
        cstring;                              // title
        align word;
        rect;                                 // bounds of title
        rect;                                 // bounds of input list
        byte = $$CountOf(DefEntryList);
        array DefEntryList  {
        cstring;                              // default values

    };
        cstring;                              // option returned--each value will
                                              // be preceded with this option
        cstring;                              // help text for entry
```

Figure 14-6 shows a sample Defines window with two defines entered.

Figure 14-6 A multiple regular entry



Here is the data definition for this control:

```
NotDependent {}, MultiRegularEntry   {
    "Defines:",
    {20, 35, 35, 125},
    {40, 30, 120, 225},
    {},
    "-d",
    "Type in multiple #defines here  (such as LANGUAGE=French)"
},
```

The empty braces after the Defines window coordinates indicate that there are no default strings.

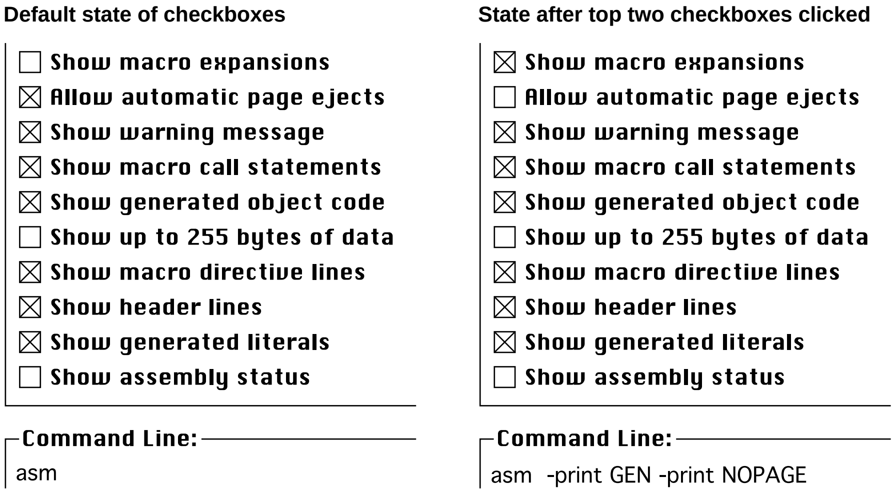
Checkbox

The checkbox control is likely to be the control used most often because it corresponds to the on/off options typical of MPW tools. Here is the case declaration for the checkbox controls:

```
case CheckOption:
    key byte = CheckOptionID;
    byte NotSet, Set;            // whether button is set
    rect;                        // bounds
    cstring;                     // title
    cstring;                     // option returned
    cstring;                     // help text for button
```

The `byte NotSet, Set` field is used to set the button's default state. The option is returned only when the button is not in its default state. Figure 14-7 shows a set of checkboxes in their default state and again after the top two checkboxes have been clicked.

Figure 14-7 Checkboxes in default state and after changes



The following case statement produces the checkboxes shown in Figure 14-7:

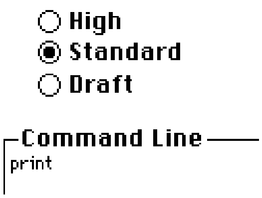
```
notDependent { }, CheckOption {
        NotSet, {20, 10, 36, 235}, "Show macro expansions",
                "-print GEN",
        "Expand macros in the listing file." },
notDependent { }, CheckOption {
        Set, {35, 10, 51, 235}, "Allow automatic page ejects",
                "-print NOPAGE",
        "Controls whether the Assembler sends automatic page ejects
                to the listing file" },
notDependent { }, CheckOption {
        Set, {50, 10, 66, 235}, "Show warning message",
                "-print NOWARN",
        "Controls both the display and count of warning messages." },
notDependent { }, CheckOption {
        Set, {65, 10, 81, 235}, "Show macro call statements",
                "-print NOMCALL",
        "Controls the listing of macro call statements." },
```

```
notDependent { }, CheckOption {
        Set, {80, 10, 96, 235}, "Show generated object code",
                "-print NOOBJ",
        "List generated object code or data for each listed line." },
notDependent { }, CheckOption {
        NotSet, {95, 10, 111, 235}, "Show up to 255 bytes of data",
                    "-print DATA",
        "Controls whether object data is shown in full
                (up to 18 lines) or limited to one line." },
notDependent { }, CheckOption {
        Set, {110, 10, 126, 235}, "Show macro directive lines",
                "-print NOMDIR",
        "Controls whether macro directives (including conditional
                and set directives) are shown in the listing." },
notDependent { }, CheckOption {
        Set, {125, 10, 141, 235}, "Show header lines",
                "-print NOHDR",
        "Controls whether header lines are printed in the listing." },
notDependent { }, CheckOption {
        Set, {140, 10, 156, 235}, "Show generated literals",
                "-print NOLITS",
        "Controls listing of literals produce by PEA and LEA machine
                instructions." },
notDependent { }, CheckOption {
        NotSet, {155, 10, 171, 235}, "Show assembly status",
                "-print STAT",
        "Controls display of assembly status in the listing." },
```

Radio Buttons

The simplest set of radio buttons offers several mutually exclusive options. For example, the Print option radio buttons shown in Figure 14-8 let you choose High or Standard or Draft. The Standard option is the default.

Figure 14-8 Radio buttons with default setting



Here is the case declaration for radio buttons:

```
case RadioButtons:
    key byte = RadioButtonsID;
    byte = $$CountOf(radioArray);                     // # of buttons
    wide array radioArray {
        rect;                                         // bounds
        cstring;                                      // title
        cstring;                                      // option returned
        byte NotSet, Set;                             // whether button is set or not
        cstring;                                      // help text for button
        align word;
        };
```

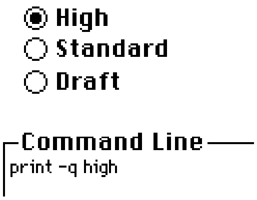
To make a button the default, specify `Set` in the `byte` field. For example, to make the middle radio button the default, as shown in Figure 14-8, declare the middle Standard button set, as follows:

```
notDependent { }, RadioButtons {
    {
        {115, 300, 130, 400}, "High", "-q high", NotSet,
            "Print the selected files in the highest quality"
            "available from the printer.",
        {132, 300, 147, 400}, "Standard", "-q standard", Set,
            "Print the selected files in the normal quality mode.",
        {149, 300, 164, 400}, "Draft", "-q draft", NotSet,
            "Print the selected files in the fastest way possible"
            "at the expense of quality."
    }
```

If one of the buttons is set, Commando accepts this as the default value, but does not display the option in the command line.

In the previous example, no option is passed to the command line because the middle button is explicitly declared the default. If a button other than the default is clicked, Commando passes the appropriate option to the command line, as shown in Figure 14-9.

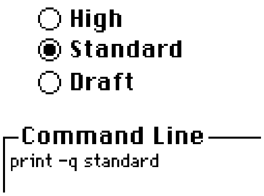
Figure 14-9 Clicking a button other than the default



If none of the buttons is set as the default, Commando assumes the first button declared to be the default value and displays that option in the command line.

With respect to the previous example, if you want the default radio button to display its option in the command line, you must change the order in which you declare the radio buttons so that the middle button is declared first. Be sure that all buttons are `NotSet`. The result is shown in Figure 14-10.

Figure 14-10 No button specified as set

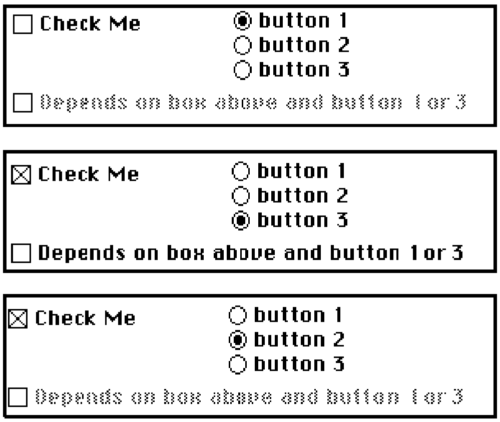


Commando considers a cluster of radio buttons to be one item. Remember that Commando numbers each item sequentially in the order of its appearance in the resource description file. When an item is dependent on a specific radio button within a cluster of radio buttons, the number of the individual button is placed in the upper 4 bits of the item number that describes the entire cluster of radio buttons. For example, consider a radio button cluster that is item 5 and contains six radio buttons. To have a dependency on button 3 you would write the following in Rez syntax:

```
(3<<12) + 5
```

Figure 14-11 shows three ways in which the checkbox at the bottom of the dialog box is dependent on the upper checkbox and radio buttons.

Figure 14-11 Dependencies on radio buttons



Here is the case statement describing the operation of the dialog box shown in Figure 14-11:

```
notDependent { }, CheckOption {
    NotSet, {15, 15, 31, 100}, "Check Me", "-root", ""
},
And { {1, 3} }, CheckOption {
    NotSet, {65, 15, 81, 450}, "Depends on box above and"
    "button 1 or 3",  "-above1", ""
},
Or { { (1 << 12) + 4, (3<< 12) + 4 } }, Dummy {
},
```

```
notDependent { }, RadioButtons { {
    {15, 150, 31, 450}, "button 1", "-b1", NotSet, "no help",
    {30, 150, 46, 450}, "button 2", "-b2", NotSet, "no help",
    {45, 150, 61, 450}, "button 3", "-b3", NotSet, "no help",
} },
```

The first `CheckOption` case is item 1 in the resource description file and the next `CheckOption` is item 2. Item 3 is a dummy item used to perform the complex dependency. Item 4 is the entire cluster of three radio buttons. Item 2 (the bottom checkbox in the sample dialog box) is dependent on item 1 (the top checkbox) and radio button 1 or radio button 3.

Boxes, Lines, and Text Titles

It is recommended that you group dialog box controls or functions within boxes. Commando supplies the facilities to draw a box (case `Box`), to draw a box with a title embedded in the upper-left corner (case `TextBox`), and to create titles in any font (case `TextTitle`).

Note
When you draw a box around a set of controls, always list the box declaration after listing the other controls. Otherwise, the Dialog Manager might confuse which control is selected when using the Commando editor. ◆

Controls defined by `Box` and `TextBox` case declarations cannot depend on other controls, nor can other controls depend on them. Commando would allow you to set up such a dependency, but the line or box would not respond to the state of the determining item. Controls defined by `TextTitle` case declarations, on the other hand, can be dependent on another control.

Box

Use the box control to draw boxes around controls or to draw lines. To draw a horizontal line, make `rect` 1 pixel high; to draw a vertical line, make `rect` 1 pixel wide. For example, to draw a horizontal line, you might set `rect` to {10, 10, 11, 100}. Here is the case declaration for box controls:

```
case Box:                                   // can be used to draw lines too
    key byte = BoxID;
    byte   black, gray;                     // color of object
    rect;                                   // bounds of box or line
```

Text Box

Use the text box control to draw a box with the title embedded in the line at the upper-left corner. This is the recommended way of naming boxes in Commando dialog boxes. (See the sample dialog box template in Figure 14-1 on page 14-8.) Here is the case declaration for text boxes:

```
case TextBox:                        // Draws a box with title in upper-left
    key byte = TextBoxID;
    byte   black, gray;              // color of object
    rect;                            // bounds of box or line
    cstring;                         // title
```

For example, this declaration gives you the results shown in Figure 14-12:

```
notDependent { },  TextBox  {
    gray,
    {105, 295, 169, 405},
    "Quality"
} ,
```

Figure 14-12 A box with an embedded title

Text Title

Use the text title control to draw text in any font. Here is the case declaration for text titles:

```
case TextTitle:
     key byte = TextTitleID;
     byte  plain;                            // style of text
     rect;                                   // bounds of title
```

```
     int  systemFont;                   // font number to use
     int  systemSize;                   // font size to use
     cstring;                           // the text to display
```

For example, you can use this case declaration

```
notDependent { }, TextTitle {
    bold + italic, {20,20,40,100},
    systemFont, 12, "So Cool..."
} ,
```

to write "So Cool..." in a cool way:

So Cool...

Pop-Up Menu

A pop-up menu offers the user a convenient way to select an item from a list of windows, aliases, fonts, or other related items. Commando manages the associated windows, aliases, fonts, and MPW Shell variables. Here is the case declaration for pop-up menus:

```
case PopUp:
    key byte = PopUpID;
    byte  Window, Alias, Font, Set;         // pop-up type
    rect;                                   // bounds of title
    rect;                                   // bounds of pop-up line
    cstring;                                // title
    cstring;                                // option returned
    cstring;                                // help text for pop-up
    byte  noDefault, hasDefault;            // hasDefault if 1st item
                                            // is "Default Value"
```

The last field, `byte noDefault, hasDefault`, tells Commando whether the pop-up menu has a default value.

- If the pop-up menu does not have a default value, the first value in the pop-up list is automatically selected and passed to the command line.

- If the pop-up menu does have a default value, then Commando adds an item of the form “Default *Value*” to the front of the list. For example, in a list of fonts, the first item is “Default Font.” When this value (such as a font or file) is selected, no value is displayed in the window that generates the pop-up menu.

Here is an example of the case statement for a pop-up menu with a default value:

```
notDependent { }, PopUp  {
    Font,
    {21, 20, 37, 60},
    {20, 60, 39, 160},
    "Font",
    "-f",
    "Popup help message",
    hasDefault
},
```

Figure 14-13 shows the resulting window title and pop-up menu.

Figure 14-13 A pop-up menu with a default value

Here is the case statement for a pop-up menu with no default value:

```
notDependent { }, PopUp  {
    Font,
    {46, 20, 62, 60},
    {45, 60, 64, 160},
    "Font",
    "-f",
    "Popup help message",
    noDefault
},
```

The pop-up menu that results is shown in Figure 14-14.

Figure 14-14 A pop-up menu without a default value

Editable Pop-Up Menu

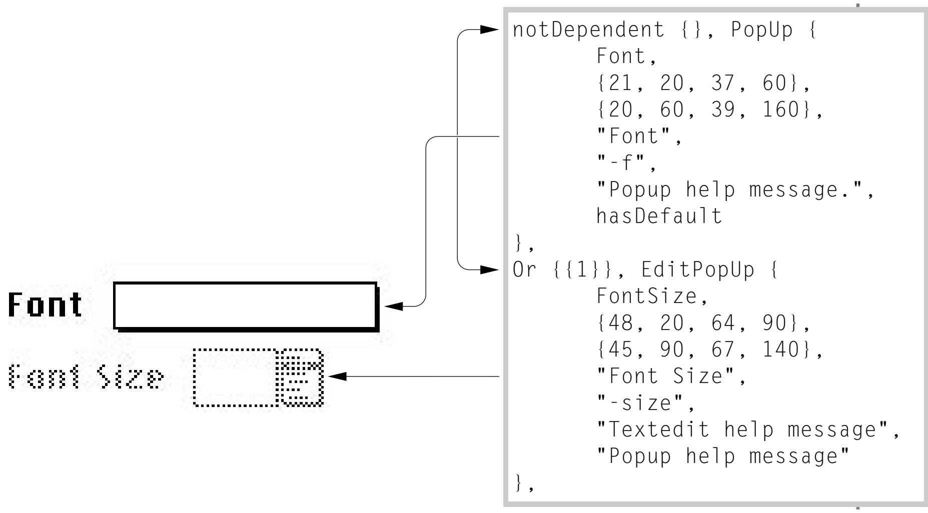
Pop-up menus associated with a text-editing box can be edited. The user can choose existing values from a list and still be able to enter completely new values. Here is the case declaration for an editable pop-up menu:

```
case EditPopUp:
    key byte = EditPopUpID;
    byte  MenuTitle, MenuItem,            // Type of editable pop-up
            FontSize, Alias, Set;
    rect;                       // bounds of title
    rect;                       // bounds of text-edit area
    cstring;                    // title
    cstring;                    // option to return
    cstring;                    // help text for text-edit
    cstring;                    // help text for pop-up
```

- If the editable pop-up menu type is `MenuItem`, the menu must be dependent on another editable pop-up menu of the `MenuTitle` type so that the control recognizes which menu item to display.
- If the editable pop-up menu type is `FontSize`, the menu must be dependent on a pop-up menu of the `Font` type so that the control recognizes which sizes of the font exist.

The example in Figure 14-15 shows how the Font Size editable pop-up menu is made dependent on the current font.

Figure 14-15 How the Font Size menu dependency is handled



If the user selects a particular font in the Font box, the font sizes that actually exist are outlined. In the example in Figure 14-16, the Monaco font has been selected in the Font box. The 9 Point item has been selected with the mouse and is highlighted. The user can type any font size in the Font Size box.

Figure 14-16 A Font Size pop-up menu with a font selected

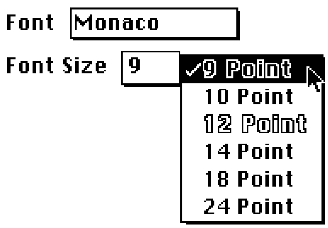
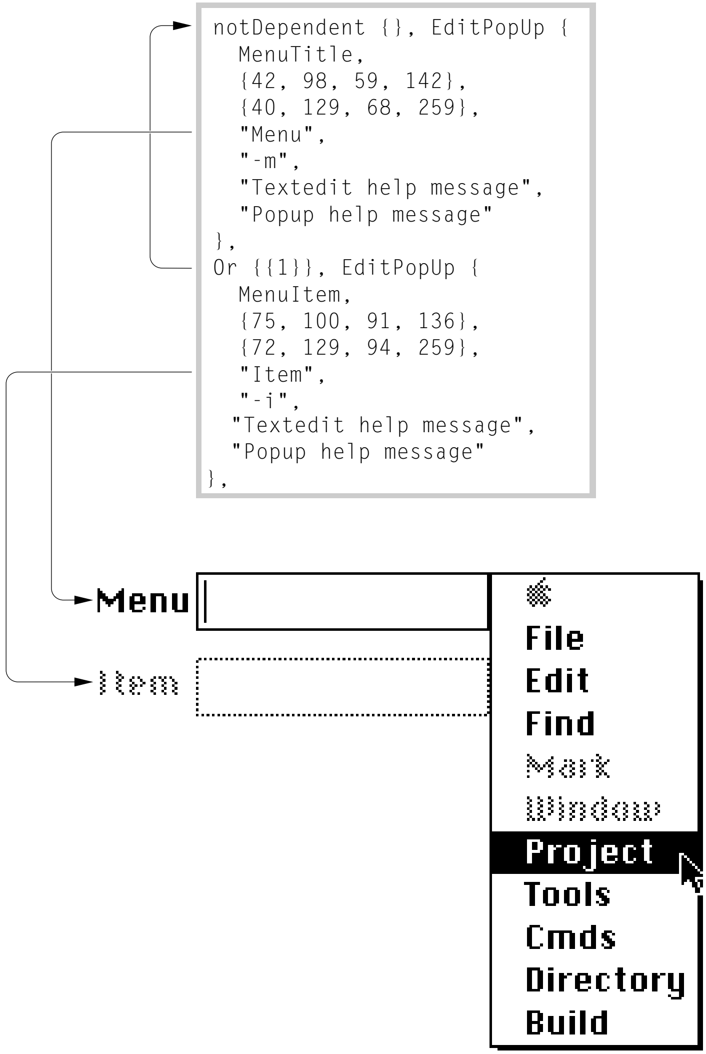


Figure 14-17 demonstrates how one editable pop-up menu can be dependent on another.

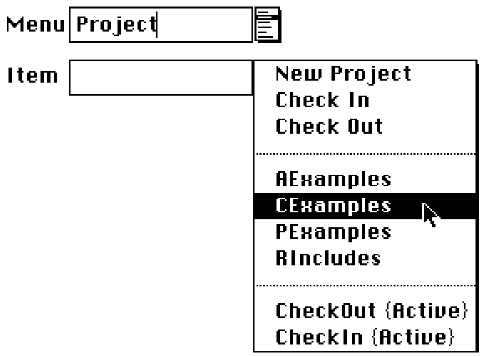
Figure 14-17 One pop-up menu dependent on another



Because `MenuItem EditPopUp` is dependent on `MenuTitle EditPopUp`, the `MenuItem` control is dimmed until the user selects a menu from the Menu pop-up or until the user types a menu name in the Menu text-editing box.

After the user selects Project (shown in Figure 14-17), the Item pop-up menu is enabled, as shown in Figure 14-18.

Figure 14-18 Menu and Item pop-up menus



List

The list controls display lists from which users can make multiple selections. This control displays a list of four types of things:

- volumes (inserted disks are recognized and added to the list)
- MPW Shell variables
- windows
- aliases

Here is the case declaration for lists:

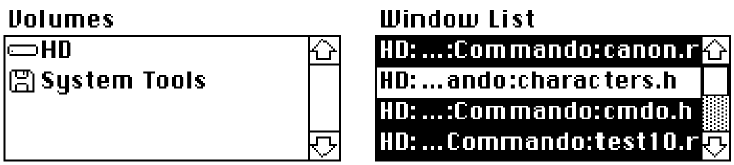
```
case List:                                     // puts up list of items & allows
                                               // multiple selections
    key byte = ListID;
    byte  Volumes, ShellVars,                  // list contents
              Windows, Aliases;
    cstring;                                   // option to return before each item
    cstring;                                   // title
    align word;
```

```
rect;                                          // bounds of title
rect;                                          // bounds of list selection box
cstring;                                       // help text for selection box
```

Here is the case statement that produces the two examples shown in Figure 14-19. The second example shows that the user has already selected a window.

```
notDependent { }, List {
    Volumes,
    "",
    "Volumes",
    {20,30,35,120},
    {37,30,101,200},
    "Help message"
},
notDependent { }, List {
    Windows,
    "-w",
    "Window List",
    {20,220,35,303},
    {37,220,101,400},
    "Help message"
},
```

Figure 14-19 List control examples



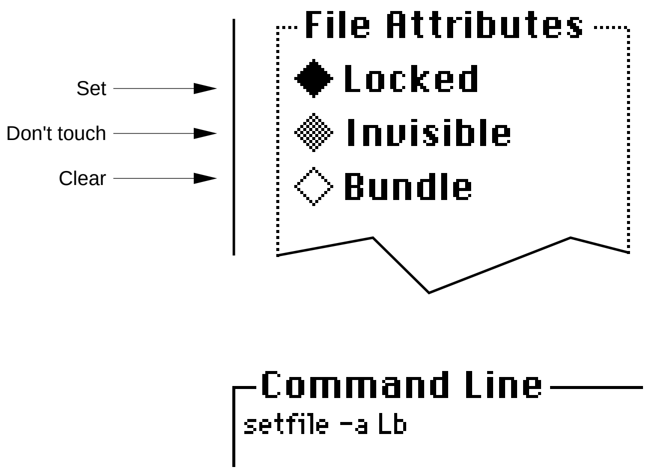
Three-State Buttons

Three-state buttons can have the states Set, Clear, and Don't Touch. They were created to handle the `SetFile` and `SetPrivilege` commands. Both of these commands deal with the setting or clearing of flags. Here is the case declaration for three-state buttons:

```
case TriStateButtons:
    key byte = TriStateButtonsID;
    byte = $$CountOf (threeStateArray);                    // # of buttons
    cstring;                                               // option returned
    wide array threeStateArray {
        align word;
        rect;                                              // bounds
        cstring;                                           // title
        cstring;                                           // for Clear state
        cstring;                                           // for DontTouch state
        cstring;                                           // for Set state
        cstring;                                           // help text for button
        };
```

Figure 14-20 shows how three-state buttons are displayed.

Figure 14-20 Three-state buttons



Here is the case statement that produces the example shown in Figure 14-20:

```
notDependent { }, TriStateButtons {
    "-a",
    {
    {40, 25, 58, 135}, "Locked", "l", "", "L",
        "This button affects the file \"Locked\" attribute.",
    {58, 25, 76, 135}, "Invisible", "v", "", "V",
        "This button affects the file \"Invisible\" attribute.",
    {76, 25, 94, 135}, "Bundle", "b", "", "B",
        "This button affects the file \"Bundle\" attribute.",
    {94, 25, 112, 135}, "System", "s", "", "S",
        "This button affects the file \"System\" attribute.",
    {112, 25, 130, 135}, "Inited", "i", "", "I",
        "This button affects the file \"Inited\" attribute.",
    {130, 25, 148, 135}, "On Desktop", "d", "", "D",
        "This button affects the file \"On Desktop\" attribute."
    }
},
```

Icons and Pictures

You can place icons, pictures, or both in Commando dialog boxes. This item cannot be dependent on any other item, nor other items on it. Here is the case declaration for icons and pictures:

```
case PictOrIcon:
    key byte = PictOrIconID;
    byte Icon, Picture;                        // display a picture or icon
    int;                                       // resource ID of icon
    rect;                                      // display bounds
```

The icon shown in Figure 14-21 is produced by an `'ICON'` resource with an ID of 0, located in the System file.

Figure 14-21 Sample icon

Here is the case statement that generates the icon shown in Figure 14-21:

```
notDependent, PictOrIcon {
    Icon, 0, {20, 20, 52, 52}
},
```

Files and Directories

There are four ways you can make files and directories available in Commando dialog boxes:

- as individual items for both input and output
- as multiple files for input only
- as multiple files and directories for input only
- as multiple new files for output

Having a file or directory available for input only means that a standard file dialog box is displayed when the command requires a file or directory on which to act. Having a file or directory available for input or output allows the user to write over an existing output file without going through the standard file dialog box.

Individual Files and Directories

You use the `Files` case to allow users to select a single file or directory that can be used for input or output. This case supports seven combinations of files as determined by the constants `InputFile`, `InputFileOrDir`, `InputOrOutputFile`, `InputOrOutputOrDir`, `OutputFile`, `OutputFileOrDir`, and `DirOnly`.

Figure 14-22 shows the declaration of the `Files` case. Because this declaration is heavily commented, it was not possible to include the comments in the figure. To see these comments, open the `Cmdo.r` file in the Interfaces folder and choose the Files item from the Mark menu.

Figure 14-22 Resource description for “individual files and directories” controls

```
case Files:
       key byte = FilesID;

       byte  InputFile,
             InputFileOrDir,
             InputOrOutputFile,
             InputOrOutputOrDir,

             OutputFile,
             OutputFileOrDir,
             DirOnly;

       switch {
             case OptionalFile:
                    key int = 0;
                    rect;
                    rect;
                    cstring;
                    cstring;
                    cstring;
                    cstring;
                    cstring;
                    byte dim, dontDim;
                    cstring;
                    cstring;
                    cstring;

             case RequiredFile:
                    key int = 1;
                    rect;
                    cstring;
                    cstring;
                    cstring;
       };

       switch {
             case Additional:
                    key byte = 0;
                    cstring;
                    cstring    FilterTypes = ":";
                    cstring;
                    cstring;
                    byte = $$CountOf(TypesArray);
                    align word;
                    array TypesArray {
                           literal longint      text = 'TEXT',
                                                 obj  = 'OBJ ',
                                                 appl = 'APPL',
                                                 da   = 'DFIL',
                                                 tool = 'MPST';
                    };

             case NoMore:
                    key byte = 1;
       };
```

Each group of files uses its own case

This case generates a pop-up

This case generates a button

Some of the elements in the listing are as follows:

- You use the `OptionalFile` case to display a pop-up menu with two or three items. The first item is used to select a default file or to select no file. The second and third items are used to select input or output files.
- You use the `RequiredFile` case to display a button; this would be appropriate when a file is required and the user does not have the choice of selecting a default file or no file.
- You use the `Additional` case when you specify the `InputFile`, `InputFileOrDir`, `InputOrOutputFile`, or `InputOrOutputOrDir` constant. This case allows you to specify additional information to have Commando display only those files with a certain extension or only files of a certain type, for example, files of type `'TEXT'` or `'MPST'`.

Listing 14-2 shows the resource code for the “individual files and directories” controls that appear in Figure 14-22.

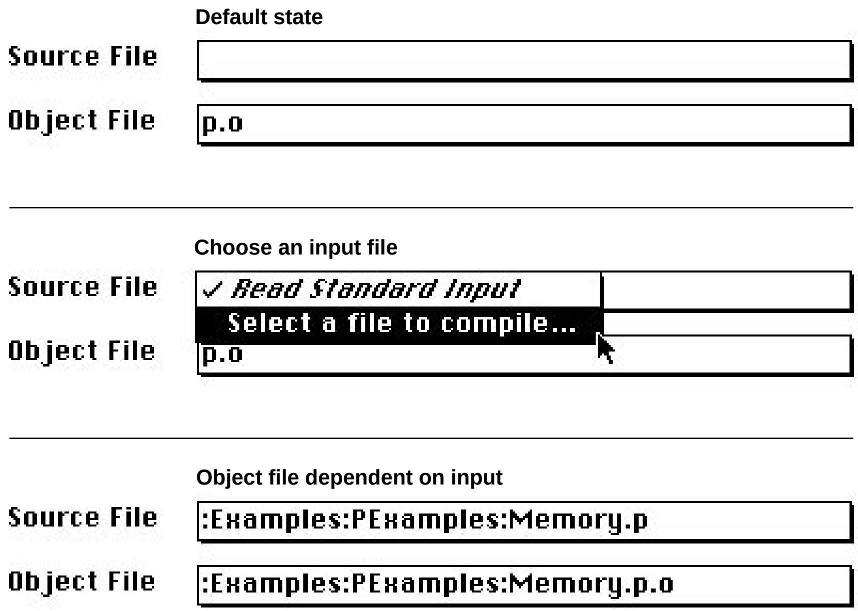
Listing 14-2 Resource code for the “individual files and directories” controls

```
notDependent { } , Files {
    InputFile,
    OptionalFile {
        {20,20,40,130},
        {20,100,40,300},
        "Source File",
        "", "", "",
        "Help message here.",
        dim,
        "Read Standard Input",
        "Select a file to compile…",
        "",
    },
    Additional {
        "",
        ".p",
        "Files ending in .p",
        "All text files",
        {text}
    },
},
```

```
Or {{1}}, Files {
    OutputFile,
    OptionalFile {
        {50,20,70,100},
        {50,100,70,300},
        "Object File",
        "p.o", "-o", ".o",
        "Help message here.",
        dontDim,
        "Send object code to p.o",
        "Select an object file…",
        "",
    },
    NoMore {},
},
```

Figure 14-23 shows the control resulting from the resource code in Listing 14-2. The control is shown first in its default state, then as it appears after the user selects an input file, and finally as it appears after Commando produces the object file associated with the input file selected by the user.

Figure 14-23 Examples of “individual files and directories” controls



Multiple Files and Directories for Input and Output

The case `MultiFiles` enables users to select multiple files and directories for input and output. This case includes four cases representing subtypes within `MultiFiles`:

- case `MultiInputFiles`
- case `MultiDirs`
- case `MultiInputFilesAndDirs`
- case `MultiOutputFiles`

Here is the type declaration for the `MultiFiles` case:

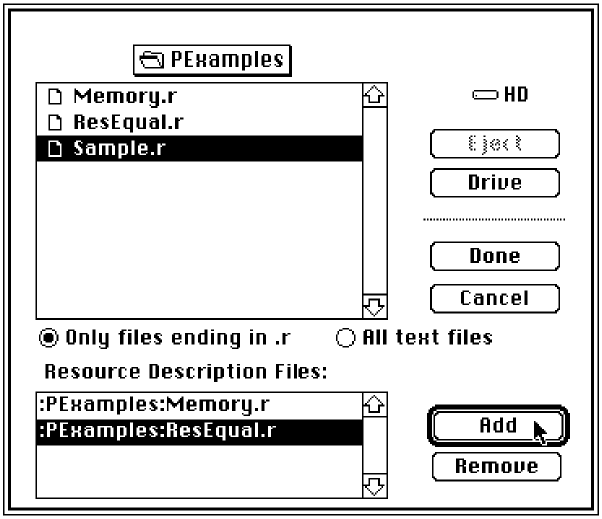
```
case MultiFiles:
     key byte = MultiFilesID;
     cstring;           // button title
     cstring;           // help text for button
     align word;
     rect;              // bounds of button
     cstring;           // message like "Source files to compile:"
     cstring;           // option returned before each filename. Null is OK
     switch {
     case MultiInputFiles:
        key byte = 0;
        byte = $$CountOf (MultiTypesArray);                 // specify up to 4 types
        align word;
        array MultiTypesArray {
            literal longint
            text = 'TEXT',              // desired input file type, do not specify a
            obj = 'OBJ ',               // type, that is {}, to display all types
            appl = 'APPL',
            da = 'DFIL',
            tool = 'MPST';
        };
        cstring FilterTypes = ":";
                                    // Preferred file extension (that is, ".c").
                                    // If null, no radio buttons are displayed.
                                    // If FilterTypes is used, the radio buttons
                                    // will toggle between showing files with only
                                    // the types below, and all files.
        cstring;                    // title of only files button
        cstring;                    // title of all files button
    case MultiDirs:
            key byte = 1;
    case MultiInputFilesAndDirs:
            key byte = 2;
    case MultiOutputFiles:
            key byte = 3;
};
```

Figure 14-24 shows a standard file dialog box controlled by resource code using the `MultiFiles` case. Here is the resource definition:

```
notDependent {}, MultiFiles {
    "Description Files…",
    "Select resource input files to compile",
    {60, 330, 80, 468},
    "Resource Description Files:", "" ,
    MultiInputFiles {
        {text},
        ".r",
        "Only files ending in .r",
        "All text files"
        },
},
```

When the user clicks the button Resource Description Files in the Rez Commando dialog box, the standard file dialog box shown in Figure 14-24 is displayed.

Figure 14-24 An example of multiple input files

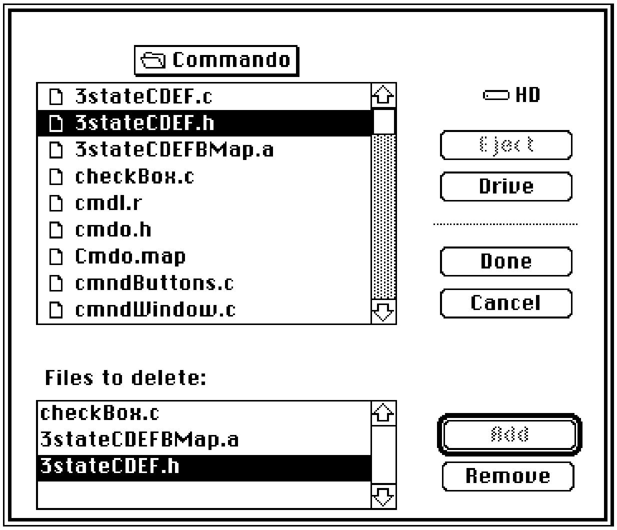


In the example in Figure 14-24, two resource files have just been added. When you specify a file extension for the `FilterTypes` field, two radio buttons allow the user to see only those files that have the specified extension or to see all files, regardless of their extension. In either case, only files that have a file type matching one of those specified in the resource are displayed. Up to four file types may be displayed. If no file type is specified, all files are eligible for display.

If no file type or file extension is specified in the `'cmdo'` resource, then no radio buttons are displayed, as shown in Figure 14-25. Here is a resource definition that does not specify a file extension:

```
notDependent {}, MultiFiles {
    "Files to delete…",
    "Select files to delete",
    {60, 330, 80, 468},
    "Files to delete:",
    "-d" ,
    MultiInputFiles {
        {},
        "",            /* no file extension specified */
        "",
        ""
    },
},
```

Figure 14-25 An example of multiple input files with no file extension specified



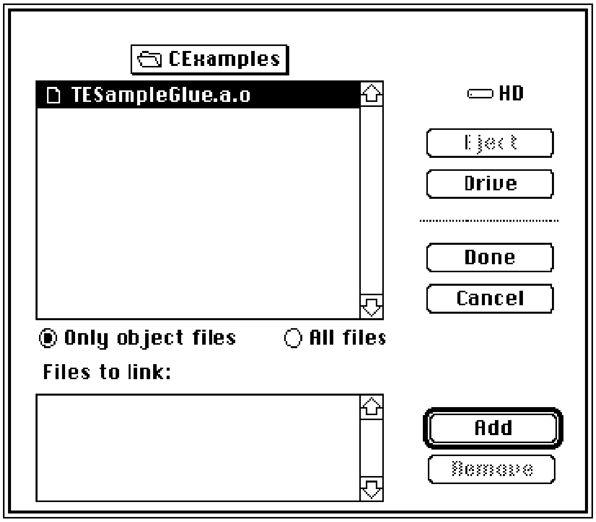
Sometimes the type of a file is more important than the file's extension. A tool could identify object files by the file type (`'OBJ '`) rather than by the file extension. If you specify a file type for the `literal longint` field and no extension for the `FilterTypes` field, the radio buttons toggle between showing files matching the specified type or types and showing all files, regardless of type.

Here is an example of how to define this behavior:

```
notDependent {}, MultiFiles {
    "Files to link…",
    "Select files to link",
    {60, 330, 80, 468},
    "Files to link:",
    "-1" ,
    MultiInputFiles {
        {'OBJ '},
        FilterTypes,
        "Only object files",
        "All files"
    },
},
```

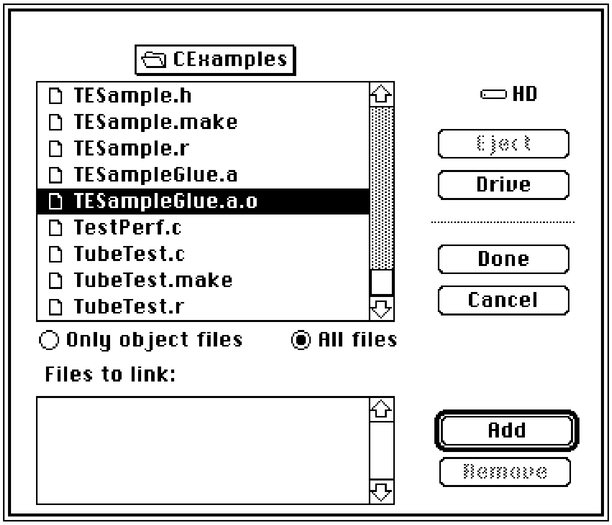
In Figure 14-26, `TESampleGlue.a.o` is the only file in the CExamples directory that has a type of `'OBJ '`.

Figure 14-26 An example of multiple input files with object files specified



After the “All files” radio button is clicked, all files in the CExamples directory are displayed, as shown in Figure 14-27.

Figure 14-27 An example of multiple input files with all files specified



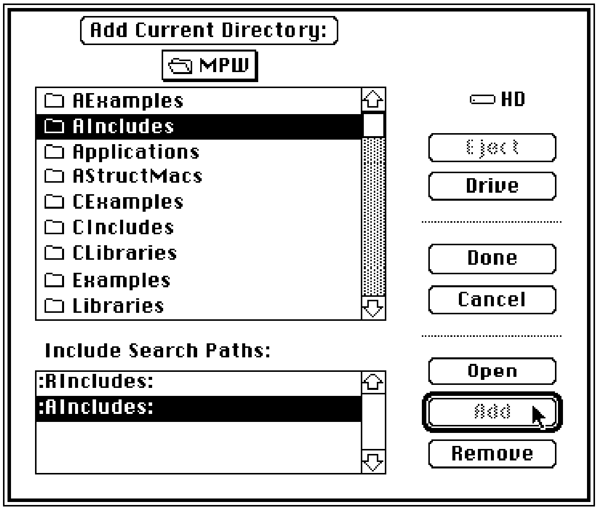
Multiple Files and Directories for Input Only

The case `MultiDirs` of the case `MultiFiles` enables the user to select multiple directories for input only. Here is the resource definition:

```
NotDependent {}, MultiFiles {
    "Include Paths…",
    "Help message for directory button.",
    110, 330, 130, 468},
    "Include Search Paths:",
    "-s",
    MultiDirs {},
},
```

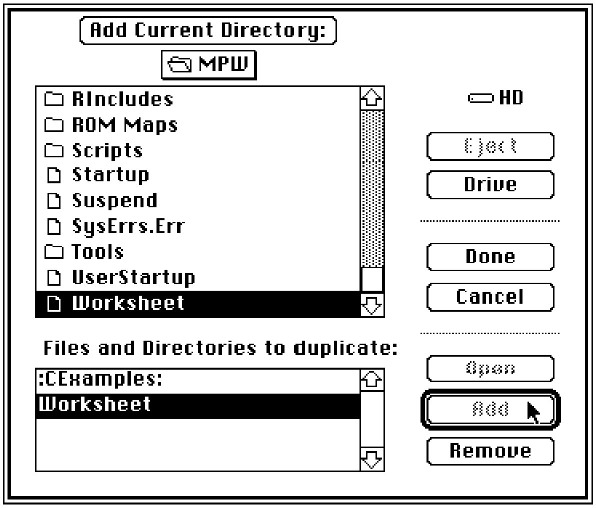
The first item in the case statement, `"Include Paths…"`, is the button in a frontmost dialog box that generates the file dialog box shown in Figure 14-28. “Include Search Paths:” is the title of the list at the bottom of the dialog box. Two folders have just been selected from the upper list and added to the Include Search Paths list just below.

Figure 14-28 Multiple directories for input



Another file dialog box that is used to select multiple files and directories is shown in Figure 14-29.

Figure 14-29 An example of a “directories” control for multiple input files



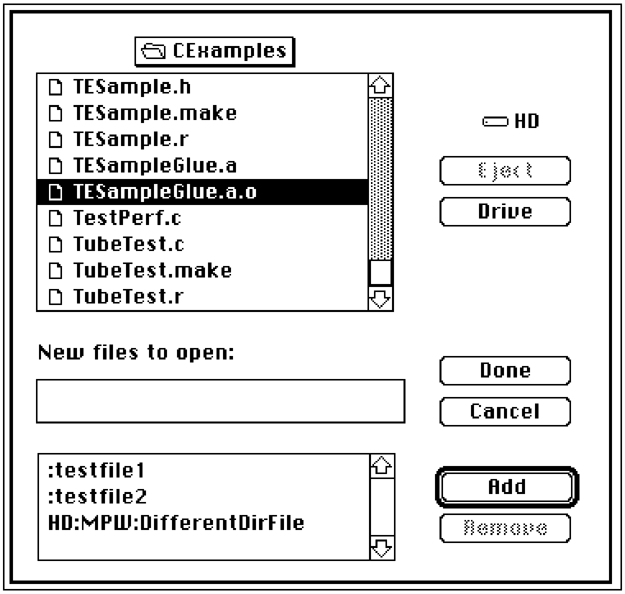
Here is the resource definition that produces the dialog box shown in Figure 14-29:

```
NotDependent {}, MultiFiles {
    "Files to duplicate…",
    "This button brings up a dialog box allowing"
        "selection of files and directories to duplicate.",
    {25, 50, 45, 230},
    "Files and Directories to duplicate:",
    "",
    MultiInputFilesAndDirs {}
},
```

Multiple New Files

The case `MultiOutputFiles` of the case `MultiFiles` allows the user to select multiple files for output. As shown in Figure 14-30, the user can use the text-editing box labeled “New files to open” to enter the name of new files that are to receive output.

Figure 14-30 An example of “directories” control for multiple output files



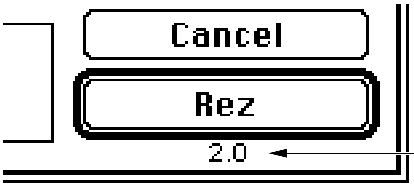
This resource code results in the example shown in Figure 14-30:

```
notDependent { }, MultiFiles {
    "New Files…",
    "Help message for button",
    {110, 330, 130, 468},
    "New files to open:",
    "-n",
    MultiOutputFiles { },
};
```

Version

You can place a version string in your Commando dialog boxes for your own identification purposes, as shown in Figure 14-31. The version string is centered below the Do It button.

Figure 14-31 Displaying a version string



To display a version number, you use the case `Versiondialog box`, whose declaration is as follows:

```
case Versiondialog box:                 // display dialog box when version #
                                        //  is clicked
    key byte = Versiondialog boxID;
    switch {
        case VersionString:             // version string embedded right here
            key byte = 0;
            cstring;                    // version string of tool (e.g. V2.0)
        case VersionResource:           // vers string comes from another res
            key byte = 1;
            literal longint;            // resource type of Pascal string
                                        //  containing version string
            integer;                    // resource id of version string
    };
    cstring;                            // version text for help window
    align word;
    integer     nodialog;               // ID of 'DLOG' resource
};
```

You should keep the following in mind when using this declaration:

- If there is no modal dialog box to display when the version string is clicked, set the resource ID to 0 (no dialog box).
- If the version string comes from another resource (`VersionResource`), the string must be the first thing in the resource, and the string must be a Pascal-style string. An `'STR '` resource is an example of such a resource.
- If the modal dialog box is to have a filter procedure, the procedure must be linked as an `'fltr'` resource with the same resource ID as the dialog box.

The version string may be embedded in the `'cmdo'` resource by using the `VersionString` case or the version string may come from a resource using the `VersionResource` case. If the version comes from a resource, the resource must contain a Rez-style `pstring`. You can use this with the SetVersion tool to read its `'MPST'` resource.

As usual, the help string is a string that is displayed when the version string is clicked. Typically, this help string contains more detailed information about the author and version.

For extra flair, a dialog box may be zoomed out when the version string is clicked. If a dialog box is specified, you must specify the resource ID of the `'DLOG'` resource to display. Commando simply calls the `ModalDialog` routine to display that dialog box.

If you want to have a custom filter procedure, you must compile the filter procedure as a stand-alone resource with a resource type of `'fltr'` and with the same ID as the `'DLOG'` resource. Set the `visible/invisible` flag in the `DLOG` resource to `invisible`. Commando moves the `'DLOG'` window so that the bounds specified by `boundsrect` in the `'DLOG'` resource are relative to the bounds of the Commando dialog box.

Note

If you do not specify a `Versiondialog box` item, Commando attempts to add one for you by looking for a `'vers'` resource with an ID of 1. If it finds this resource, Commando displays the short version string under the Do It button. When the version string is clicked, Commando displays the long version string in the help window. If a `'vers' (1)` resource is not found, Commando looks for a `'vers' (2)` resource. If one is not found, no version string is displayed. ◆

Using Nested Dialog Boxes

Complex tools may require more than one dialog box to display all the options. When there are several nested dialog boxes, all of them are called from buttons in the first dialog box. It's best to avoid calling nested dialog boxes from other nested dialog boxes.

Figure 14-32 shows how dialog box 2 can be called from dialog box 1.

All items in a nested dialog box have an implied dependency on the nested dialog box button. When a nested dialog box button is disabled (dimmed), all the controls in that nested dialog box act as if they were disabled.

Figure 14-32 Setting up nested dialog boxes

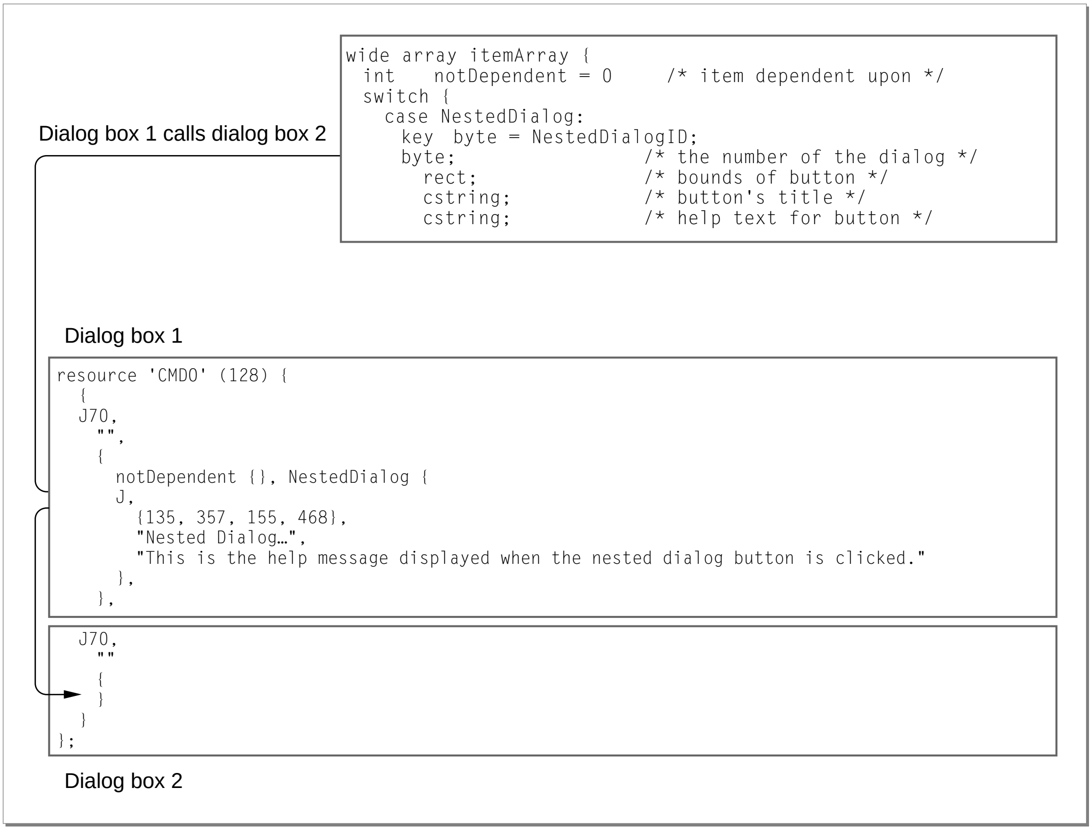
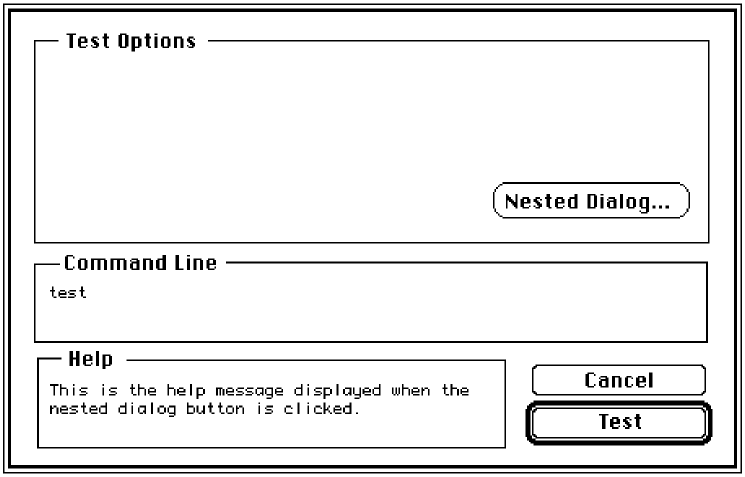


Figure 14-33 shows the recommended placement of buttons that display a nested dialog box.

Clicking the Cancel button in a nested dialog box reverts all its controls to their state before the nested dialog box was opened, thus returning the user to dialog box 1. Clicking the Do It button (typically labeled “Continue”) saves the current state of all controls in the nested dialog box and returns the user to the first dialog box.

Figure 14-33 Placement of nested dialog box buttons



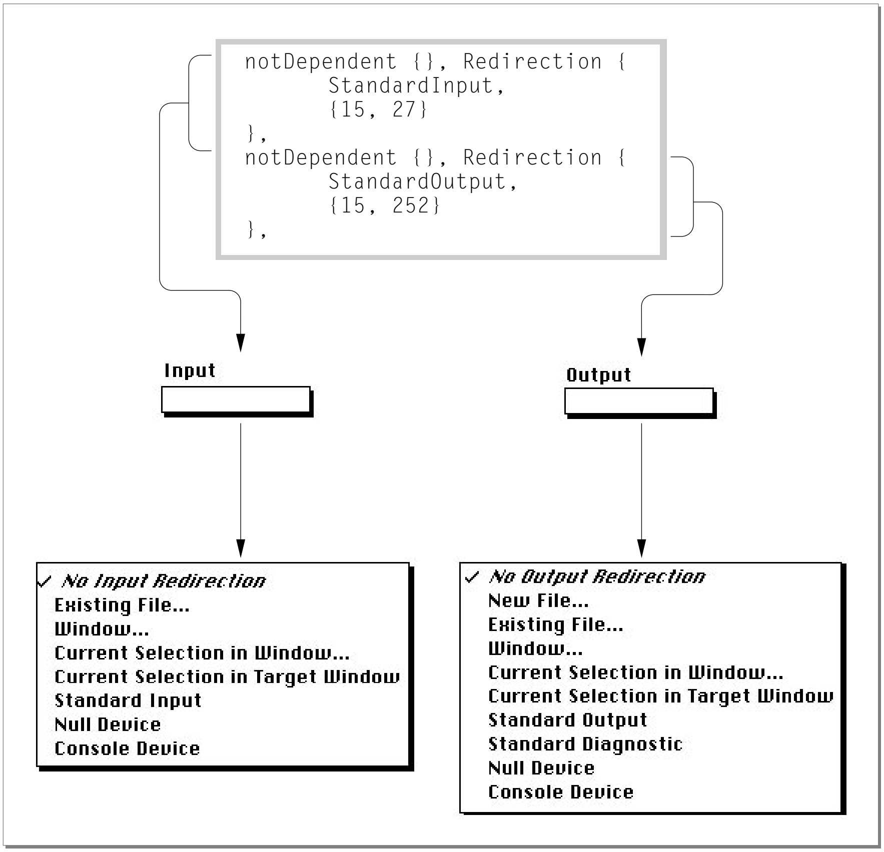
Redirection

Redirection is the easiest control to add to a Commando resource description file. If you specify the type of redirection desired and the point location of the upper-left corner, Commando takes care of the rest. Here is the case declaration for the redirection control:

```
case Redirection
    key byte = RedirectionID;
    byte StandardOutput,                // the type of redirection
        DiagnosticOutput,
        StandardInput;
    point;                              // upper-left point of the Commando dialog
```

Figure 14-34 shows the corresponding resource definition along with its results.

Figure 14-34 How to obtain input and output redirection



Note
Redirection controls have a fixed size: 112 pixels wide and 35 pixels high. ◆

A Commando Example

The best way to learn how to create Commando dialog boxes is to study an actual `'cmdo'` resource for an existing MPW tool. Choose a tool, explore the operation of the controls in its Commando dialog box, and then use DeRez to generate a readable version of the tool's `'cmdo'` resource.

To obtain the resource description file for a tool's `'cmdo'` resource, use this syntax:

```
DeRez {MPW}Tools:toolName Cmdo.r -only cmdo
```

To obtain the resource description file for an MPW Shell command's `'cmdo'` resource, use this syntax:

```
DeRez "{MPW}MPW Shell" Cmdo.r -only "'cmdo'(∂"Command-name∂")"
```

The resource description file for the ResEqual tool's Commando resource is shown here. This file, called `ResEqual.r`, is stored in the PExamples folder.

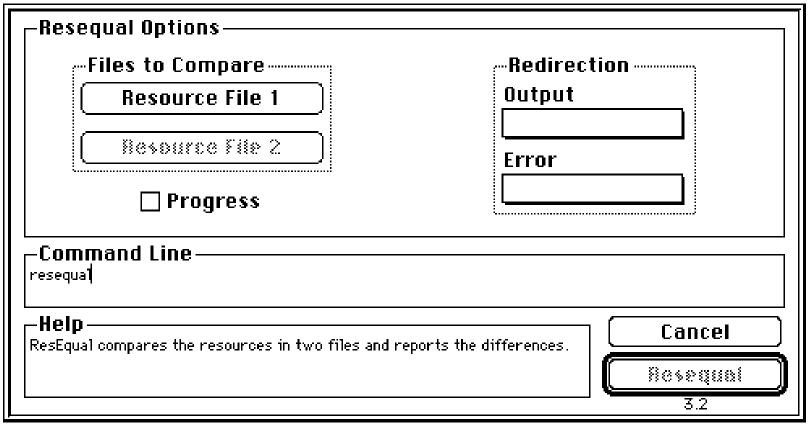
```
#include "cmdo.r"
resource 'cmdo' (355) {
    {
            240,
            "ResEqual compares the resources in two files and reports
             the differences.",
            {
                    NotDependent {}, Files {
                            InputFile,
                            RequiredFile {
                                 {40, 40, 60, 190},
                                 "Resource File 1",
                                 "",
                                 "Select the first file to compare.",
                            },
                            Additional {
                                "",
                                FilterTypes,
                                "Only applications, DA's, and tools",
```

```
                        "All files",
                        {
                            appl,
                            tool,
                            da
                        }
                    }
            },
            Or {{1}}, Files {
                    InputFile,
                    RequiredFile {
                         {70, 40, 90, 190},
                         "Resource File 2",
                         "",
                         "Select the second file to compare.",
                    },
                    Additional {
                         "",
                         FilterTypes,
                         "Only applications, DA's, and tools",
                         "All files",
                         {
                         appl,

                         tool,
                         da
                    }
                }
        },
        NotDependent {}, TextBox {
            gray,
            {30, 35, 95, 195},
            "Files to Compare"
        },
        NotDependent {}, CheckOption {
            NotSet,
            {105, 75, 121, 155},
            "Progress",
            "-p",
            "Write progress information to diagnostic"
            "output."
```

```
                },
                NotDependent {}, Redirection {
                    StandardOutput,
                    {40, 300}
                },
                NotDependent {}, Redirection {
                    DiagnosticOutput,
                    {80, 300}
                },
                NotDependent {}, TextBox {
                    gray,
                    {30, 295, 121, 420},
                    "Redirection"
                },

                Or {{2}}, DoItButton {
                },
            }
        }
};
```

The above resource code generates the frontmost dialog box of the ResEqual tool, which is shown in Figure 14-35.

Figure 14-35 A Commando example of the frontmost ResEqual dialog box



Building SIOW Applications

Contents

The SIOW (Simple Input/Output Window) package enables programs that were not originally written for the Macintosh to read from and write to a window.

After a brief description of the SIOW package, this chapter discusses

- how routines in the SIOW libraries intercept read or write calls to the console and redirect these to a window
- how to use SIOW to read from a window
- the menus displayed by SIOW
- how to create an SIOW application
- restrictions on main routines and initialization and termination routines in SIOW applications
- how to debug an SIOW application

About the SIOW Package

The SIOW package was developed to enable programs that were not originally written for the Macintosh to exhibit, at least partially, typical Macintosh appearance and behavior. You can launch a program built using SIOW as you would a Macintosh application; you double-click its icon or you select its icon and choose Open from the Finder File menu (or press Command-O to activate the Open item). Input and output interactions with the program take place in a window, and you terminate an input operation by pressing either Return or Enter.

The SIOW package contains a resource file, `SIOW.r`; a header file, `SIOW.h`, that is used by the resource file; and three libraries:

- `PPCSIOW.o` for PowerPC runtime
- `SIOW.o` for classic 68K runtime
- `NuSIOW.o` for CFM-68K runtime

The SIOW library intercepts any low-level read or write calls to the console driver. It recognizes calls using the following predefined files:

- `stdin`, `stdout`, and `stderr` for C
- `cin`, `cout`, and `cerr` for C++
- `INPUT`, `OUTPUT`, and `DIAGNOSTIC` for Pascal

The SIOW library also recognizes the file descriptors 0, 1, and 2.

I/O to other files is not affected. SIOW is designed to work with any language that can be linked with the MPW libraries.

When the console driver is first called, SIOW creates a window, places it frontmost on the screen, and displays a menu bar with the File, Edit, Font, Size, and Apple menus. This window has a zoom box, a size box, vertical and horizontal scroll bars, and bears the name Untitled until the user renames it by choosing Save As from the File menu. For additional information about the use of menus, see “Using the SIOW Menus” on page 15-7.

▲ **WARNING**
SIOW is not recommended for programs that are considered true Macintosh applications. If your program already brings up windows and handles I/O, do not use this package. ▲

How SIOW Works

Reading from and writing to files is controlled by buffers. When a buffer needs to be flushed either because it is full or because you are switching from reading to writing or writing to reading, SIOW is called. SIOW draws the needed window and prepares for the I/O command.

If you want to have SIOW show you each write as it occurs in your program, you need to flush the buffers after each I/O statement by calling `fflush`. SIOW is designed this way because flushing buffers when they are full (file buffering) is much faster than writing to the window after every line (line buffering). Remember that if you write to `stdout`, the default buffering is file buffering, whereas `stderr` is line buffered.

When writing occurs for the first time, the characters are displayed at the top of the window (left-aligned). After that, writing occurs at the end of any existing text in the window. The newline (`\n`) and tab (`\t`) characters are recognized.

SIOW provides three ways to read from the window:

- If, following receipt of output, the user types characters and presses either Return or Enter, only the typed characters are read.

  Consider these statements:

  ```
  printf("Enter your name:");
  gets(UserName);
  ```

  After the user types a response, the corresponding line might look like this:

  ```
  Enter your name:Clarus|
  ```

  The vertical bar indicates the blinking caret. When the user presses Return or Enter, the value of `UserName` is the string `Clarus`. SIOW keeps track of and returns the new characters written to the last line.

- If the user moves the insertion point by using the mouse or the arrow keys, the entire line that contains the insertion point is read.

  Consider these statements:

  ```
  printf ("hello world");
  printf ("Select the line above");
  gets (str1);
  ```

  After the user positions the cursor in the topmost line and clicks the mouse button, the lines look like this:

  ```
  hello world|
  select the line above
  ```

  The vertical bar indicates the blinking caret. When the user presses Return or Enter, the value of `str1` is the string `"hello world"`. If the caret is in any line other than the last line written, SIOW returns that entire line to the read call.

- If the user selects text and presses Return or Enter, all of the selected material is read.

  Consider the first example:

  ```
  printf("Enter your name:");
  gets(UserName);
  ```

  After the user types a response, for example,

  ```
  Enter your name:Clarus|
  ```

  if the user were to select the word `name` and press Return or Enter, the value of `UserName` would be `name`. In this case, SIOW returns the selection to the read call. (The vertical bar indicates the blinking caret.)

Building the SIOW Sample Application

The Examples:SIOWExamples folder contains an SIOW sample application, a version of the Count tool, derived from the version in the CExamples folder. You can use Count to count the characters, lines, or both in a specified list of files. In the SIOW version of Count, the parameters—whether a count of lines, characters, or both, and the list of files to be examined—are provided interactively at execution time. In the version of Count in the CExamples folder, the parameters are declared on the command line.

To build and launch the SIOW application Count, follow these steps:

1. **Use the Set Directory item from MPW's Directory menu to set the directory to SIOWExamples. This folder resides in the Examples folder.**
2. **Build the Count application by executing the following command:**

   ```
   BuildProgram Count ∑∑ "{worksheet}"
   ```

   The `BuildProgram` command uses the file `Count.make`, which is included in the SIOWExamples folder.
3. **You can now launch the Count program from the Finder as you would launch any application.**

Figure 15-1 shows the window displayed when you launch the Count application that you created in the preceding steps. The following section, "Using the SIOW Menus," describes the effect of selecting items from the menus displayed when the Count application is active.

Figure 15-1 Count SIOW application

Using the SIOW Menus

An application built as an SIOW application (for example, the Count sample application) displays five menus in the menu bar. Table 15-1 describes the File, Edit, Font, and Size menu selections. (The Apple menu is not application-specific, so it is not described here.)

Table 15-1 SIOW menus

Menu and item	Description
File Save As...	Saves the contents of the window as a file of type `'TEXT'`, creator `'????'`. A dialog box is displayed asking the user to enter the name of the new file. The saved file can be opened from MPW; an attempt to open it from the Finder results in the message "Application not found." If you open the file under System 7, the Finder offers to open it with SimpleText or TeachText.
Save	Updates a `'TEXT'` file with the current contents of the window. If nothing has changed, nothing is done. If the user has not specified a filename, the same dialog box is displayed as for the Save As item.
Page Setup...	Presents the dialog box used for the printed page layout.
Print…	Presents the Print dialog box. The name of the window and the page number are added to the bottom of each printed page. The font and font size chosen for the window are used for printing.
Quit	Quits the program by calling the `ExitToShell` routine. If the window contents were changed but not saved, a dialog box is displayed to give the user a chance to save changes.
Edit	Implements Undo, Cut, Paste, and Copy menu items in the usual way.
Font	Displays a list of fonts available in the user's system. The default font is Monaco. SIOW does not support styled fonts (outline, underline, bold, italic). Once the user selects a new font or fonts, the window is redrawn using the specified font. SIOW windows can display text in only one font at a time.
Size	Displays a list of font sizes: 9, 10, 12, 14, and 18 points. Sizes that look best with the selected font are outlined. The default font size is 9. Once the user selects a new font size, the window is redrawn using the specified font size.

Creating an SIOW Application

Assuming that a build script already exists for your program as a tool, you need to make the following modifications to build the program as an SIOW application:

1. Add the appropriate SIOW runtime library to your list of libraries to be linked:
   - `"{PPCLibraries}"PPCSIOW.o` for PowerPC runtime. This library must *replace* `StdCRuntime.o` in your command line.
   - `"{Libraries}"SIOW.o` for classic 68K runtime.This library must appear in the command line prior to `MacRuntime.o`.
   - `"{CFM68KLibraries}"NuSIOW.o` for CFM-68K runtime. This library must appear in the command line prior to `NuMacRuntime.o`.
2. Be sure that the program type is set to `'APPL'`.
3. Use the `-a` option of the `Rez` command to append the `SIOW.r` resource description file.

```
Rez -a "{RIncludes}"SIOW.r -o SIOWApp
```

Listing 15-1 shows a sample makefile for building an SIOW application.

Listing 15-1 Sample makefile for a PowerPC runtime SIOW application

```
# Makefile to build mooSIOW from    mooProg.c
#                                   mooProg.h
#                                   mooProg.r

# VARIABLE DEFINITIONS

# Define object files that PPCLink will combine.
Objects = mooProg.c.o
```

```
# Define the standard libraries to link with.
PPCLibs =   "{SharedLibraries}"InterfaceLib ∂
            "{SharedLibraries}"StdCLib ∂
            "{PPCLibraries}"PPCSIOW.o ∂
            "{PPCLibraries}"PPCCRuntime.o

# DEFAULT BUILD RULE

all ƒ mooSIOW

# COMPILE DEPENDENCIES

mooProg.c.o ƒ mooProg.c mooProg.h
     MrC mooProg.c -o mooProg.c.o

# TARGET DEPENDENCIES
# Note PPCLink options -c to set creator, -fragname to name fragment.
# Output file is type 'APPL' by default.

mooSIOW ƒƒ mooSIOW.make  {Objects} {PPCLibs}
    PPCLink -o mooSIOW ∂
        {Objects} ∂
        {PPCLibs} ∂
        -fragname mooCowApp ∂
        -c 'MOOF'

mooSIOW ƒƒ mooSIOW.make mooProg.r SIOW.r
    Rez mooProg.r SIOW.r -append -o mooSIOW
```

In addition to the three runtime architecture versions, you can also build a fat SIOW application that combines multiple runtime versions. See Chapter 5, “Building Fat Binary Files,” for information about building fat binary files.

Useful Function Pointers

The SIOW environment provides several function pointers that you may want to use when writing your SIOW application.

To use these functions, you must provide compatible function definitions and initialized function pointers in the link list prior to the SIOW library. Otherwise, these functions do not exist.

The functions are as follows:

- The "user event loop" hook:

  ```
  extern Boolean (*__siowEventHook) (EventRecord *theEvent);
  ```

  The function pointed to by `__siowEventHook` is called each time through the SIOW event loop with the most recent event returned from either `WaitNextEvent` or `GetNextEvent`. If `__siowEventHook` returns `true`, SIOW takes no further action. If it returns `false`, SIOW attempts to handle the event itself.

- The "user window is up" hook:

  ```
  extern void (*__siowWindowHook) (WindowPtr theWindow);
  ```

  The function pointed to by `__siowWindowHook` is called just after creating the window specified by `theWindow` parameter, and just prior to displaying it.

- The "user exit" hook:

  ```
  extern void (*__siowExitHook) (Boolean abort);
  ```

  The function pointed to by `__siowExitHook` is called just after completing the SIOW `atexit` routine and just before it returns. If the SIOW environment terminates normally, the Boolean `abort` parameter is set to `false`. In other cases (for example, if SIOW terminates because it runs out of memory or because it cannot open or save a window), `abort` is set to `true`.

The Main Symbol and SIOW

The SIOW libraries replace the default main entry point. An application that defines its own main symbol cannot use SIOW unless the user-defined main symbol calls the default main entry point.

The SIOW main entry point first performs all initializations done by the default main entry point. Then it initializes its own internal routines, installs an `atexit` routine to clean up, and then calls the user's `main` function.

Using Initialization and Termination Routines in SIOW Applications (PowerPC and CFM-68K Only)

SIOW initialization routines are called after standard initialization routines (that is, after the main entry point and before calling the user's `main` function). SIOW termination and cleanup occur immediately after the `main` function exits (using `atexit()`) and before the standard termination routines. This procedure means that any initialization or termination routine (for example, the constructors and destructors for C++ static objects) cannot rely on the SIOW environment.

Input read from an initialization or termination routine will be taken from a file `stdin`, if it exists. Output written from an initialization or termination routine is sent to the file system file `stdout` or `stderr` in the current directory.

IMPORTANT

SIOW closes and reopens the files `stdout` and `stderr` (redirecting output to a window) when it initializes. If you write to either file from a termination routine, you will overwrite anything written to the corresponding file by an initialization routine. ▲

SIOW initialization and termination are controlled by the StdCLib library initialization and termination routines. Note that for buffered I/O, the SIOW environment is not fully initialized (it neither initializes QuickDraw nor displays its window) until the output buffer is first flushed. For nonbuffered I/O, SIOW initialization is not completed until a character is written to the output.

Debugging SIOW Applications

You need to flush buffers when writing output statements for debugging purposes to be sure that you see such output before a program crash. Keeping in mind that `stderr` is line buffered whereas `stdout` is file buffered, to ensure that you obtain debugging output, you should send output to `stderr`, making sure that the output ends with `\n`, or include `stdio.h` in your source and follow each output to `stdout` with the call `fflush (stdout)`.

C H A P T E R 1 6

Managing Projects With Projector

Contents

Projector is an integrated set of tools and scripts used to control source files. Projector allows several users to work simultaneously on one project by regulating access to files checked in and out of the Projector database. Projector also maintains revisions to a file and comments associated with these revisions. This gives all users a complete and detailed history of changes to a file and the reasons for the changes.

This chapter begins with an overview of Projector. A tutorial follows that you can use to learn how you create and mount a project, create a checkout directory, check files into and out of the project, and obtain information about files and revisions.

Then the section "Working With Projector Files" explains how you create revision branches for parallel development and how you work with file revisions. It also describes Projector commands and options introduced in MPW 3.3 that allow you to

- obtain the history of a revision without having to mount the project
- obsolete a Projector file
- rename a Projector file

The final section, "Working With Projector—A Quick Reference," provides a summary of Projector commands not covered in the tutorial and describes Projector icons. For complete information on options to Projector commands, see the description of these commands in the *MPW Command Reference*.

Overview

Projector is a collection of built-in MPW commands and windows that help programmers (both individuals and teams) control and account for changes to all the files (documentation, source, applications, and so on) associated with a software development project. A **project** is a conceptual entity that can contain any of the following:

- all revisions of one or more files
- revision information such as author, date, and other comments
- **subprojects,** which are subsets of the larger project

Subprojects let you structure a project in a way that mirrors a given hierarchical directory structure.

At startup, the MPW Shell executes the `UserStartup` file, which causes the Project menu to be appended to the MPW Shell's menu bar. The Project menu offers you a convenient way to execute the main Projector commands required to create a project and to check files into and out of the project.

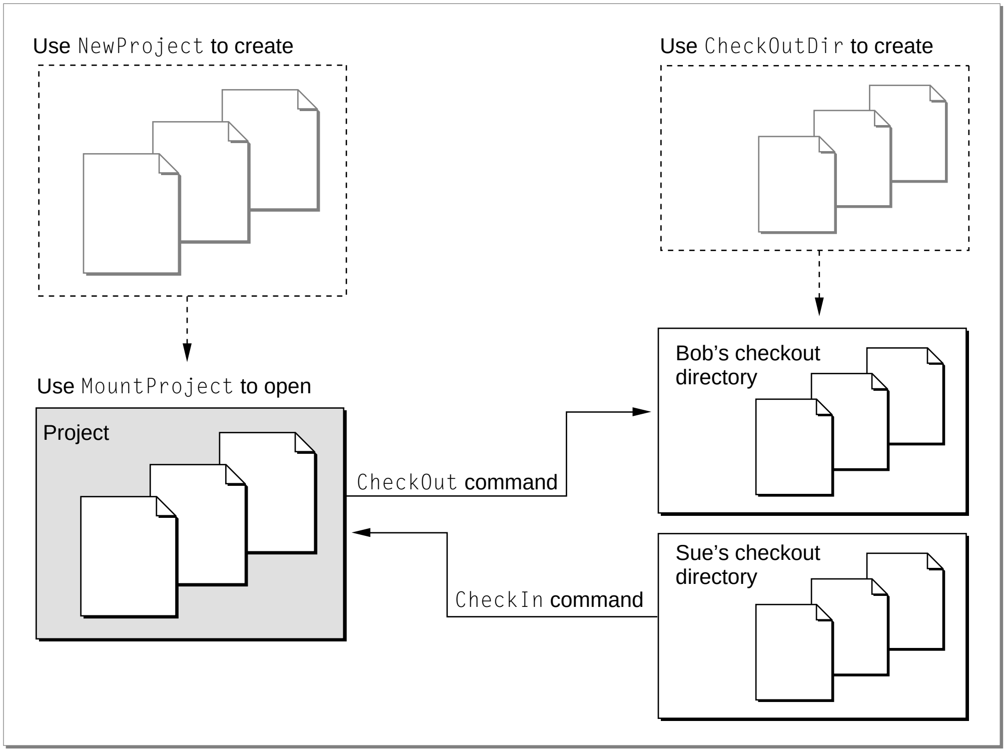
Working with Projector involves

- creating and mounting a project
- creating files and checking them into the project
- creating a checkout directory in which you place files you check out from Projector when you want to read these files or modify them
- checking files out, modifying them, and checking them back into the project

The process of checking files in and out causes Projector to build a revision tree for each file that contains all of the file's revisions. Each time a file is checked back in, its revision number is increased. You can open any revision for reading only, or you can check the revision out and modify it.

Projector allows different users to view in different ways the files maintained in its database. Each user has independent control of the mapping between the local directory, called a **checkout directory,** in which he or she keeps the files, and the hierarchy used for their storage in the Projector database. Figure 16-1 shows a project directory, a checkout directory, and the commands you use to move files from one to the other.

Figure 16-1 Checking files in and out



Projector also provides a command, `NameRevisions`, that allows you to associate a name with a specific set of file revisions, which might comprise a specific release or version of a product. You can then use the name alone to trigger the selection of just those source files required to build a version of the product.

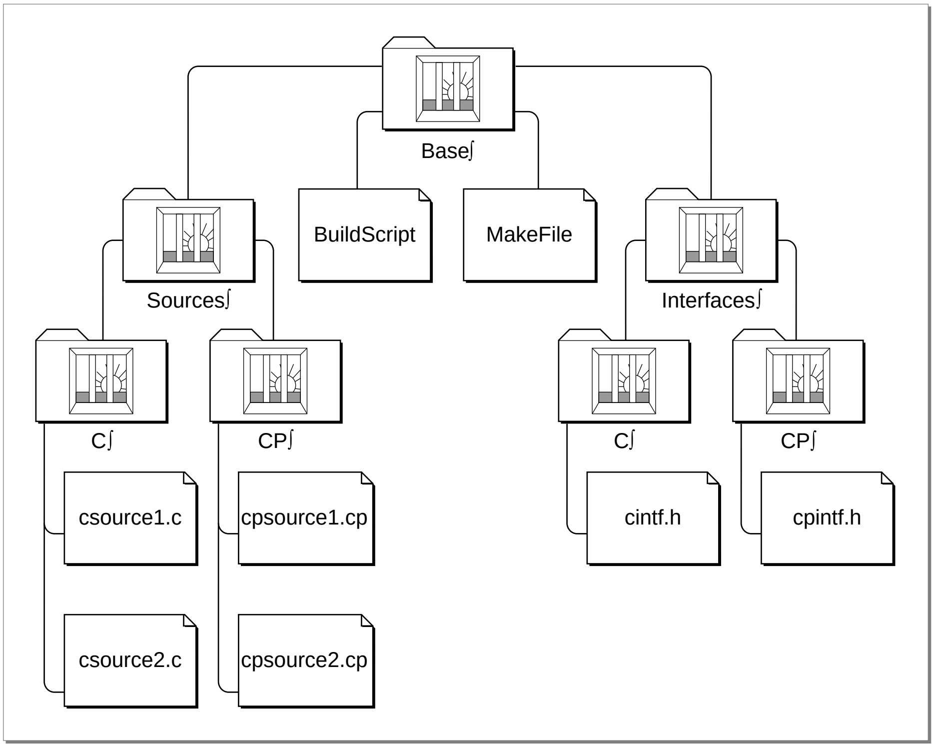
The next section describes the organization of files in a project and the format of project names.

Projects and Directories

A project is structured like a directory in a hierarchical file system in that it can contain files and other subprojects in the same way that a directory can contain files and other directories. The fact that projects can contain subprojects allows you to maintain a directory structure that reflects the structure of the project. It also allows you to check files out of the project and into your own checkout directory and maintain the same hierarchical relationships between files and directories as you have between files and projects.

The difference between a filename in your project and a filename in a directory is that the filename in a project represents the file's revision tree and is also a pointer to information about that file and information about each revision of the file, while a filename in a directory is only one version of the file. Figure 16-2 illustrates a simple project hierarchy.

Figure 16-2 Project hierarchy



The Base∫ project contains two subprojects and two files. The Sources∫ subproject contains two subprojects, each of which contains two source files. The Interfaces∫ subproject contains two subprojects, each of which contains one interface file.

Project names and directory names are formed in a similar way except that components of project names are separated by the symbol ∫ (Option-B) whereas components of directories are separated by colons. Thus the Sources∫ subproject shown in Figure 16-2 is specified as Base∫Sources∫ and the file csource1.c is specified as Base∫Sources∫C∫csource1.c. As in directory names, you can omit the terminal separator from project names; for example, Base∫Sources∫ is the same as Base∫Sources.

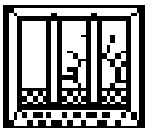
Note

Although the current project is conceptually parallel to the current directory, you cannot specify a project as a partial project name relative to the current project. That is, if the current project is Base∫, you cannot refer to the subproject Base∫Sources∫ as ∫Sources∫. ◆

When you create a project, you are actually creating a directory whose name is the project name. This directory always contains two files, one called `_CurUserName`, which is invisible to the Finder but shows up in some dialog boxes, and one called `ProjectorDB`, which contains all of the project data. If the project has subprojects, each subproject contains its own `_CurUserName` and `ProjectorDB` files.

When you open a project directory, you can see the `ProjectorDB` file for that directory. The file is represented by the icon shown in Figure 16-3.

Figure 16-3 The `ProjectorDB` file icon



Checking Out Files

Checking out a file from a project means that you are asking Projector to place the specified revision of a file into your checkout directory.

When you check out a file, Projector adds a `'ckid'` (check ID) resource to the resource fork of your file. The `'ckid'` resource identifies the file as belonging to a specific Projector project. This resource includes the file's revision number, information about whether the file is write-protected, and the text of the revision information. When you check the file back in, Projector uses the information in the `'ckid'` resource to update its revision tree for that file.

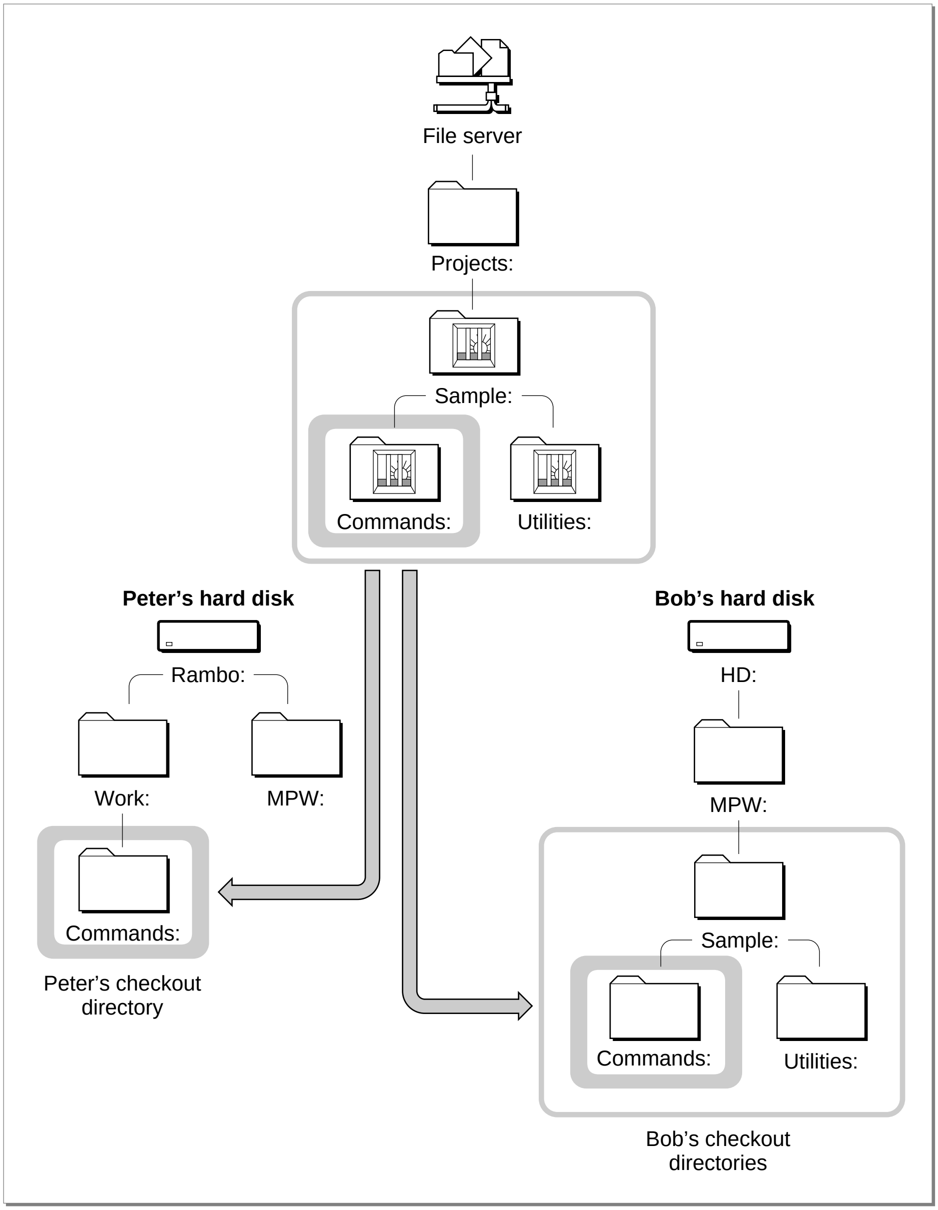
You can check files out as read-only or modifiable. If you check a file out as modifiable and you want Projector to preserve your changes, you must check the file back in. This enters the modified text as a new revision of that file in the Projector database.

Checkout Directories

Projector checks out revisions to users by placing a copy of the checked-out file in the user's checkout directory. You specify the checkout directory by executing a `CheckOutDir` command. If the file belongs to a nested project, you can use `CheckOutDir` to generate a checkout directory structure that parallels your project hierarchy. That is, if the project `Clarus` contains the subprojects `Dog` and `Cow`, `CheckOutDir` can create subfolders in your checkout directory with the same names. Then if you check out a file from the `Cow` subproject, it will automatically go into the `Cow` folder of your checkout directory.

The location of the project directory is the same for everyone, but the checkout directory can be different for each user. Figure 16-4 shows a project that is mounted on a file server. Bob and Peter can both access the file server from their local machines. Peter has created the checkout directory `Rambo:Work:Commands:`. Bob has created the checkout directory `HD:MPW:Sample:` with the `CheckOutDir` recursion option that creates the subfolders Commands and Utilities. When Peter checks out files from the Commands∫ project, they are placed by default in the `Rambo:Work:Commands:` directory. Bob's files from the Commands∫ project, on the other hand, are placed in the `HD:MPW:Sample:Commands:` directory. Note that if Bob checks out files from the Utilities∫ project, they are placed in the `HD:MPW:Sample:Utilities:` directory.

Figure 16-4 Checkout directories



Using Projector—A Tutorial

Now that you have a basic understanding of projects and checkout directories, the following tutorial shows you how to

- create a project and several subprojects
- mount the project and subprojects you have created
- create a checkout directory
- check files into a project
- obtain information about Projector files
- check out a file, modify it, and check it back in

Using this tutorial, you will construct the project hierarchy shown in Figure 16-2 on page 16-6. The tutorial should take about one hour to complete. Make sure that you do not quit MPW before you complete the tutorial to avoid having to repeat the steps required to mount the project. This tutorial assumes that the `UserStartup` file is executed when MPW is launched and that the Project menu is appended to the MPW menu bar.

Creating a Project

You will begin by creating a project and subprojects using the New Project menu item from the Project menu.

1. **Launch MPW if you have not done so already.**
2. **Enter the following command in the MPW worksheet window to create the new ProjectDemo directory:**

   ```
   NewFolder ProjectDemo
   ```

3. **Choose New Project from the Project menu.**

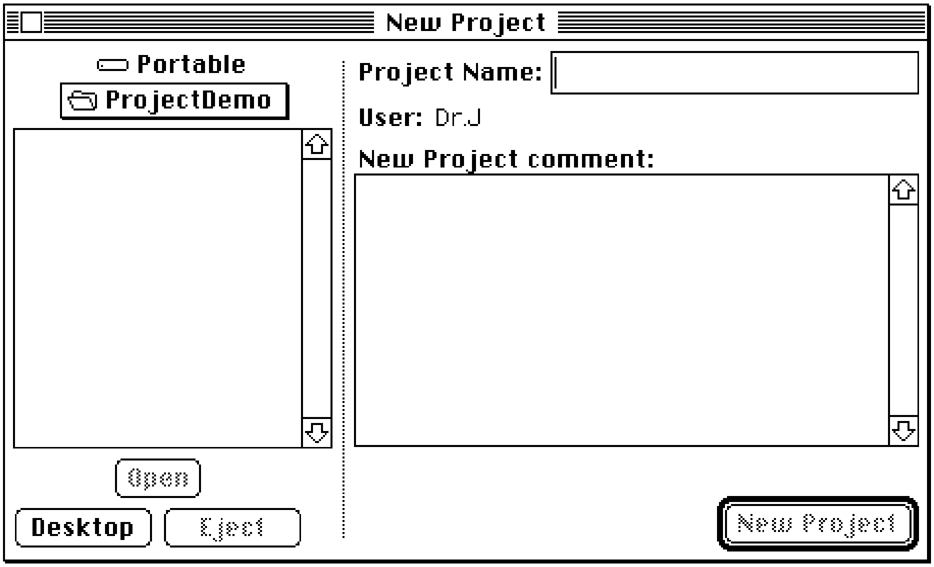
   The MPW Shell displays the New Project window, shown in Figure 16-5. This is the window you use to create a new project. The window stays open until you click the close box. You can move the window around the screen by dragging its title bar.

The right side of the New Project window contains two text-editing fields that you use to specify the name of the project you want to create and to associate comments with that project. In the right side of the window, there is also an additional field that, by default, displays the value of the MPW variable `{User}`. To change this value, enter the desired string in the MPW Worksheet window; for example, the following command sets the user name to Dr.J:

```
Set user Dr.J
```

Note that the user name changes to the desired value in the New Project window as soon as you enter the `Set` command.

Figure 16-5 The New Project window



The left side of the New Project window contains the same elements as a standard file dialog box. The current drive is shown at the top, and right below it is a pull-down list that shows you the current directory and that you can use to navigate through your directories and drives. Below the pull-down list is a window from which you can select a directory to open.

4. If ProjectDemo is not shown as the current directory, click the directory title and choose the directory you want from the pull-down list until ProjectDemo is the currently open directory.

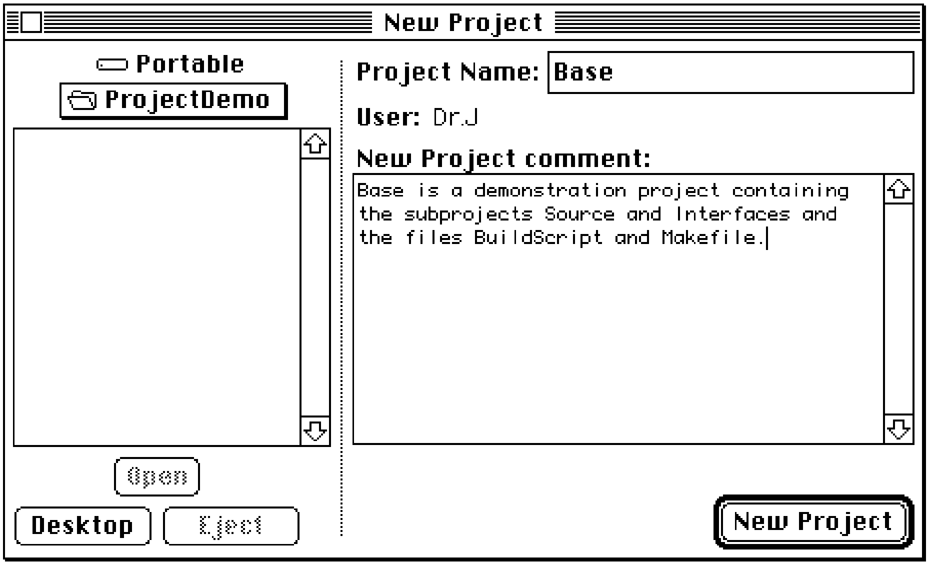
Now you will create the new project Base in the ProjectDemo directory.

5. Type `Base` in the field labeled Project Name.

6. Press the Tab or Return key to move the insertion point to the "New Project comment" box, and type a comment.

When you are finished, the New Project window should look similar to the one shown in Figure 16-6.

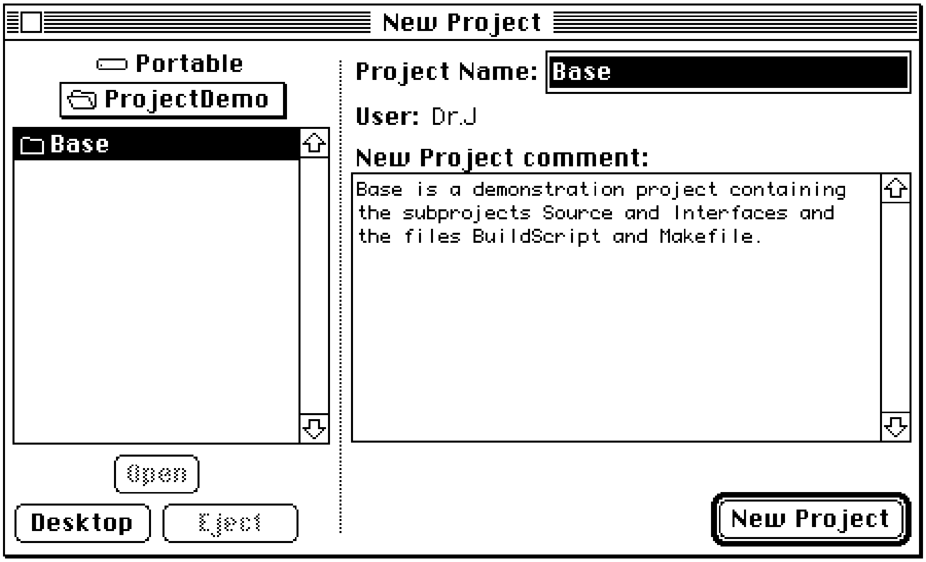
Figure 16-6 New project name and comment



7. **Click the New Project button or press Enter.**

   After clicking the New Project button, the New Project window looks like the window shown in Figure 16-7. Projector has just added a directory containing the empty database for the project Base. You can see its name in the left-hand list.

Figure 16-7 Base project



Now, you will be adding two subprojects to the Base directory.

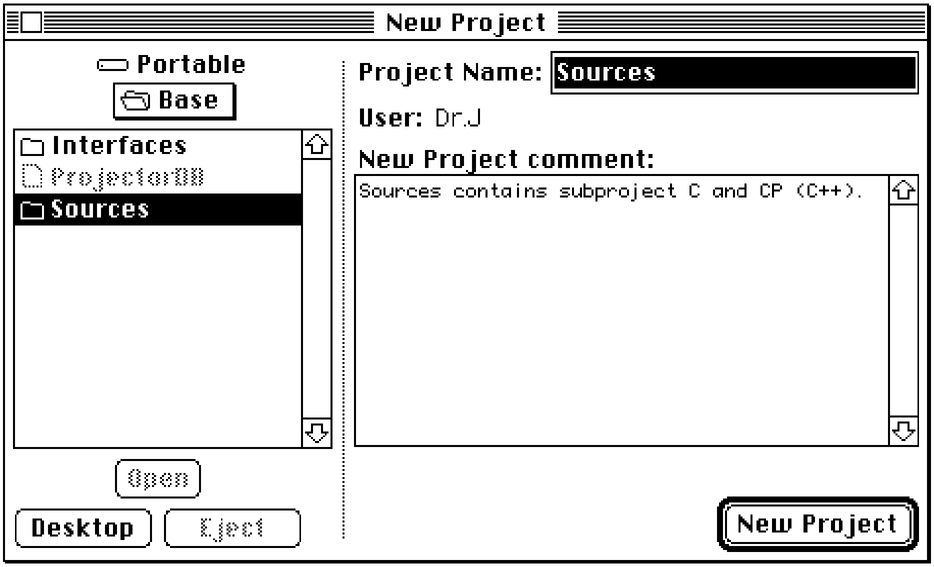
8. **Make Base your current directory by clicking Base to select it and then clicking the Open button or by double-clicking Base.**

9. **Type** `Interfaces` **in the Project Name box, add a comment if you wish, and click New Project.**

10. **Type** `Sources` **in the Project Name box, add a comment if you wish, and click New Project.**

Figure 16-8 shows how the New Project window looks after you have created the Sources and Interfaces subprojects.

Figure 16-8 Creating subprojects



Look at the list of files and directories in the left side of the New Project window. Note that the file `ProjectorDB` is visible but dimmed. Projector needs this file, but it is never accessible to the user. (Filenames are always dimmed in the New Project window.) Now, check to see how this file looks from the Finder.

11. **Return to the Finder.**

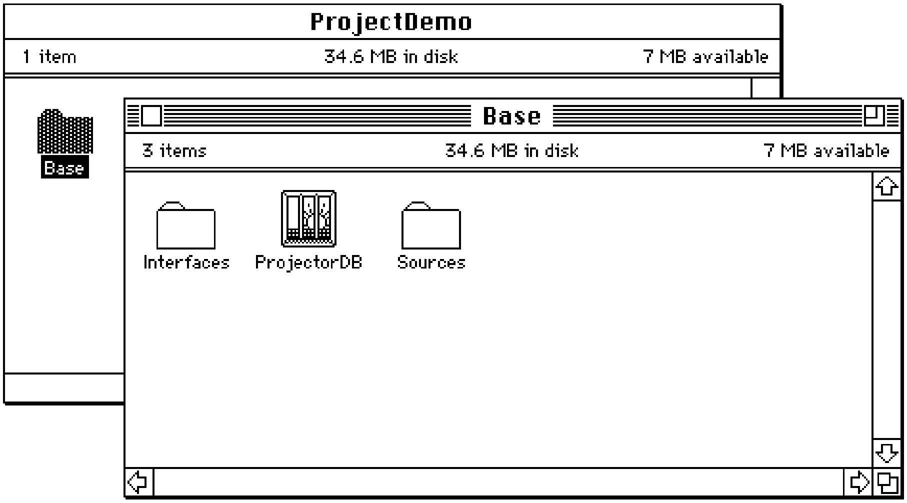
    There's no need to close the New Project window.

12. **Open the ProjectDemo folder, and then open the Base folder.**

    As shown in Figure 16-9, the Sources folder, the Interfaces folder, and the `ProjectorDB` file are visible.

    If you open the Sources and Interfaces folders, you will see a `ProjectorDB` file in each. All files and revisions checked into the Base project or the Sources and Interfaces subprojects are contained in their respective `ProjectorDB` file.

Figure 16-9 Project and subproject folders viewed in the Finder



Before continuing to the next part of the tutorial, you'll also need to create the subprojects C and CP in the Sources project and in the Interfaces project. Return to the New Project window.

13. **Make Interfaces the current project, type `C` in the Project Name box, and click New Project. Type `CP` in the Project Name box and click New Project.**

14. **Make Sources the current project, type `C` in the Project Name box, and click New Project. Type `CP` in the Project Name box and click New Project.**

Mounting a Project

Before using Projector to store, update, or retrieve files stored in projects and subprojects, you must mount these projects and subprojects. The `MountProject` command makes the MPW Shell aware of the project and makes the project data available to the user. Prior to mounting, projects are not accessible by Projector.

Since you have just used the New Project window to create your projects and subprojects, these are already mounted. Normally, when you're working with Projector, you will be accessing existing projects. However, every time you quit the MPW Shell or shut down your machine, all projects are unmounted. The next time you want to check files into or out of a project, you need to remount the projects. The following steps show you how to do that.

You can either quit and relaunch the MPW Shell to unmount your current projects, or you can use the `UnmountProject` command in the MPW Worksheet.

1. **Enter the following command in the Worksheet window:**

   ```
   UnmountProject -a
   ```

   The `-a` option to the `UnmountProject` command unmounts all currently mounted projects. Now you can practice mounting your projects.

2. **Use the Set Directory menu item to make ProjectDemo your current directory.**

3. **Enter the following command:**

   ```
   MountProject Base
   ```

   Mounting a project automatically mounts all of its subprojects, so only this one command is required. If for some reason MPW cannot mount the project, it returns an error message.

4. **To check whether the project you mounted in step 3 has been mounted, enter the following command in the MPW Worksheet window:**

   ```
   Project
   ```

   The MPW Shell displays the name of the current project, in this case Base.

   The `Project` command returns the name of the current project in the same way that the `Directory` command returns the name of the current directory. Note, however, that the value of the current directory and the value of the current project can be different. That is, using the `Project` command to set the current project does not make the current project the current directory.

   You can also use the `Project` command with a project or subproject name as an argument to set the current project. This method is demonstrated in step 5.

5. **Choose Set Project from the Project menu, and select Base∫Interfaces∫ as the current project.**

6. **Choose Set Project from the Project menu, and select Base∫ as the current project.**

   You can display a list of the mounted projects by using the `MountProject` command with no arguments.

7. **Enter the command**

   ```
   MountProject
   ```

   MPW displays

   ```
   MountProject 'Athena:MPW:ProjectDemo:Base:'
   ```

   By default, MPW returns the name of the current project preceded by the `MountProject` command. You can save these command lines and use them to mount your projects the next time you restart the MPW Shell.

8. **To display the names of all subprojects in Base∫, enter**

   ```
   MountProject -r
   ```

   Because you used the `-r` (recursive) option with the `MountProject` command, the MPW Shell displays

   ```
   MountProject 'Athena:MPW:ProjectDemo:Base:'
   MountProject 'Athena:MPW:ProjectDemo:Base:Interfaces:'
   MountProject 'Athena:MPW:ProjectDemo:Base:Interfaces:C:'
   MountProject 'Athena:MPW:ProjectDemo:Base:Interfaces:CP:'
   MountProject 'Athena:MPW:ProjectDemo:Base:Sources:'
   MountProject 'Athena:MPW:ProjectDemo:Base:Sources:C:'
   MountProject 'Athena:MPW:ProjectDemo:Base:Sources:CP:'
   ```

Relating Directories to Projects

You have now created a tree (or hierarchy) of projects into which you can check files. But before you begin to do that, you should first set up a directory structure into which you can place files you check out from your projects and subprojects. These directories are called *checkout directories*. It is best if the checkout directory duplicates the structure of your project.

You create a checkout directory by executing the `CheckOutDir` command. In its simplest form, this command takes two arguments: a project name and a directory name. The effect of executing this command is twofold:

- It sets a default checkout directory so that subsequent `CheckOut` commands addressed to the specified project copy the files to the default directory.
- It creates the directory you specify for the `CheckOutDir` command if it does not already exist.

The `CheckOutDir` command also has a very useful `-r` (recursive) option. If you specify this option, `CheckOutDir` also creates subdirectories corresponding to all subprojects and gives the subdirectories the same names as their corresponding subprojects.

In this exercise, you will create a set of checkout directories that parallels the project Base. You will also be placing the checkout directories in the same directory that contains Base, :ProjectDemo.

1. **Choose Set Project from the Project menu and set the current project to Base.**
2. **Choose Set Directory from the Directory menu to set the current directory to ProjectDemo.**
3. **In the MPW Worksheet window, enter the command**

   ```
   CheckOutDir -r -project Base∫ BaseCkOut
   ```

   If the project you want to specify is already the current project, you can use the command

   ```
   CheckOutDir -r BaseCkOut
   ```

   `CheckOutDir` uses the name of the current project by default.

To make sure the checkout directories have been created, you can return to the Finder, or you can use the `CheckOutDir` command with no arguments to display a list of the directories that correspond to the current project.

If you simply enter `CheckOutDir`, Projector displays the name of the directory that corresponds to the root project; for example:

```
CheckOutDir -project Base∫ 'Athena:MPW:ProjectDemo:BaseCkOut:'
```

If you enter `CheckOutDir` with the `-r` option, Projector displays the names of the current project, all its subprojects, and the corresponding checkout directories. The sample listing below includes artificial line breaks to fit `CheckOutDir` output into the text margins.

```
CheckOutDir -project Base∫ 'Athena:MPW:ProjectDemo:BaseCkOut:'
CheckOutDir -project Base∫Interfaces∫
    'Athena:MPW:ProjectDemo:BaseCkOut:Interfaces:'
CheckOutDir -project Base∫Interfaces∫C∫
    'Athena:MPW:ProjectDemo:BaseCkOut:Interfaces:C:'
CheckOutDir -project Base∫Interfaces∫CP∫
    'Athena:MPW:ProjectDemo:BaseCkOut:Interfaces:CP:'
CheckOutDir -project Base∫Sources∫
    'Athena:MPW:ProjectDemo:BaseCkOut:Sources:'
CheckOutDir -project Base∫Sources∫C∫
    'Athena:MPW:ProjectDemo:BaseCkOut:Sources:C:'
CheckOutDir -project Base∫Sources∫CP∫
    'Athena:MPW:ProjectDemo:BaseCkOut:Sources:CP:'
```

Using the Check In and Check Out Windows

Now that you've created your projects and checkout directories, you can begin to check files into the project by using the Projector Check In and Check Out windows. You display these windows by choosing Check In and Check Out from the Project menu.

- You use the Check In window to move a file into the Projector database. (The file does not have to reside in the checkout directory to be checked in.)
- You use the Check Out window to move a file from the Projector database into your checkout directory.

You can also use the MPW commands `CheckIn` and `CheckOut` to accomplish the same thing, but it is recommended that you use them only in scripts. The Check In and Check Out windows, like the New Project window, can be moved anywhere on the screen and remain open until you close them. The two windows are partially keyed to each other so that changing the current project in either window affects information displayed in the other window. Most of the figures in this section show both windows to illustrate how they are connected, although in practice you work with only one window at a time.

First, you need to create some files in the appropriate folders in your checkout directory so that you have actual files to check in and out.

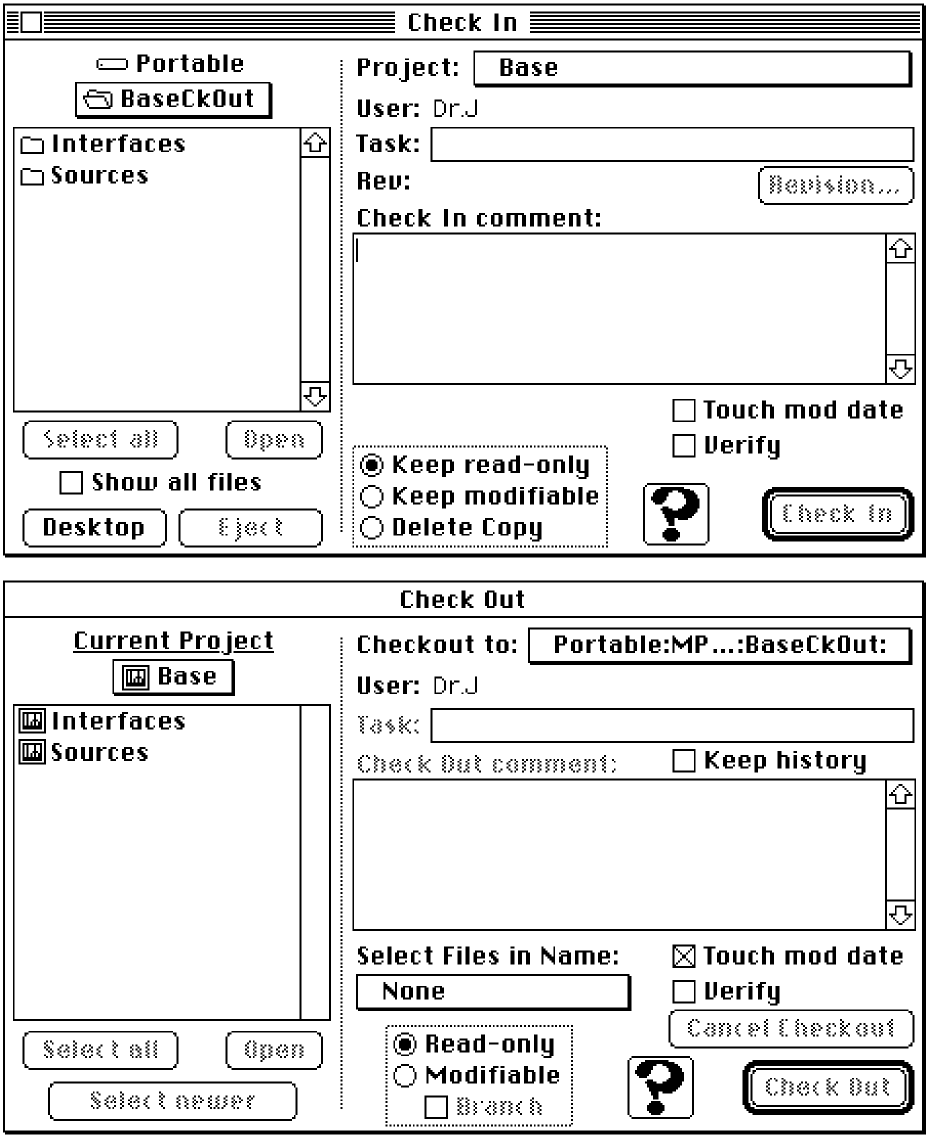
1. **Choose Set Directory from the Directory menu and make BaseCkOut your current directory.**
2. **In the MPW Worksheet window, enter the following commands:**

   ```
   echo "xxx" > MakeFile
   echo "xxx" > BuildScript
   echo "xxx" > :Sources:C:csource1.c
   echo "xxx" > :Sources:C:csource2.c
   echo "xxx" > :Sources:CP:cpsource1.c
   echo "xxx" > :Sources:CP:cpsource2.c
   echo "xxx" > :Interfaces:C:cintf.h
   echo "xxx" > :Interfaces:CP:cpintf.h
   ```

   These commands create the specified files and write the string `xxx` to them.
3. **Choose Set Project from the Project menu and make Base the current project.**
4. **Choose Check In from the Project menu, and then choose Check Out from the Project menu.**

   The MPW Shell displays the two windows shown in Figure 16-10. At the upper right of the Check In window is the Project pop-up menu. The text displayed is the name of the current project. The current project in this case is Base. If no project is mounted, the text reads “Root level projects.”

Figure 16-10 Check In and Check Out windows



5. **Press Base to see a list that contains the selectable names of all subprojects. Subprojects are indented under the projects to which they belong.**

   The left side of the Check In window contains a display that is similar to a dialog box used for opening files. Until you create a checkout directory, it merely displays the contents of the current directory. If you have created a checkout directory for the currently selected project, the name of that directory is automatically displayed on the left. As you can see in Figure 16-10, the BaseCkOut directory is shown as the directory in which you can place files checked out from the current project.

   You can freely navigate in the left-hand portion of the window and select any directory and file on any mounted volume. If you close and re-open the Check In window or if you change the current project and return to it, the current checkout directory is once again set to BaseCkOut.

   Note that the Check In and Check Out windows track each other. The current project in the Check In window is shown as the current project directory in the Check Out window. The current checkout directory in the Check In window is shown as the “Checkout to” directory in the Check Out window.

6. **Press the current project title in the Check In window and select the subproject Interfaces.**

   Note that the Check Out window now lists Interfaces as your checkout directory.

7. **Select the Base project again in the Check In window.**

   Note how the Check Out window is again updated.

   The left-hand display of the Check Out window allows you to navigate through the project hierarchy and selection of files belonging to the Projector database. Whatever project you select as the current project is also displayed in the Project pop-up menu in the Check In window.

8. **Double-click Interfaces in the project list in the left side of the Check Out window.**

   The name of the project listed in the Check In window should be Interfaces.

9. **Click the pop-up in the left side of the Check Out window, scroll down to highlight Base in the project list, and release the mouse button to select it as the current project.**

   You can change projects using either one of the windows. Projector automatically reverts to the directory specified by the `CheckOutDir` command when you deselect and reselect the project.

   You can also use the Check In window to display all files in the current checkout directory, whether or not they are checked into a project.

10. **Once more look at the Check In window as shown in Figure 16-10.**

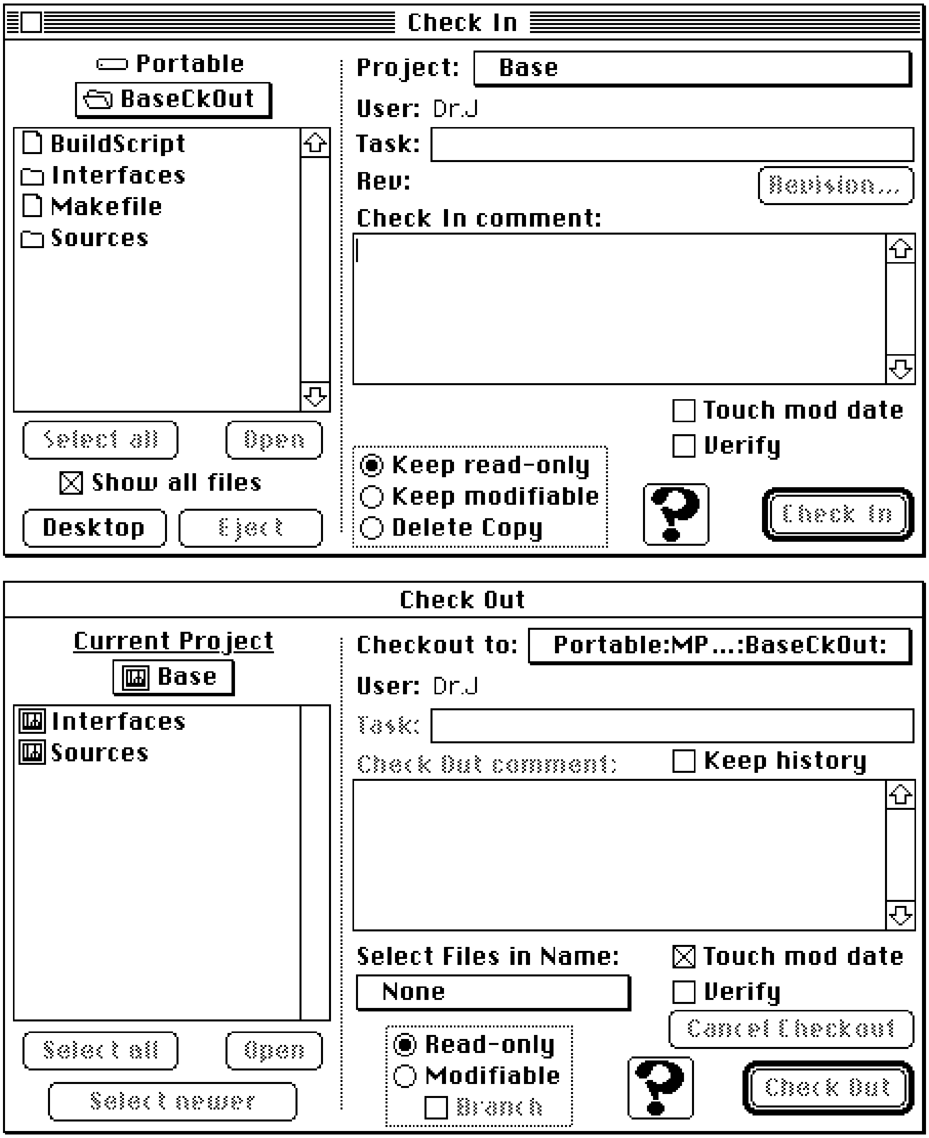
    Two subdirectories are shown in the directory list: Interfaces and Sources. No filenames are visible even though the files `BuildScript` and `MakeFile` reside in the directory BaseCkOut.

    The reason no filenames are displayed is that the Check In window by default only lists files belonging to the current project. Since the project Base is brand new, it does not yet contain any files.

11. **Click the "Show all files" checkbox in the Check In window.**

    The `MakeFile` and `BuildScript` files are now visible in the directory list, as shown in Figure 16-11.

Figure 16-11 Displaying all files in a Check In window



Now that you're familiar with the Check In and Check Out windows and how they're connected, it's time to check in a file.

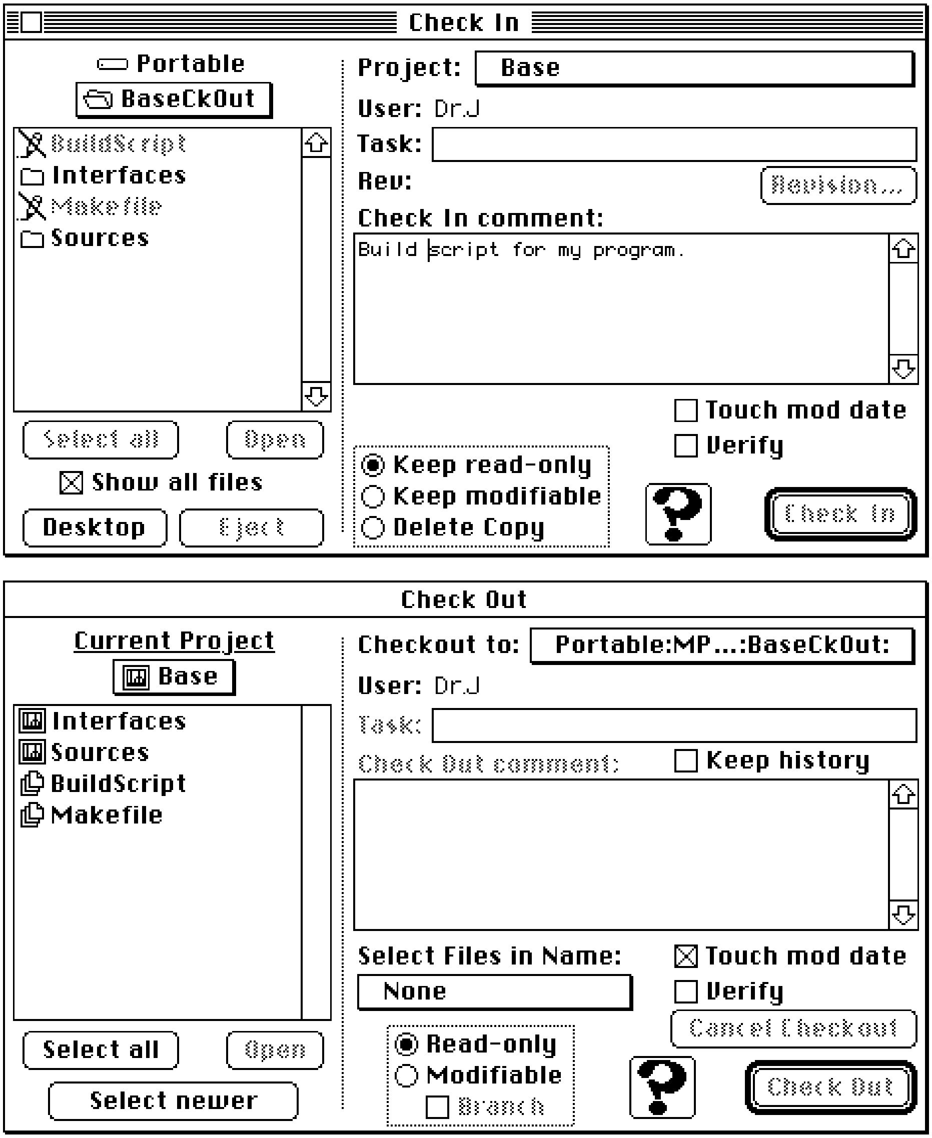
12. **Select MakeFile from the list in the Check In window, enter a comment if you want to, and click the Check In button.**

13. **Select BuildScript from the list in the Check In window, enter a comment if you want to, and click the Check In button.**

    It is not required that you enter a comment in the "Check In comment" box, but it's a good habit to get into. Maintaining a detailed history of file revisions is one advantage of using a project management system.

    After you do this, the Check In and Check Out windows should look like the ones shown in Figure 16-12. Note the icons used in both of these windows for files, directories, and projects. The section "Projector Icons" on page 16-54 provides detailed information about the meaning of these icons. In this case, because the radio buttons in the Check In window were left at their default setting (Keep read-only), the names of the two files are now dimmed, indicating that because they were checked in and because the copies in the directory BaseCkOut are now read-only, they cannot be checked in again. If you now open the copy of MakeFile or BuildScript in the checkout directory, you will see that same icon in the upper left-hand corner of the file window; any attempt to write to that file will fail.

Figure 16-12 Files that have been checked in



You will shortly be working with the source files shown in Figure 16-2 on page 16-6, but before you can do this, you need to check them into the Projector database.

14. **Check in the files you created in step 2 on page 16-20. Select each file from the list in the Check In window and click the Check In button.**

    When you are done, your subprojects should contain the following files:

    ```
    Base∫Sources∫C∫csource1.c
    Base∫Sources∫C∫csource2.c
    Base∫Sources∫CP∫cpsource1.c
    Base∫Sources∫CP∫cpsource2.c
    Base∫Interfaces∫C∫cintf.h
    Base∫Interfaces∫CP∫cpintf.h
    ```

15. **Use the list in the Check Out window to make Base the current project.**

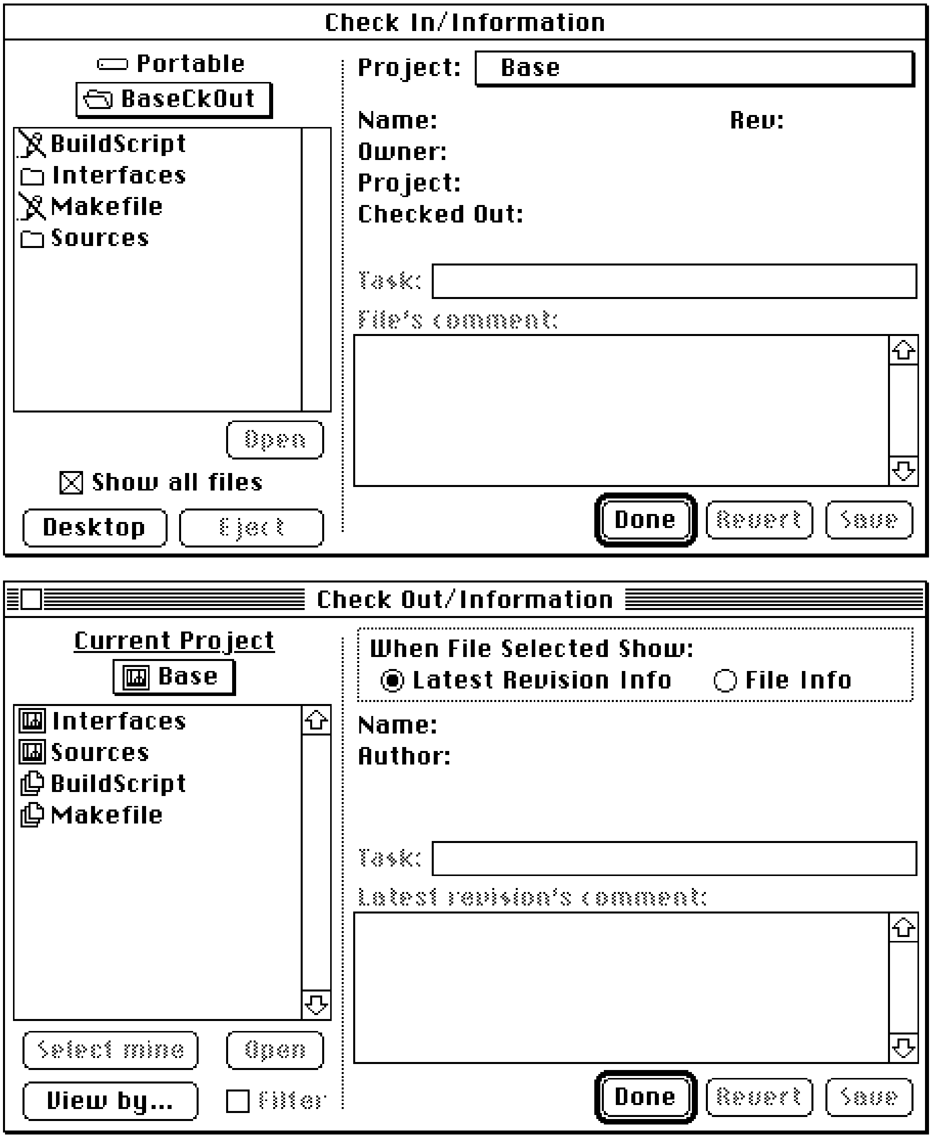
Obtaining Information About Files and Revisions

For each file revision you check into or out of a project, Projector maintains the following information: the name of the file, the owner of the file, the revision number of the file, the name of the project to which the file belongs, and the date the file was checked in or out. You can display this information by clicking the button displaying a large question mark in the Check In or Check Out window.

1. **In the Check In window, click the question mark button, and then do the same in the Check Out window.**

   Notice how the right side of each window changes to display an information window. The title of the Check In window now reads Check In/Information; the title of the Check Out window reads Check Out/Information. Some of the other buttons in these windows are also modified. The windows are shown in Figure 16-13.

Figure 16-13 The Check In/Information and Check Out/Information windows

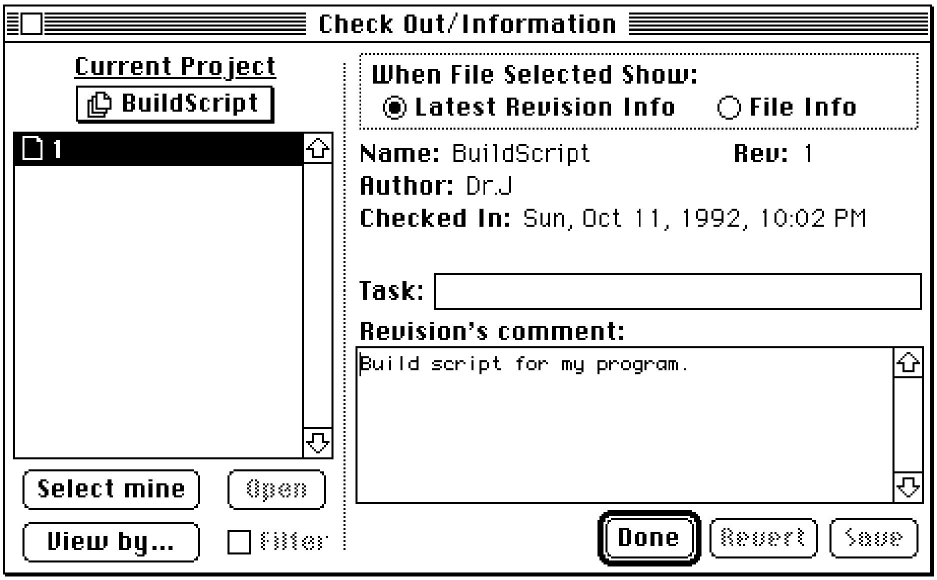


2. **Select the file** `BuildScript` **from the left of the CheckOut/Information window and click the Open button.**

   Notice how the window has changed once more; Figure 16-14 shows the new CheckOut/Information window. This window demonstrates the next level of access, that of the file revision. You see a regular document icon labeled "1," the revision number of the only existing revision. As you check the file out for modification and check it back in, additional such icons are displayed, corresponding to all existing revisions. This window allows you to fetch earlier revisions of the file.

   The Latest Revision Info and File Info buttons in the Check Out/Information window permit, respectively, the choice of a comment that applies to the specific revision selected or to all revisions contained in the file. The term *latest* for the Latest Revision Info button reflects the fact that the default selection is the latest revision; you can, in fact, obtain information for any revision by selecting the revision and then clicking the Latest Revision Info button.

Figure 16-14 The revision and file information window



Adding a File Revision

Now you're going to check out a modifiable copy of your `source1.c` file and of your `source2.c` file. (You should have already checked in these files when you followed the steps in "Using the Check In and Check Out Windows" starting on page 16-19.)

1. **Click the Done button in the Check In/Information window, and click the Done button in the Check Out/Information window.**

   In the Check Out window, the current project should still be Base.

2. **In the Check Out window, double-click Sources from the list, and then double-click C.**

3. **Select the `csource1.c` file and click the Modifiable button.**

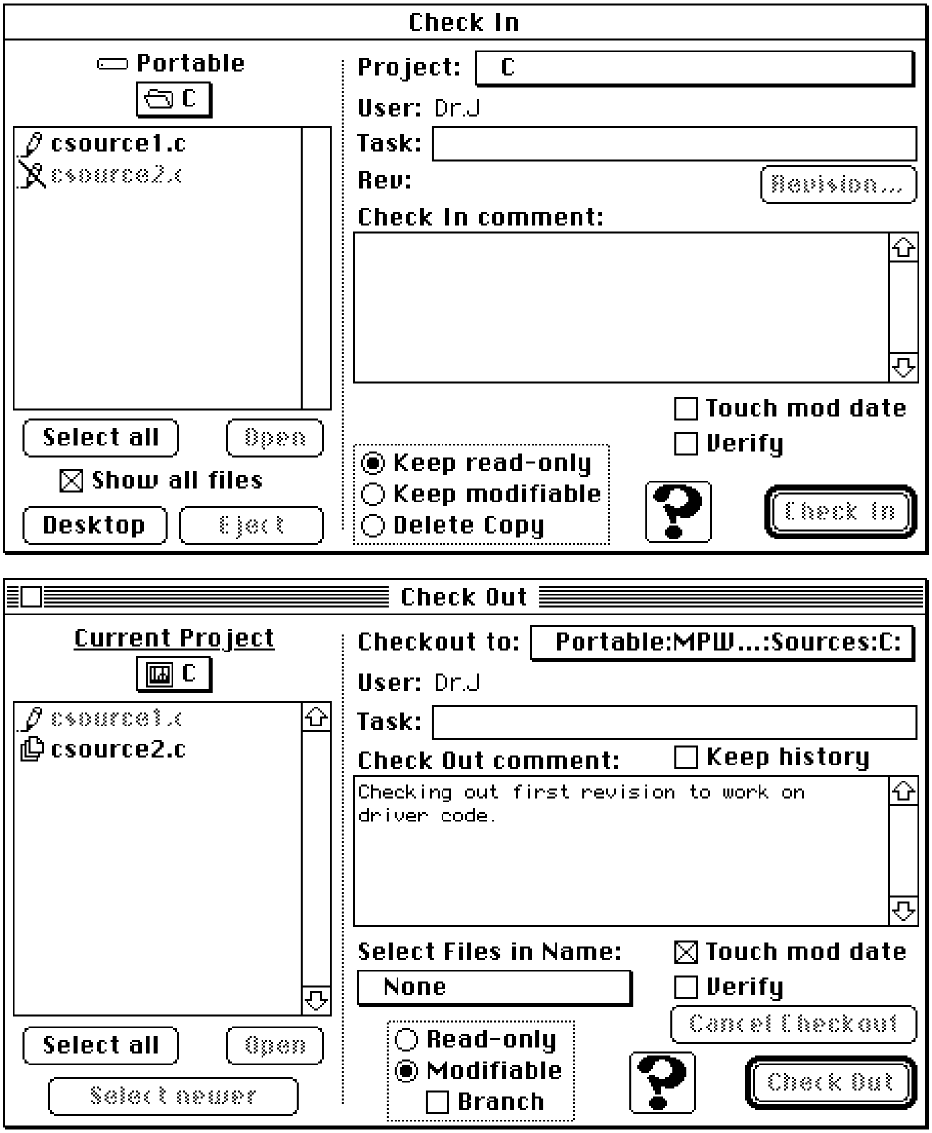
   You're checking the file out in order to work on it, so you'll want a modifiable copy.

4. **Type a comment in the "Check Out comment" box if desired.**

5. **Click the Check Out button.**

   The file is now checked out to its corresponding directory, `BaseCkOut:Sources:C`. Figure 16-15 shows the Check In and Check Out windows after you have checked out a modifiable copy of the `csource1.c` file. Note that the file icon in the Check In window has been changed to indicate that the file is present in your checkout directory and that it is no longer write-protected. The file icon in the Check Out window has also been changed to indicate that you checked out a copy of the file `csource1.c` for modification.

Figure 16-15 A checked out file



6. **Display the Check Out/Information window by clicking the question mark button, and select** `csource1.c`**.**

   Notice that the revision being modified is called "1+" to indicate that version 1 of the file is now being revised.

   When a file has been checked out for modification, the filename is dimmed in the Check Out/Information window. You can check out a read-only copy of the file, but if you want to check out a modifiable copy, you can only do so by creating a branch to the file's revision tree. To select the file, hold down the Option key while you select the filename. You can then open the file to display the revisions as before. Notice that when a file that is checked out for modifications is forced open in this way, the Branch option in the Check Out window is automatically selected.

7. **Check** `csource1.c` **back in by selecting the filename in the list in the Check In window and clicking Check In.**

   Projector displays a dialog box advising you that no changes have been made to the file and asking whether you want to check it back in.

8. **Click Yes.**

9. **Select** `csource1.c` **in the Check Out/Information window.**

   Notice that its revision level has been changed to 2.

The process of checking out a file as modifiable, editing it, and checking it back in produces what is called the *main trunk* of the revision tree: a series of file revisions numbered by default 1, 2, 3, and so on. You can use the Revision button in the Check In window to create gaps in this sequence. That is, if revisions 1 through 4 exist so that revision 5 would be created next, clicking this button makes it possible to name the next revision with a number greater than 5. For more information see "Branching" on page 16-38.

Check In and Check Out Shortcuts

Now that you have worked with the Check In and Check Out windows, it is time to present some shortcuts and additional information about working with these windows.

First, note that the action-key equivalent to clicking a default button like New Project, Check In, and Check Out is the Enter key, not the Return key. You must use the Enter key because keystrokes, including that from the Return key, are sent to the comment field in these three windows if the comment field is the current keyboard target.

Keyboard Navigation Features

Version 3.3 of the MPW Shell allows you to select files from the New Project, Check In, and Check Out windows in the same way that you select files from the standard file and Chooser dialog boxes. When a file list is a keyboard target and you type the first few characters of a file or directory name, the selection bar is moved to the matching item in the list. (In the Check Out window, this only works for files, not subprojects.) You can also use the arrow keys to move the selection bar up and down in the file list one item at a time.

- If the selection bar is on a subproject or directory, pressing Command–Down Arrow opens that item and displays the contents of the subproject or directory.
- If the selection bar is on a subproject or directory, pressing Command–Up Arrow steps up to the next higher level in the hierarchy.

To make one of the text-edit fields the active target, use the Tab key to step from target to target, or Shift-Tab to step backwards through the targets.

Table 16-1 lists the keyboard equivalents for controls in the Check In, Check In/Information, Check Out, Check Out/Information, and New Project windows. Whenever possible, the keyboard equivalents use the initial letter of the control; for example, you press Command-Shift-S for the Save button.

Table 16-1 Keyboard equivalents for Projector window controls

Window	Key combination	Equivalent to clicking
All	Cmd-Shift-O	Open button
Check In	Cmd-Shift-A	Select all button
	Cmd-Shift-D	Desktop button
	Cmd-Shift-E	Eject button
	Cmd-Shift-F	Delete Copy button
	Cmd-Shift-I	Get info (?) button
	Cmd-Shift-M	Keep modifiable button
	Cmd-Shift-N	Revision button
	Cmd-Shift-?	Get info (?) button
	Cmd-Shift-R	Keep read-only button
	Cmd-Shift-S	Show all files checkbox
	Cmd-Shift-T	Touch mod date checkbox
	Cmd-Shift-Y	Verify checkbox
Check In/Information	Cmd-Shift-D	Desktop button
	Cmd-Shift-E	Eject button
	Cmd-Shift-R	Revert button
	Cmd-S	Save button
	Cmd-Shift-S	Save button
Check Out	Cmd-Shift-A	Select all button
	Cmd-Shift-B	Branch checkbox
	Cmd-Shift-H	Keep history checkbox
	Cmd-Shift-I	Get info (?) button
	Cmd-Shift-M	Modifiable button
	Cmd-Shift-N	Select newer button
	Cmd-Shift-?	Get info (?) button
	Cmd-Shift-R	Read-only button
	Cmd-Shift-T	Touch mod date checkbox
	Cmd-Shift-Y	Verify checkbox
Check Out/Information	Cmd-Shift-F	File Info button
	Cmd-Shift-L	Latest Revision Info button
	Cmd-Shift-M	Select mine button
	Cmd-Shift-R	Revert button
	Cmd-S	Save button
	Cmd-Shift-S	Save button
New Project	Cmd-Shift-D	Desktop button
	Cmd-Shift-E	Eject button

Canceling File Modifications

If you have checked out a file for modification, selecting the file in the Check Out window activates the Cancel Checkout button. If you click this button, Projector changes the status of the file to read-only if it is in the checkout directory and discards any changes made to the file while it was modifiable. Projector displays a dialog box that allows you to confirm your decision.

Selecting File Revisions

To select and open a file at the same time, press the Option key while clicking the Check Out button.

To select more than one file in the Check Out or Check In window, do the following:

- For a contiguous selection, select the first file by clicking it. Then hold down the Shift key and click a second filename. This selects the first file, the second file, and all intervening files.
- For a discontinuous selection, select the first file by clicking it, then hold down the Command key and click the other files you want to select.

To select the latest revision on the main trunk of all files in the current project, click the “Select all” button in the Check Out window. This does not check out these files, it only selects them. To check out the selected files, click the Check Out button as usual.

Another way of selecting the latest revisions is to click the “Select newer” button in the Check Out window. In this case, Projector selects those files whose newest (nonbranch) revision is not already in your checkout directory, except for any revision that is on a branch. The assumption is that if you have checked out a branch, you intend to keep it.

Conversely, if you hold down the Option key while clicking “Select newer,” Projector selects revisions of those files only if a copy of those files already resides in your checkout directory. This is equivalent to using the `CheckOut` command with the `-update` option. The sense is this: Select file revisions for checkout by the same criterion as “Select newer,” but do not check out revisions of any files that have not already been checked out. Just update the files that you have already checked out.

Setting a File's Modification Date

If you select the "Touch mod date" checkbox in the Check In window, Projector sets the date of latest modification in the file system directory to the time the file is checked in. If you select the "Touch mod date" checkbox in the Check Out window, Projector sets the date of latest modification in the file system directory to the time the file is checked out.

By default, the "Touch mod date" checkbox is selected in the Check Out window and not in the Check In window. Although this can cause unnecessary revisions of this date, it guarantees an update every time a user checks out the file, meaning that tools such as Make always assume they are being presented with a new version. If this default is not used and more than one person is working on a file, it is possible for a user to check out a revised file and execute a makefile that contains a dependency on that file without Make realizing that the file has been updated.

Identifying Revisions That Have Been Checked Out

To determine which revisions of the files are currently checked out of a project, open the Check Out/Information window. Select the file from the list, and open the file in question to display its revision tree. Click the "Select mine" button. If a revision of that file exists in its checkout directory, that revision is selected in the list.

Working With Projector Files

If you finished the preceding tutorial, you should now be familiar with creating and mounting projects and checking files into and out of projects.

This section describes the way in which you can organize files in a project. It explains how Projector keeps track of project files by using the `'ckid'` (check ID) resource and how you

- create branches in the revision tree of a project, allowing parallel development to occur
- compare file revisions
- delete file revisions
- name a set of file revisions

- obtain information about files and revision
- obtain a revision's history without mounting the project

The 'ckid' Resource

Projector maintains a `'ckid'` (check ID) resource in the resource fork of all files that belong to a project. Projector creates this resource the first time a file is checked in and uses the resource to identify each file's name, project, user, revision number, and so on. When you check out a file, Projector includes the `'ckid'` resource in the resource fork of the file.

The Check In/Information window displays Projector's current information on the selected file (that is, the contents of the file's `'ckid'` resource). If you are checking in a new file, the corresponding Check Out/Information window is blank because Projector has not yet created a `'ckid'` resource for it. If you are checking in a revision that was checked out for modification, you can modify the "File's comment" or Task field in the Check In/Information window. These changes are saved in the revision's `'ckid'` resource.

If you check out a read-only copy of a file revision and you then check out a modifiable copy of that same revision, Projector does not recopy the data of the revision; it simply updates its `'ckid'` resource to reflect the new checkout.

It is possible for the `'ckid'` resource to become corrupt or, in some cases, to be deleted from the checked-out file.

- Some programs can inadvertently delete Projector's `'ckid'` resources from files. When a program such as Microsoft Word saves a file, it deletes the file's `'ckid'` resource. To prevent this from happening, see Appendix A, "The `'ckid'` Resource Format," for information on the support that your application must provide for files that are checked into the Projector database.
- If you use the Save As command to save a file, the `'ckid'` resource remains with the old file and is not copied to the new file.

If the `'ckid'` resource of a revision is corrupted or removed, Projector cannot identify the revision, which becomes an orphan file, no longer belonging to any project.

If you still need to check the file in, you must move or rename your copy, cancel the checkout of the revision that is damaged, check out the revision again, and use the `TransferCkid` command to move the Projector information from the checked-out revision to your orphan file.

You can also use the `TransferCkid` command if you want to check in a branch as the latest revision to the main trunk of a file. The section "Deleting Revisions" on page 16-41 explains this procedure.

Note
The structure of the `'ckid'` resource is subject to change by Apple. ◆

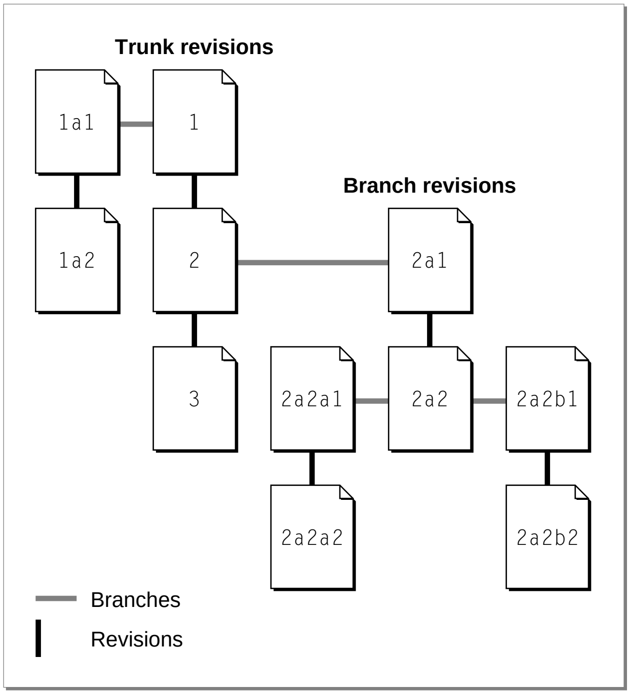
Branching

During the development process, you sometimes want to pursue parallel development while work on the main trunk proceeds. The revisions belonging to the parallel development are said to be a branch of the main trunk.

Branching is recursive. You can create a branch that diverges from an existing branch and you can do this to any desired depth. You can also create multiple branches from the same revision. Branch revisions use a numbering scheme that allows you to visualize the tree by knowing only the revision numbers. This scheme is described in the next section, "Revision Numbers."

Branching from the last revision is simple. For example, if the current revision of `csource1.c` is 3, then all you need to do is to click the Branch checkbox before checking out the file as modifiable. While the file is checked out, it is labeled Revision 3a+. After you check it back in, it is named revision 3a1. A second parallel branch from the main trunk would be labeled 3b1 after you check it back in. If you check out 3b1 for modification, edit it, and then check it back in, it becomes 3b2. Similarly, a branch from 3a2 would become, after check-in, 3a2a1. Figure 16-16 shows how this numbering scheme is used to designate revisions, branches, and branch revisions to any file checked into Projector.

Figure 16-16 Revisions, branches, and branch revisions



Revision Numbers

To specify a particular revision in a command, append a comma followed by the desired revision number to the end of the filename. In other words, `file.c,3` refers to revision 3 of `file.c`. (Because commas are used to indicate revision numbers, they are not allowed in project filenames.)

If you do not specify a revision number, Projector checks out the latest (nonbranch) revision of the specified file. For example, this command checks out the latest (current) revision of `file.c`.

```
CheckOut file.c
```

Regardless of what the current revision is, the following command checks out revision 3 of `file.c`:

```
CheckOut file.c,3
```

This command checks out the latest revision on branch 4a:

```
CheckOut file.c,4a
```

By default, revision numbers are assigned in numeric order by whole numbers, that is, 1, 2, 3, . . . 100, and so on. However, you can use double numeration instead, that is, 1.1, 1.2, 1.3, . . . 1.99, 1.100, 1.101, . . . 2.1, 2.2, and so on. The syntax for double numeration is

major [. *minor*...]

When you check in a new revision, Projector automatically increases its revision number by 1; for example, from 4 to 5, or 4.9.2 to 4.9.3. You can override this action when you check in the file by doing the following:

- Click the Revision button in the Check In window and use the dialog box that is displayed to specify a revision number for the file you check in.
- Specify the number of the revision when you use the `CheckIn` command to check in the file. For example, the following command checks in `file.c`, forcing the revision to 4.1:

  ```
  CheckIn file.c,4.1
  ```

  This command is legal only if the revision that was checked out was less than 4.1—for example, 4, 3.9, 4.0.9, or 2, and so on.

The only restriction to the number you specify is that it must be sequentially greater than the revision that was checked out.

Comparing Revisions and Merging Branches

The MPW Shell's Project menu includes the two menu items Compare Active and Merge Active.

- The Compare Active menu item calls the script `CompareRevisions` and allows you to identify differences between revisions.

- The Merge Active menu item calls the script `MergeBranch` and allows you to find the differences and selectively to copy and paste material from a branch to the main trunk. This is the method used when work on a branch proves fruitful and you want to incorporate that work into the main line of the project.

Choosing Compare Active from the Project menu allows you to compare differences between two revisions of a file. To use this menu item, do the following:

1. Check out the revision you're interested in.
2. Open the file. It should now be your active window.
3. Choose Compare Active from the Project menu.

Projector displays a selector window naming the revision number of the active window and listing all other revisions. Select the revision you want to compare the active window to. The `CompareFiles` script compares the two files and displays the differences.

Choosing Merge Active from the Project menu allows you to selectively copy and paste differences between the latest checked-out (modifiable) revision of a file and one of its branches. To use this menu item, do the following:

1. Check out a modifiable copy of the latest main trunk revision.
2. Check out the branch revision you're interested in and open the file. It should now be the active window.
3. Choose Merge Active from the Project menu.

The `MergeBranch` script compares the two files. It then highlights (selects) the differences so that you can cut and paste selections from one file to the other.

Deleting Revisions

Deleting any revision except the latest file revision is easy. You use the `DeleteRevisions` command to delete old revisions of a file by specifying the oldest revision you want to keep. For example, the command

```
DeleteRevisions SecretProject.p,25
```

deletes all revisions of `SecretProject.p` prior to, but not including, revision 25.

The command

```
DeleteRevisions SecretProject.p
```

deletes all revisions of `SecretProject.p` except the latest.

You can delete entire branches by naming the branch (for instance, *filename*.c`,22a`). Otherwise, the `DeleteRevisions` command affects only revisions on the main trunk.

You cannot delete revision *n* of a file while leaving revisions 1 through *n*–1 intact. This restriction prevents confusion of the revision numbering scheme.

You can remove a file and all its revisions by using the `-file` option of the `DeleteRevisions` command.

You can delete the latest revision of a file by taking the last known good revision and checking it in as the latest revision. This procedure is illustrated by the following example.

Suppose you check out revision 10 of a file for modification, change it, and check it back in. You then decide that this latest revision, revision 11, is useless, and you want to revert to revision 10. To do this, you must make revision 10 look to Projector like the latest revision of the file. The following steps explain how you do this.

1. Check out revision 11 for modification.
2. Check out revision 10 to a different directory.
3. Use the `TransferCkid` command to transfer the `'ckid'` resource from revision 11 to revision 10. For example:

   `TransferCkid MyFile` ***DifferentDirectory***`:MyFile`

Once the resource is copied to the revision you want to keep, that revision contains all the information formerly associated with the revision you're trying to delete: its filename (that's why you put it in a different directory), its revision number, and other project information. Revision 10 is now revision 11+. When you check it back in, it will become revision 12.

Note that the old revision 11 (the one you wanted to delete) becomes an orphan file after you transfer its `'ckid'` resource. This means that it is no longer recognized as belonging to a project and cannot be checked in again. You can keep it or delete it; in its orphaned state, it cannot confuse Projector.

▲ **WARNING**
Once you delete revisions, there is no way to recover them. ▲

Naming a Set of Revisions

Projector allows you to associate a name with a chosen set of file revisions. You can, for example, assign all revisions corresponding to a given release the name Alpha1. You can then use the Select Files in Name pop-up menu in the Check Out window to select from a list of defined revision names. All files associated with the name you choose are then selected, giving you a fast way to check out the source files for that release.

To assign a name to a set of revisions, you use the `NameRevisions` command. The basic syntax of the `NameRevisions` command is

`NameRevisions` ***Name FileRevisionName*** [***FileRevisionName***]...

The following command associates all the files in the Base project (but not its subprojects) with the name FirstRelease:

```
NameRevisions -project Base -a FirstRelease -r
```

The `-a` option specifies all files in the current project. The `-r` option executes the command recursively on the current project and all its subprojects.

The revision name you specify must begin with a character and cannot contain commas, dashes, or the symbols >, <, ≥, and ≤. Revision names are not case sensitive; they are maintained on a per-project basis and can refer, at most, to one revision per revision tree in that project.

If you make a mistake or want to cancel the assignment, you use the `DeleteNames` command. For example,

```
DeleteNames FirstRelease
```

Revision names can be public or private, static or dynamic. The following two sections explain these distinctions.

Public and Private Names

A revision name can be either public or private.

- If a name is public, which is the default, the name is recorded in the Projector database and is displayed in the Check Out windows of all users.
- If a name is private, it exists only for the convenience of the user who defines it and retains its assigned value only for the duration of the current MPW session. Private names are displayed first in the Select Files in Name pop-up menu in the Check Out window and are separated from public names by a dotted line.

  To make a name private, specify the `-private` option with the `NameRevisions` command used to create the name.

  To make a private name permanent, you must include the `NameRevisions` command that creates it in one of your `UserStartup` scripts.

Static and Dynamic Names

You can also specify a name defined with the `NameRevisions` command as being static (the default) or dynamic.

- If a name is static, it is expanded to the revision level of the files associated with the name when the name is defined.
- If a name is dynamic, it is expanded to the revision level of the files associated with the name when the name is used.

To make a name dynamic, you specify the `-dynamic` option with the `NameRevisions` command used to create the name.

If you define the name Alpha1, as follows:

```
NameRevisions -a Alpha1
```

all versions of the files in the current project that exist at the time you use the command are associated with the name Alpha1. If at some future time when the files have been revised several times, you select the versions associated with Alpha1, you will obtain the versions that existed when you defined the name.

If you define the name Latest, as follows:

```
NameRevisions -a -dynamic Latest
```

all the latest versions of the files in the current project that exist at the time you use the version name are associated with the name Latest.

It is also possible to define a version name that is associated with static versions of some files and dynamic versions of others. For example, the following command

```
NameRevisions -dynamic Work file.c,4 file.h,3 library.c
```

specifies a specific revision for `file.c` and `file.h`, but the latest revision to `library.c`. The name Work might be equivalent to

```
file.c,4 file.h,3 library.c,5
```

at one time, and equivalent to the following names at a later point in time:

```
file.c,4 file.h,3 library.c,21
```

To list the files currently associated with a revision name, use the `NameRevisions` command with the revision name as a parameter. For example,

```
NameRevisions -s Alpha
```

The `-s` option displays one name per line. The names are not listed in alphabetical order.

Redefining a Revision Name

You can use the `NameRevisions` command to append names to an existing list or to replace names in an existing list.

The following command appends the file new`Resources.h` to the existing set of names associated with Beta:

```
NameRevisions Beta newResources.h -project Prometheus
```

The following command replaces the filenames associated with Beta with the files `newResources.h`, `newInterfaces.h`, and `source.c` in the current project.

```
NameRevisions Beta -replace newResources.h ∂
                    newInterfaces.h source.c
```

In some cases, files are replaced even if you don't use the `-replace` option—for example, if you use this command to define the revision name Alpha:

```
NameRevisions Alpha A,1 B,2 C,3
```

The following command replaces file `A,1` with file `A,4`:

```
NameRevisions Alpha A,4
```

The following command replaces file `A,1` with the latest revision of file `A`:

```
NameRevisions Alpha A
```

To delete a revision name, you use the `DeleteNames` command.

Retrieving Information About Files and Revisions

All information regarding a project, including all revision trees, revisions, comments, and so on, is kept in a single file called `ProjectorDB` of type `'MPSP'`.

There are two ways to get information out of Projector:

- Use the `ProjectInfo` command. This is well-suited to batch-type processes; for example, if you want a list of all revisions, including comments made by a particular user to a particular file.
- Click the question mark button in the Check Out window. This brings up the Check Out/Information window, which allows you to select information for the latest revision or for an entire revision tree (file). In the Check Out/Information window, you can also click the "View by" button, which brings up a dialog box that allows you to specify additional selection criteria.

Project, File, and Revision Information

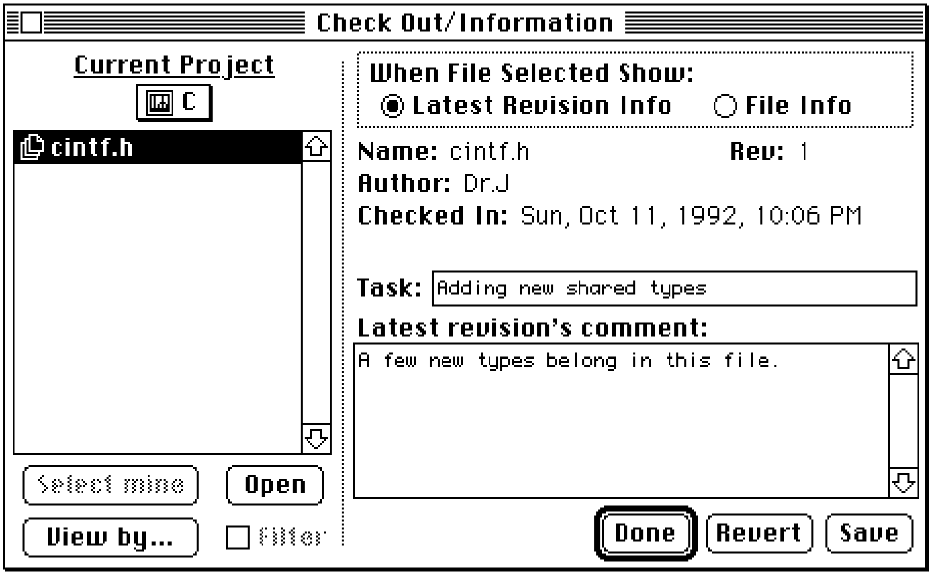
The information that you can retrieve from a project includes

- project information
  - author
  - last modification date of the project
  - project comment

- revision tree (file) information
  - author
  - date that the original file was added to the project
  - last modification date of the revision tree
  - revision tree comment
- revision information
  - author
  - task
  - date that the revision was created
  - revision comment

Whether Projector displays information about a project, file, or revision is determined by the items selected in the Check Out/Information window. Figure 16-17 shows the Check Out/Information window.

Figure 16-17 The Check Out/Information window



The information displayed is determined as follows:

- If you select a project in the project list, information about that project is displayed. The specific information displayed depends on which button is selected, Latest Revision Info or File Info.
- If you select a file from the project list and you click the Latest Revision Info button, Projector displays information about the status of the latest revision.
- If you select a file from the project list and you click the File Info button, Projector displays information about that revision tree.

Double-clicking a file in the project list displays its revision tree. The latest revision is selected by default, and its status information is displayed. Selecting another revision displays its status.

The “Latest revision’s comment” and Task text that has already been entered is editable so that you can make changes or additions to old comments.

Obtaining the History of a Revision

MPW 3.3 introduced a new facility that allows you to obtain the history of a Projector file without mounting the project. This is accomplished by storing in the `'ckid'` resource of a file all of the information that is normally displayed by executing the `ProjectInfo` command: author, check-in date, task, and comment.

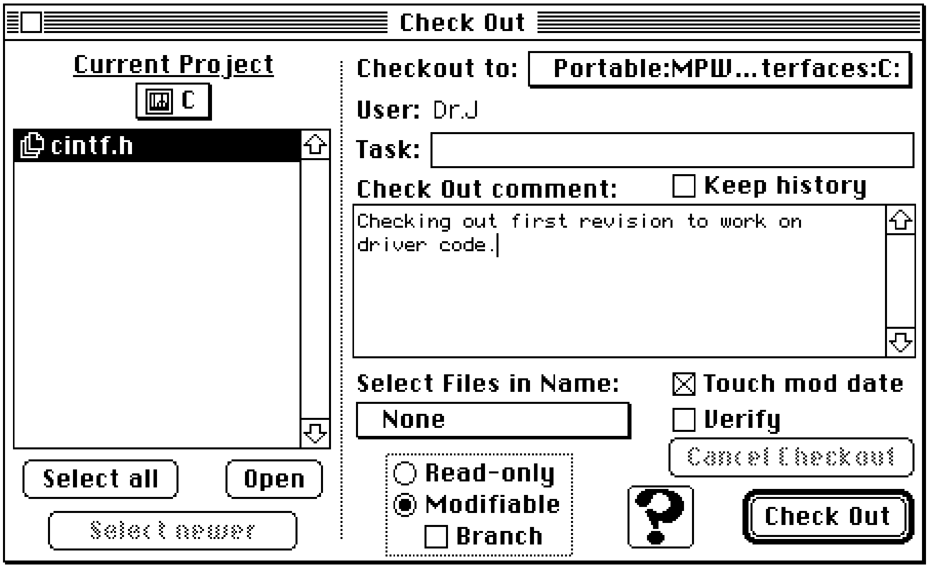
You can have a file’s history stored in the `'ckid'` resource in one of two ways:

- Use the `-history` option to the `CheckOut` command; for example,

  `CheckOut -history` ***filename***`[,`***n***`]`

  where *filename* specifies the name of the file and *n*, the revision number.
- Click the “Keep history” checkbox in the Check Out window before you check out the file. Figure 16-18 shows the location of the “Keep history” checkbox in the Check Out window.

Figure 16-18 The “Keep history” checkbox



The history you retrieve using either of the preceding methods is for those revisions that are the specified revision’s ancestors. If you check out the latest revision of a file that has no branches, you obtain a complete history for the file. If you check out a revision that is on a branch, the history is for the revisions that are on a direct path from that revision back to the root (the initial revision).

You can use one of two methods to display the history of the file:

- Specify the `-comments` option to the `ProjectInfo` command. For example,

  ```
  ProjectInfo -comments MySources.c
  ```

  displays the checkout data for the revision, followed by a full check-in history for that revision and its ancestors.

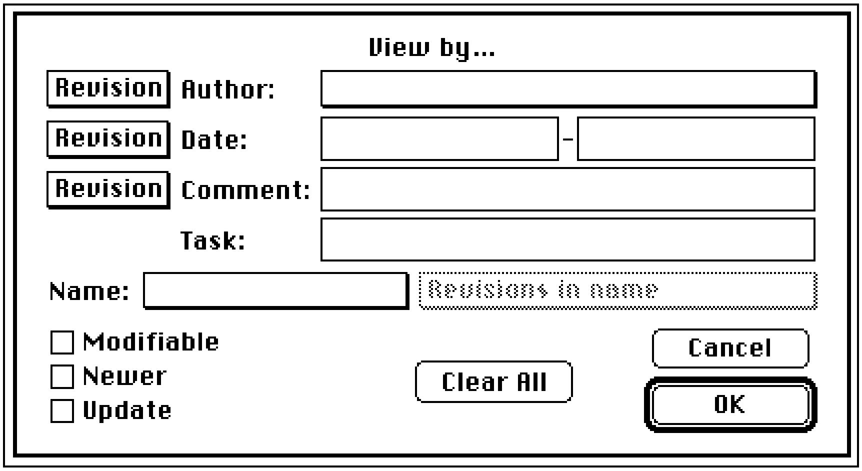
  If you use the preceding method to display a file’s history and the project is mounted, the format used to display the information depends on whether you specify a leaf name (`MySources.c`) for the file or a full or partial pathname (`:MPW:MySources.c`). The information displayed, however, is the same.

■ Click the question mark button in the Check In window and select the file. This displays the entire history in the comment box whether or not the project is mounted.

Using the "View by" Dialog Box

The "View by" dialog box, shown in Figure 16-19, provides different criteria that you can use to filter the revisions in the list. Only revision trees or revisions that match your criteria are displayed.

Figure 16-19 The "View by" dialog box



To display the "View by" dialog box, open the Check Out/Information window and click the "View by" button. Table 16-2 describes the effects of selecting different items in the "View by" dialog box.

Table 16-2 Projector information selection criteria

Item	Effect
Author	The author of a revision tree or revision. All the authors known to the project are listed in a pop-up menu. Select the desired author from the list.
Date	The file modification date or revision creation date. Type in the starting and ending dates. The format is *dd*/*mm*/*yy* [*hh*:*mm*[:*ss*] [AM|PM]]. To specify "on or since a date," enter the starting date in the first box and leave the second box empty. To specify "before or on a date," enter the ending date in the second box and leave the first box empty.
Comment	File or revision comments. Type in either a literal string or a regular expression in slashes (/*regular-expression*/).
Task	Task comments. Type in either a literal string or a regular expression in slashes.
Name	All your private names followed by the project's public names are displayed in a pop-up menu. Select the desired name from the list. Once you select a name, you can also specify a relation to that name (for example, to list all the revisions since alpha). Select the desired relation from the pop-up menu next to the name.
Modifiable	List only those revisions checked out for modification.
Newer/Update	List only those revisions that would be checked out by using the corresponding option to the `CheckOut` command.

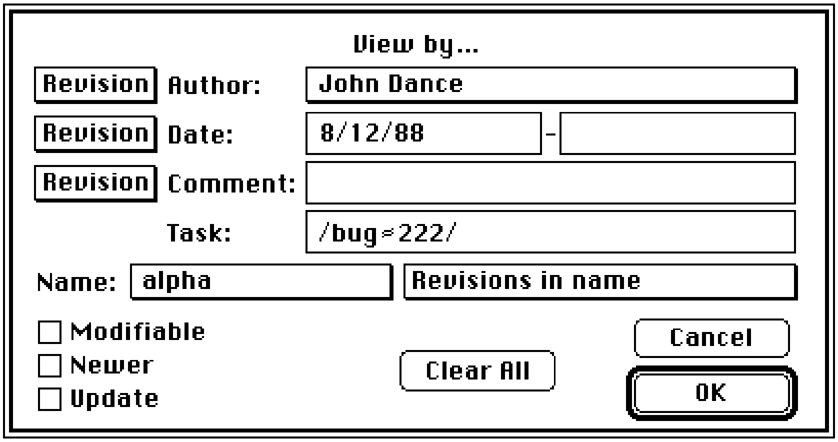
For the author, date, and comment items, you must specify whether each should be applied to revision trees or to revisions. You do this by choosing Revision or File from the pop-up menu to the left of Author, Date, or Comment.

The display of all the revision trees is affected unless you specify a "file" filter from the Revision/File pop-up menu. For example, in Figure 16-20, the user has specified a filter to list all revisions in alpha, created by John Dance, on or after August 12, 1988, dealing with Bug #222.

The following `ProjectInfo` command is equivalent to the "View by" dialog box shown in Figure 16-20.

```
ProjectInfo -a 'John Dance' -d '≥8/12/88' -t '/bug≈222/' -n alpha
```

Figure 16-20 The “View by” dialog box with selection criteria



Working With Projector—A Quick Reference

The tutorial included in this chapter allows you to become familiar with Projector, mainly through MPW’s Project menu. This section presents additional reference information; it covers

- automating the mounting of projects
- the meaning of Projector icons
- manipulating projects and files
- Projector commands

Rules for Using Projector

Observe the following rules when using Projector:

- Assign unique filenames to all files in a project.
- Do not delete revisions out of sequence.
- Do not use commas in filenames.

- Do not hyphenate symbolic names created with the `NameRevisions` command.
- Do not create a symbolic name that is the same as the name of a project file.

Using a Script to Mount a Project

Before you can check files out of or into a project, you need to mount the project and specify a checkout directory. If the project resides on a server, you also need to mount the server. The sample script shown in Listing 16-1 contains the commands you need to do this. You can modify the sample script to suit your situation and call it from MPW whenever you want to work with Projector.

The `Choose` command mounts the server containing the project. You can use the `-askpw` or `-askpv` option of the `Choose` command to be prompted for a secure server password or for a secure volume password, respectively.

Listing 16-1 A sample script to set up Projector

```
#Define variables used in this script.
Set MyProjectServer DevTools:Zeus:Thunder
Set MyProject Thunder:HotCoals
Set MyCheckOutDirPath MyVolume:CheckOut:
Set MyCheckOutDirName NewCompiler

# Mount the server containing the project and prompt for server password.
Choose {MyProjectServer} -askpw

# Mount the project.
MountProject {MyProject}

# Display the names of the mounted project and subprojects. This command is not
# required; it's just used to check that the specified projects have been mounted.
MountProject -r

# Make the checkout directory the current directory.
Directory {MyCheckOutDirPath}

# Establish all checkout directories recursively for the project and subprojects.
CheckOutDir -r {CheckOutDirName}
```

```
# Display the directories set up by the previous command. This command is not
# required; it’s just a check that the specified directories have been set up.
CheckOutDir -r
```

Projector Icons

As you browse through the project hierarchy in Projector windows, look for the following visual cues that convey revision ownership.

Icons Displayed in the Check In Window

Read-only icon. The file is a read-only file belonging to the current project.

Modified read-only icon. The file is a modified read-only file belonging to the current project.

Regular document icon. The file does not belong to any project. It is visible only when “Show all files” is selected.

Pencil icon. The file is checked out from the current project for modification by the current user.

Lock icon. The file is checked out from the current project for modification by another user.

Question mark icon. The file belongs to a project other than the current project. It appears in the Check In window only when “Show all files” is selected.

Question mark with plus icon. The file is modifiable and belongs to another project. This icon is displayed in the Check In window only when “Show all files” is selected.

Corrupt `'ckid'` resource icon. This icon is displayed in the Check In window only when “Show all files” is selected.

Obsolete file icon. This icon designates a file that has been made obsolete using the `ObsoleteProjectorFile` command.

Icons Displayed in the Check Out Window

 Project icon. The file is a Projector database (`ProjectorDB`) file.

 Projector revision-tree icon. The file has been checked into Projector.

 Regular document (single revision) icon. The specified revision is currently available. This icon is visible when an individual revision tree is displayed.

 Pencil icon. When a project is displayed so that all its revision trees are listed, the pencil icon means that the latest revision of the main trunk is checked out for modification by the current user. When an individual revision tree within a project is displayed (a list of revisions), the pencil icon means that the particular revision is checked out for modification by the current user.

 Lock icon. When a project is displayed so that all its revision trees are listed, the lock icon means that the latest revision of the main trunk is checked out for modification by another user. When an individual revision tree within a project is displayed (a list of revisions), the lock icon means that the particular revision is checked out for revision by another user.

Manipulating Projects and Files

Anyone who has write access to a project (through an AppleShare file server) can

- move, rename, and delete projects
- delete revisions that are no longer needed
- rename a file in a project
- delete files that do not belong in the project
- modify a read-only file
- obsolete files that are no longer used
- reverse the process that makes a file obsolete when the file is needed again

The following sections explain the commands and procedures used to do these things.

Moving, Renaming, and Deleting Projects

You can move or rename a project by using the Finder or the regular MPW commands. Renaming the project directory renames the project.

No other Finder or MPW operations are allowed on project directories.

There are two points to keep in mind when moving or renaming a project:

- When you move or rename a project, the project hierarchy changes. The `MountProject` commands and scripts used to set up Projector must be modified to reflect the new location of the project.
- You should move or rename projects only when no revisions are checked out for modification. After the project has been changed, *all* read-only copies should be checked out again. This is recommended because Projector puts the project name in the resource forks of revisions during checkout. Once the project is moved or renamed, the information is no longer valid.

You can delete an entire project by deleting the folder containing the project. Use the Finder or the MPW Shell's `Delete` command. Be aware that once you delete the project, all files and their revisions are lost.

Modifying a Read-Only File

If you check out a read-only copy of the file, want to modify it, and do not have access to the Projector database, you can use the `ModifyReadOnly` command to remove the read-only restriction from the file.

After modifying the file, you can check it back in as a new revision if the revision you are checking back in is, in fact, the latest revision. A conflict can arise if, in the meanwhile, another user has checked out a modifiable copy of the file. When you check the file back in, Projector displays a message letting you know if such a conflict exists. There are two possibilities:

- Another user has checked out a modifiable copy of the file and checked it back in. In this case, you should use the `CompareRevisions` command to make sure that you are not overriding changes that the other user has made to the file. If changes have been made, they have to be merged. To check the file back in as the latest revision, you have to check out the latest revision, transfer the `'ckid'` resource of the file to your revision, and then check your revision back in as the latest revision. The procedure is explained in "Deleting Revisions" on page 16-41.

- Another user has checked out a modifiable copy of the file but has not checked it back in. In this case, Projector does not allow you to check your copy back in. In an emergency, you can cancel the other user's checkout, or if time permits, you can contact the other user and work out a mutually satisfactory solution.

Making a File Obsolete

MPW 3.3 introduced a new command, `ObsoleteProjectorFile`, that causes a file within a project to become inactive. This means the following:

- You cannot check out the file for modification or create additional revisions of the file.
- If you use the `-a` (all) or `-newer` option of the `CheckOut` command, this file is not included in the checkout.
- You can check out existing revisions of the file as read-only by using the `-obsolete` option to the `CheckOut` command.
- The `ProjectInfo` command displays information for the file with the flag `Obsolete` on the lines bearing such information.
- If the obsolete file is part of a set of files named with the `NameRevisions` command, it is still included in a selection by name, provided that the checkout is read-only. If the checkout is not read-only, the file is not checked out as part of the named set, and a warning is displayed that an attempt has been made to check out a modifiable obsolete file.

The syntax for the `ObsoleteProjectorFile` command is the following:

`ObsoleteProjectorFile [-p] [-u` ***user***`][-project` ***project***`]` ***filename...***

Option/parameter	**Description**
`-p`	Writes progress information to standard output.
`-u` *user*	Allows you to specify a user name other than that of the current user.
`-project` *project*	Names the project that contains the files.
filename	Specifies the name of the file to be made obsolete.

The following examples illustrate the effect of various commands on the file `MySource.c`, which has been made obsolete.

The following command is invalid because the `-obsolete` option is missing:

```
CheckOut MySource.c
```

In the following example, the last revision of the file is checked out read-only:

```
CheckOut -obsolete MySource.c
```

In the following example, revision 3 of the file is checked out if it exists:

```
CheckOut -obsolete MySource.c,3
```

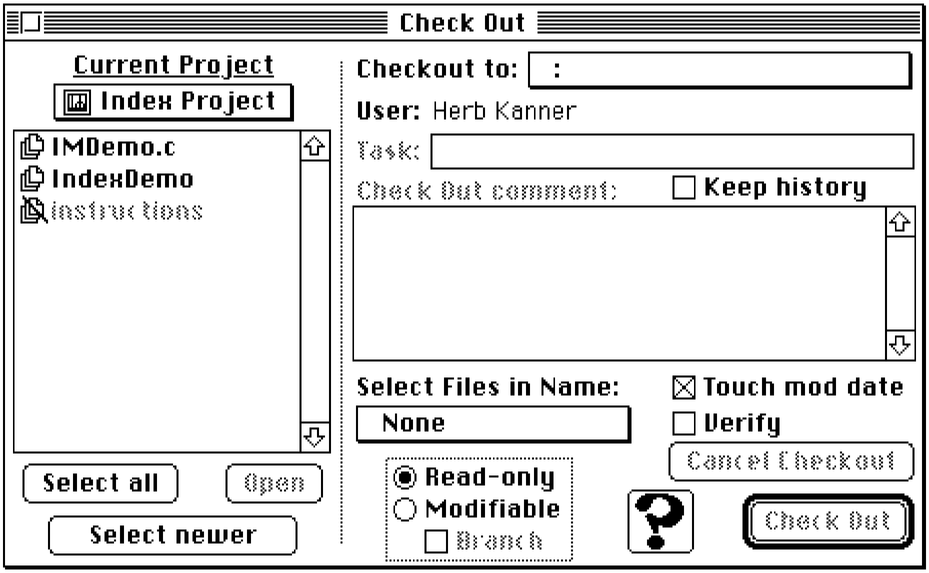
The following command checks out `MySource.c` if it was part of the set Beta1.

```
CheckOut -obsolete Beta1
```

Using the Check Out Window With Obsolete Files

Obsolete files are indicated in the Check Out window by an icon with a line through it and by the filename being dimmed, as shown in Figure 16-21.

Figure 16-21 Obsolete files in the Check Out window



You select an obsolete file by pressing the Option key when you click the filename.

Handling obsolete files using the Check Out window or the `CheckOut` command is subject to the following restrictions:

- You cannot check out a modifiable copy of an obsolete file.
- Selecting or specifying a revision name selects obsolete files if they are part of the named set.
- Clicking the “Select all” or “Select newer” button does not select obsolete files.

▲ **WARNING**
All users of a Projector database should use the same version of the MPW Shell when using this command. If some users are using a version of the MPW Shell that does not recognize obsolete files, they will be able to modify and check in a new revision of such a file. To the user of an MPW Shell that does recognize obsolete files, the file is still obsolete, but a new revision will have mysteriously appeared. ▲

Recovering an Obsolete File

You can reverse the process that makes a file obsolete by using the command `UnobsoleteProjectorFile`. The command syntax is

`UnobsoleteProjectorFile [-p] [-u` ***user***`] [-project` ***project***`]`
filename…

The options and parameters are the same as for the `ObsoleteProjectorFile` command described on page 16-57.

Renaming a File

MPW 3.3 introduced the `RenameProjectorFile` command, which allows you to rename a Projector file. The syntax of the command is the following:

`RenameProjectorFile [-p] [-u` ***user***`] [-project` ***project***`] ∂`
oldName newName

Option/parameter	Description
`-p`	Writes progress information to standard output.
`-u` *user*	Allows you to specify a user name other than that of the current user.
`-project` *project*	Specifies the name of the project containing the file.
oldName	Specifies the file’s old name.
newName	Specifies the file’s new name.

The `RenameProjectorFile` command has the following effects:

- The filename is changed in the Projector database.
- The file's old name is deleted; you can use it to name a new file or to rename another existing Projector file.
- When you use the `NameRevisions` command to obtain the names of all files belonging to a named revision, the new name is included in the list of files.

▲ **WARNING**
If you use the `RenameProjectorFile` command to change the name of a projector file, make sure that you update scripts and makefiles in which the old name is used. ▲

Projector Commands

Table 16-3 provides a brief description of the default action of Projector commands and their main options. The commands are listed in alphabetical order. For a complete description of these commands and their syntax, see the *MPW Command Reference*.

Table 16-3 Projector commands

Command	Description
`CheckIn`	Checks a file back into the Projector database, saves any changes to the file as new revisions, and (by default) leaves you with a read-only copy of the file. Options to the command allow you to check in all files in the current directory that have been checked out for modification, or to add a new file to the project. Options also allow you to specify a task and a comment, to check a file in as a branch, to alter the modification date of the file, to keep a modifiable copy, or to delete the file from your checkout directory after checking it in. The `-verify` option verifies the check-in to protect against loss of data.

continued

Table 16-3 Projector commands (continued)

Command	Description
	You can use the Check In menu item in the MPW Project menu in place of this command.
`CheckOut`	Checks out a read-only copy of a file revision from a project and places it in the checkout directory associated with the project. You can check out a file revision or a set of file revisions. Options to the command allow you to specify a task and comment, to place the file in a directory other than that associated with the project, to check out a modifiable copy of the revision, to check out the file as a branch, to cancel the checkout of the file, and not to change the modification date of the file. A safety feature prevents your using the `-cancel` option to cancel the checkout of a file that does not belong to you. The `-history` option allows you to view the file's history without mounting the project to which the file belongs. The `-verify` option asks Projector to warn you if a read-write error prevents a successful checkout. The `-obsolete` option allows you to check out a read-only copy of an obsolete file. You can use the Check Out menu item in the Project menu in place of this command.
`CheckOutDir`	Sets the directory into which Projector file revisions are checked out. If the directory does not exist, the `CheckOutDir` command creates it. Entering this command without arguments displays the name of the current checkout directory.

continued

Table 16-3 Projector commands (continued)

Command	Description
	Options to the command allow you to set the root directory only, to set the directory recursively for all subprojects, or to reset the checkout directory to its default value—that is, the current directory.
`CompareRevisions`	Compares revisions of a file. This command calls the `CompareFiles` script both to display revisions on the screen and to highlight their differences. A Compare menu is appended to the menu bar that you can use to step through the differences and to copy and paste a selection.
`DeleteNames`	Deletes one or more symbolic names (names set by using the `NameRevisions` command).
	Options to the command allow you to delete all private names, all public names, or all names for the current project and, if desired, all subprojects as well. Public names are deleted by default.
`DeleteRevisions`	Deletes all revisions of a file (or of a branch of a file) previous to the revision you specify. You must use a special procedure to delete the latest revision.
`MergeBranch`	Merges a branch revision of a Projector file into the revision trunk. If all the file's revisions are older than the branch, the branch is checked in as the latest trunk revision. Otherwise, `MergeBranch` checks out the latest revision on the trunk and calls `CompareFiles` to allow you to copy differences from the branch into the trunk revision. When you have finished merging the files, you can check the modified trunk revision back into the project.
`ModifyReadOnly`	Allows you to modify a file that has been checked out read-only.
`MountProject`	Mounts the specified project and all its subprojects.

continued

Table 16-3 Projector commands (continued)

Command	Description
	Options to the command allow you to list projects recursively, list projects by directory pathnames, write only the names of root projects, and inhibit quoting names containing spaces or special characters.
`NameRevisions`	Creates a symbolic name to represent a set of revisions in a single project or modifies a previously defined name. By default, the name created is public.
	Specifying the command without parameters displays a list of all symbolic names in the current project and their values.
	Options to the command allow you to make a name private, public, static, or dynamic and to replace an existing value for a symbolic name.
`NewProject`	Creates a directory for a new project. Options to the command allow you to specify a comment, to open and close the New Project window, and to name the current user. You can use the New Project menu item from the Project menu in place of this command.
`ObsoleteProjectorFile`	Causes a file within a project to become inactive. You can reverse the process with the `UnobsoleteProjectorFile` command.
`OrphanFiles`	Removes the `'ckid'` resource from a file, thus severing the connection between a file and a project.
`Project`	Makes the specified project the current project or, without parameters, displays the name of the current project.
	The `-q` option to the command does not quote the project name displayed.
	You can also use the Set Project menu item in the Project menu to make a project the current project.

continued

Table 16-3 Projector commands (continued)

Command	Description
`ProjectInfo`	Displays information about the current project. If you specify the command without options or parameters, it returns information on all revisions of every file in the project. If you specify a pathname, it will return information about a specific file. Options allow you to filter the information you receive by author, comment, date, revision level, and symbolic name. The `-m` option displays information about all files checked out for modification in a project. The `-log` option prints the log information for the specified project. This information includes a record of the creation and deletion of public names and the deletion of revisions. It also keeps track of all canceled checkouts for files in the current project. You can also obtain information about the current project, revision tree (file), or individual revision by using the “View by” dialog box available from the Check Out/Information window.
`RenameProjectorFile`	Renames a Projector file.
`TransferCkid`	Moves the `'ckid'` resource from a source file to a destination file.
`UnmountProject`	Unmounts the specified projects. The `-a` option to this command unmounts all mounted projects.
`UnobsoleteProjectorFile`	Causes a file that has been made obsolete by the `ObsoleteProjectorFile` command to be modifiable—that is, it becomes a normal Projector file once again.

Measuring Performance

Contents

MPW comes with several performance tools that allow you to monitor the performance of your program.

- MrPlus provides static and dynamic information for PowerPC-based programs. This information can include data about the program structure, the number of times an instruction is executed, and how often registers are used. MrPlus can also optimize your program by reordering portions of your code in the PEF file.
- PPCProff (for PowerPC programs) and Proff (for classic 68K programs) provide profiling and performance monitoring for programs compiled from C and C++. For every routine called during program execution, the Proff tools record the identity of the called routine, the identity of the calling routine, and the time spent in each routine.
- MPW Perf provides more detailed performance monitoring for 68K-based programs compiled from C and C++. This tool samples the PC register to determine the number of times a part of your code executes. Whereas the smallest unit you can measure with Proff is a procedure or function, MPW Perf allows you to focus on “hot spots” within a routine and to measure the number of times your program executes even a single instruction.

IMPORTANT

MrPlus and PPCProff are currently in prerelease form and may change before the final release. You should check the release notes for any changes. ▲

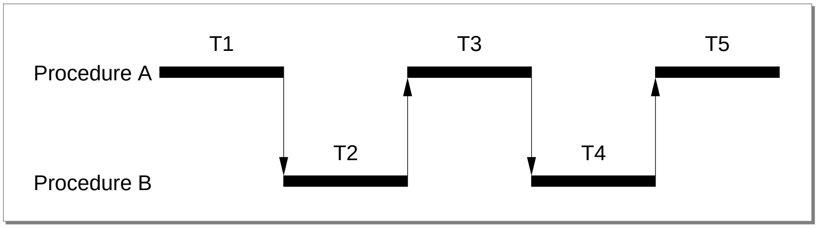
In addition, the Power Mac Debugger provides the Adaptive Sampling Profiler, which you can also use for performance measurements. For more information, see the *Macintosh Debugger Reference*.

Using Performance Measurement Tools

Before using any performance tools, you should be familiar with some of the terms used in performance measurement. **Profiling,** which occurs while your program executes, is the dynamic recording, for every routine call, of the identity of the called routine and the point from which it was called; for example, statement *n* in procedure `moo`. **Performance monitoring** in the context of profiling is the recording of the time spent in each such routine.

Now consider the calls made from procedure A to procedure B, shown in Figure 17-1.

Figure 17-1 Hierarchical time and flat time: performance measurement



An **arc** is the execution of a routine from a specific call site. In Figure 17-1, procedure A calls procedure B twice, but from two different call sites; therefore, T2 and T4 represent two distinct arcs even though they record the time it takes the same procedure, procedure B, to execute.

A **basic block** in any part of a program is the longest contiguous stretch of code that contains no branches and no points to which branches are taken (excluding the endpoints). Performance tools record the arcs between basic blocks.

The **flat time** is the amount of time spent executing the routine. With reference to Figure 17-1, if procedure A and procedure B are both monitored, the flat time for procedure A is equal to T1 + T3 + T5. The flat time for procedure A does not include segment loading (for 68K-based code) or profiling overhead for procedure A, but it does include some performance monitoring and routine call overhead for any routine called directly by procedure A. If procedure A is monitored but procedure B is not, the flat time for procedure A is equal to T1 + T2 + T3 + T4 + T5. That is, if the called routine was not built with the required profiling options or directives, the performance tool thinks that the called routine is part of its caller.

The **hierarchical time** is the amount of time spent in the called routine, plus the time spent in any routines called by it (directly or indirectly) before it returns to its caller. It does not include segment loading (for 68K-based code) or profiling overhead for the current routine. However, the hierarchical time does include this overhead for called routines. With reference to Figure 17-1, the hierarchical time for procedure A is equal to T1 + T2 + T3 + T4 + T5.

IMPORTANT

MrPlus and Perf do not measure actual time, but rather count the number of instructions executed. ▲

Although different performance tools use different approaches to monitoring your code, the procedure for them can be generalized as follows:

1. Follow the instructions in this chapter to turn on performance monitoring.
2. Run your program with performance monitoring enabled.
3. Generate a report that lists times spent executing your application's routines.

It is then up to you to interpret the performance report and to make the desired changes to your code.

For measuring the performance of PowerPC code, you can use either MrPlus or PPCProff, or possibly both. MrPlus measures performance by counting the number of instructions executed while PPCProff measures the time spent executing the instructions. This allows MrPlus finer resolution than PPCProff; you can identify problem areas down to the instruction level. However, this instruction-counting approach means that it cannot measure anything outside the fragments being monitored (for example, it cannot record time spent in a Mac OS call, which PPCProff can).

To measure performance of 68K code, you might start by using Proff to identify the most time-consuming routines and then use MPW Perf to identify the trouble spots within these routines more accurately. This procedure is especially useful if you are measuring the performance of very large programs.

▲ **WARNING**

The performance-monitoring tools are designed for temporary inclusion in your code to measure its performance. They are not designed for inclusion in commercial products because they rely on low-level system mechanisms that are not guaranteed to function correctly on all future machines. ▲

MrPlus (PowerPC Only)

MrPlus is an MPW tool that operates on PEF files. You can use MrPlus to do any of the following:

- Produce static information about an executable file, such as the size of the code and data sections, the frequency of appearance of various instructions, and the frequency of references to registers.
- Produce a modified version of the executable file that generates dynamic information about the program, such as the frequency of execution of various instructions.
- Optimize the executable file by reordering portions of code in the PEF file. These optimizations improve the utilization of the instruction cache and also reduce the number of page faults when virtual memory is enabled.

You activate MrPlus using the command

`MrPlus` ***PEFfile*** `[`***options...***`]`

where *PEFfile* is the file you wish to analyze. See the *MPW Command Reference* for a list of available options.

Static Analysis

MrPlus always performs a static analysis of the program you give it. You can add various options such as `-report imix` and `-report regs` to display static opcode usage and static register usage respectively.

Listing 17-1 show sample output generated from the following command:

```
MrPlus c_sample.ppc
```

Listing 17-1 MrPlus static analysis output

```
# Number of Sections = 3
# Code Section Size = 1024
# Data Section Init Size = 192
# Loader Descriptor: Type entry, Section 1, Offset 0000006C
# Code = 000001E0, TOC = 0000006C
# TOCMin = FFFFFF94, TOCMax = 0000012C
# Section Relocation Table: entries = 1
# sectno = 1, numrelocs = 3, offset = 00000000
# Number of Switches = 0, Unconditional Returns = 4, Conditional Returns = 0
# Number of Routines = 20, Basic Blocks = 48, Control Flow Arcs = 52
# Total Memory Used = 12587 Bytes
```

The information shown (line by line) is as follows:

- The number of sections (code, data, or loader) in the program.
- The size of the code section in bytes.
- The size (in bytes) of the portion of the data initialization section that is initialized to other than 0.
- The next two lines describe the main entry point of code fragment. The transfer vector is in section 1 (data) at offset 0x6C. The vector contains the code start address (0x200) and the initial TOC value (0x6C).
- The values of `TOCMin` and `TOCMax` indicate the range of offsets (in hexadecimal) referenced by the code in this fragment.
- The section relocation table indicates the relocations that are performed when the Code Fragment Manager loads a fragment. In this example, the table contains only one entry: section 1 (`sectno = 1`) requires three relocations (`numrelocs = 3`), and they begin at offset 0x00 in the relocation table.
- The number of switches implemented with jump tables, the number of normal return points (`Unconditional Returns`), and the number of conditional return points in the program.
- The number of routines, basic blocks, and control flow arcs in the program.
- The amount of memory used by MrPlus. You can use this value to help adjust the MPW Shell partition size.

Instrumentation Mode and Dynamic Analysis

If you specify the `-instrument` option, MrPlus generates a new executable file that contains instrumentation code in addition to the original code. This extra code records information for dynamic analysis.

You can select between two different instrumentation modes. Option `-instrument calls` specifies routine-level profiling, while `-instrument branches` specifies basic block-level profiling.

Executing MrPlus in instrumentation mode produces the following files:

- The instrumented program (default name *PEFfile*.`prof`).
- A file that maps the instrumentation counters to program counter (PC) values (default name *PEFfile*.`pmap`).
- If you had previously generated an XCOFF file for your program containing symbolic information, MrPlus generates a new symbolic file (default name *PEFfile*.`prof.xcoff`) that reflects the addition of the instrumented code.

You can run the instrumented program with whatever test data you desire. The program collects profile data as it executes and prompts you for a filename to save the information before it exits (the usual convention is *PEFfile*.`pcnt`).

After generating the profile data, you can run MrPlus again using the `-arrange` option, the `-opt dynamic` option, or both options. You must use other options to specify the other input files. For example, use `-cntin` *PEFfile*.`pcnt` to indicate the profiler data file.

MrPlus optimizes the code according to the options as follows:

- The `-arrange` option rearranges the program in memory for improved instruction cache efficiency. The `routines` argument specifies that only entire routines are rearranged relative to one another. The `blocks` argument is identical to `routines`, but with the addition that unexecuted blocks of routines are placed at the end of the code section.
- The `-opt` option removes unneeded NOPs that follow some call instructions, provides sharing of nontrivial epilogue code for multiple return routines, and replaces load/store multiple instructions with multiple simple load/stores. In addition, `-opt` specifies direct branching to dominant (that is, greater than 50% likelihood) switch targets if they exist, and sets branch prediction hint bits.

The optimized executable file is called *PEFfile*.`opt` unless you specify otherwise with the `-fragout` option.

MrPlus also displays dynamic profiling data. This data includes

- a dynamic instruction mix report (only if you select the `-report mix` option)
- a flat profile report (based on routine level or basic block level, depending on the option used)
- a conditional branch prediction report

The dynamic output generated by MrPlus includes all the static information as well as several new items. Listing 17-2 shows the output of the command

```
MrPlus c_sample.ppc -cntin c_sample.ppc.pcnt -mapin c_sample.ppc.pmap ∂
-xcin c_sample.ppc.xcoff -arrange routines -opt dynamic
```

Note that this example includes symbolic information in the file `c_sample.ppc.xcoff`.

Listing 17-2 Dynamic analysis output for MrPlus

```
# Number of Sections = 3
# Code Section Size = 1056
# Data Section Init Size = 192
# Loader Descriptor: Type entry, Section 1, Offset 0000006C
# Code = 00000200, TOC = 0000006C
# TOCMin = FFFFFF94, TOCMax = 0000012C
# Section Relocation Table: entries = 1
# sectno = 1, numrelocs = 3, offset = 00000000
# Number of Switches = 0, Unconditional Returns = 4, Conditional Returns = 0
# Number of Routines = 20, Basic Blocks = 48, Control Flow Arcs = 52
# Found Line Number Table - 22 Entries                                 See Note 1
# Routine Level Flat Profile (1% cutoff)                               See Note 2
# 50.0% 50.0% 000003F0 .Button
# 99.9% 50.0% 00000408 .SystemTask
# Conditional Branch Prediction Report                                 See Note 3
#   Correct     Taken = 000000000 (NAN(000)%)
#   Correct Not Taken = 000000000 (NAN(000)%)
# Incorrect     Taken = 000000000 (NAN(000)%)
# Incorrect Not Taken = 000000000 (NAN(000)%)
# Branches Hinted   Correctly = 000000005
# Branches Hinted Incorrectly = 000000000
```

```
# Epilog Sharing Saved 0 Instructions                                See Note 4
# Load/Store Multiple Word Replacement Added 2 Instructions          See Note 5
# Total Memory Used = 45354 Bytes
```

The following notes describe the new entries:

1. The `Found Line Number Table` contains information that maps source code line numbers with corresponding PC values in the executable code. This line appears only if you compiled some of the modules with the `-sym on` option and included an XCOFF symbolic information file.
2. The `Routine Level Flat Profile` indicates the percentage of instruction counts spent in each routine. Each routine entry (from left to right) indicates
   - the cumulative percentage spent in this routine and others above it
   - the percent of total instruction counts spent in the routine
   - the code address where the routine begins
   - the name of the routine

   In this example, the program spends 50% of its instruction counts in the routine `.SystemTask`, which is located at 0x408. The program spends 99.9% of its instruction counts in the routine `.SystemTask` and all others above it.
3. The `Conditional Branch Prediction Report` indicates whether branches are mostly taken or not taken. It can also indicate whether the hardware is predicting branches correctly or if the software is hinting branches correctly.
4. The `Epilog Sharing` line indicates how many instructions were saved by sharing epilogue code.
5. The `Load/Store Multiple Word Replacement` line indicates how many instructions were added due to breaking up multiple load/stores into simpler ones.

You can display additional information using the `-reports` option as follows:

- The `-reports imix` option displays the static and dynamic instruction usage. The static usage indicates how often an instruction appears in the program, while the dynamic report indicates how often an instruction is executed.
- The `-reports regs` option displays how often each register is read from, or written to, in the program.
- The `-reports glue` option displays the number of import glue routines required in the program and which sites access them.

If you call MrPlus with the `-monitor icache` option, the output PEF file contains added instructions that cause the hardware counters to gather additional information during execution. On MPC601 and 603 microprocessors, the extra information is the execution time as reported by the hardware time base register. When running MPC604 microprocessors, the code also reports on the number of instructions completed and the number of instruction cache misses.

When using the `-monitor icache` option, the naming conventions for the output files are ***PEFfile***`.mon` for the instrumented program, and ***PEFfile***`.mon.xcoff` for the corresponding XCOFF file.

Using MrProf

To enhance the usefulness of MrPlus, you can use the application MrProf to generate tabular and graphic representations of data taken in multiple instrumentation runs. You can obtain information such as the number of times a particular arc was traversed, the addresses of the basic blocks being traversed, as well as the source code line numbers and the routine names called. You can also generate a call graph that shows the interrelationships of calling routines.

For more information, see the MrProf documentation in the Release Notes.

PPCProff (PowerPC Only)

PPCProff provides profiling and preformance monitoring for PowerPC programs compiled from C and C++. For each arc, PPCProff records the following information:

- the number of times it was executed
- the cumulative flat time for the arc
- the cumulative hierarchical time for the arc

It also records the total time used by the profiled application.

PPCProff is an MPW tool, but to prepare your program for its use, you must add the `-profile` option and the two PPCProff libraries to your link list (you must be running PPCLink version 1.4 or later):

```
PPCLink ∂
...
-profile on PPCProffLib.o PPCProffLib ∂
...
```

When `-profile on` is specified, PPCLink adds special monitoring routines to the output file and also generates an XCOFF file that contains symbolic information.

The files `PPCProffLib.o` and `PPCProffLib` are special libraries that are required for performance monitoring. Note that `PPCProffLib` must also be available at runtime so the Code Fragment Manager can load it into memory. You can ensure this by placing a copy in the Extensions folder or including the directory containing `PPCProffLib` in the Code Fragment Manager's library search path.

Note
You do not need to recompile your code to take advantage of PPCProff. However, you may want to compile with the `-sym on` option so that PPCProff can indicate source files and line numbers. ◆

When you execute your application or shared library, the profiler routines record performance data and write it to the file `Profiler.pgh`. You can then use PPCProff to display the performance data.

To display the information in the `Profiler.pgh` file, use the command

```
PPCProff -xcoff filename.xcoff Profiler.pgh
```

where *filename*`.xcoff` is the name of the XCOFF file generated by PPCLink.

PPCProff can accept more than one `.pgh` file on the command line. For example, you can include several `.pgh` files from successive runs of your application. PPCProff then displays the averages of the performance data collected in the files.

See the *MPW Command Reference* for a listing of PPCProff options.

Listing 17-3 shows sample output from a run of PPCProff using the default options.

Listing 17-3 Sample PPCProff output

```
-- Hierarchical times --
-- Total time: 0.478 gtbus --
           hier%    flat%  #calls
          ------   ------ -------
[P0001]  99.94%    0.00%       0 <unknown caller 1> (<no file>)
         99.94%    0.01%       1 calls to <P0002> __start

[P0002]  99.94%    0.01%       1 __start (StdCRuntime.o)
   +$5c  99.89%    0.03%       1 calls to <P0003> main
   +$64   0.03%    0.03%       1 calls to <P0014> exit

[P0003]  99.89%    0.03%       1 main (c_sample1.c)
  .(12)  82.19%   22.44%       1 calls to <P0004> myPause
   .(6)   7.11%    7.11%       1 calls to <P0007> InitWindows
  .(10)   5.90%    5.90%       1 calls to <P0008> NewWindow
  .(11)   2.45%    0.02%       1 calls to <P0009> showMessage
   .(4)   2.08%    2.08%       1 calls to <P0010> InitGraf
   .(7)   0.13%    0.13%       1 calls to <P0013> InitCursor
   .(5)   0.01%    0.01%       1 calls to <P0015> InitFonts

[P0004]  82.19%   22.44%       1 myPause (c_sample2.c)
   .(4)  37.82%   37.82%    4578 calls to <P0005> SystemTask
   .(3)  21.92%   21.92%    4578 calls to <P0006> Button
   .(3)   0.01%    0.01%       1 calls to <P0006> Button

[P0005]  37.82%   37.82%    4578 SystemTask (•••synthesized glue•••)
```

The first line indicates whether the listing is sorted by hierarchical or flat times.

The next line displays the total time used by the executing application or tool in gtbus. A tbu is a single tick of the PowerPC **time base unit,** and a **gtbu** is a billion such ticks. The ratio between tbus and seconds varies depending on the speed of the processor that is executing your program. However, in most cases, referring to gtbus is sufficient for performance measurements. Note that all the time indicated is “pseudo time.” That is, PPCProff counts only the time spent by the executing program, not the time spent in any of the profiling routines.

Next, PPCProff displays information about each procedure in your program.

```
[P0001]  99.94%    0.00%       0 <unknown caller 1> (<no file>)
         99.94%    0.01%       1 calls to <P0002> __start
```

The first line for each procedure includes (from left to right) an identifier (in this example, `P0001`), the hierarchical and flat times used by the procedure, the number of calls made to the procedure, and the name of the procedure, if any.

In the `P0001` example, the procedure is not within the profiled code, so PPCProff assigns the name `<unknown caller` *x*`>` (most likely the procedure was the Code Fragment Manager launcher). The hierarchical time is high since the launcher calls the application (and all associated procedures) you are profiling, but the flat time is 0 since the launcher is not part of the application.

The next line lists an arc, in this case a call to the `__start` routine (contained in `StdCRuntime.o`). Very little (flat) time is spent in the `__start` routine since its primary purpose is to call other routines.

The listing for `P0002` describes the `_start` procedure:

```
[P0002]  99.94%    0.01%       1 __start (StdCRuntime.o)
   +$5c  99.89%    0.03%       1 calls to <P0003> main
   +$64   0.03%    0.03%       1 calls to <P0014> exit
```

The `__start` routine calls two routines, `main` and `exit`. The `main` routine consumes most of the hierarchical time, while the `exit` procedure uses very little.

Note that this listing contains hex offsets (referenced from the beginning of the procedure) for the routine calls so you can determine which call site generated a particular call. If procedure A contains two separate calls to procedure B, each call is listed as a separate arc. If you compile with the `-sym on` option, PPCProff indicates call sites using line numbers instead of hex offsets, as shown in the listing for `P0003`.

```
[P0003]  99.89%    0.03%       1 main (c_sample1.c)
  .(12)  82.19%   22.44%       1 calls to <P0004> myPause
   .(6)   7.11%    7.11%       1 calls to <P0007> InitWindows
  .(10)   5.90%    5.90%       1 calls to <P0008> NewWindow
  .(11)   2.45%    0.02%       1 calls to <P0009> showMessage
   .(4)   2.08%    2.08%       1 calls to <P0010> InitGraf
   .(7)   0.13%    0.13%       1 calls to <P0013> InitCursor
   .(5)   0.01%    0.01%       1 calls to <P0015> InitFonts
```

PPCProff does not profile routines in shared libraries, but it indicates the glue routines associated with them. For example, procedure `P0005`, `SystemTask`, is a routine in a shared library.

```
[P0005]  37.82%   37.82%    4578 SystemTask (•••synthesized glue•••)
```

PPCProff treats routines in shared libraries as a black box, so the hierarchical and flat times are identical and no arcs are listed.

If a shared library routine does call back the program you are profiling, the callbacks appear as arcs from `<unknown caller x>`. Since PPCProff does not profile shared library routines, you must determine yourself which glue routine corresponds to which `<unknown caller>`.

Proff and PrintProff (Classic 68K Only)

The Proff performance tools provide profiling and performance monitoring for 68K programs compiled from C, C++, and Pascal.

Proff records the following information for each arc:

- the number of times it was executed
- the cumulative flat time for the arc
- the cumulative hierarchical time for the arc

In addition, for calls across segments, Proff records the segment number of the calling routine and of the called routine.

▲ **WARNING**
Because of conflicting uses of the VIA timer, you cannot use the Proff library to measure the performance of applications that use the Sound Manager or the Time Manager. ▲

You enable Proff performance monitoring by the use of compiler and linker options. The following sections explain how.

Required Compiler Options

Proff is a library called `Proff.o`, contained in `{MPW}Libraries:Libraries:`. The `Proff.o` file contains the code that gathers the profiling and performance data. To make use of this library to collect the required data, you must compile the subject programs with specific options and directives (see Table 17-1) and include `Proff.o` in the link.

If the monitored code executes when A5 does not belong to the target program (ROM patch code, or an interrupt handler, for example), the code should be compiled with PC-relative branches and jumps (specify `-b3` on the SC compiler command line).

Table 17-1 Compiler options to enable performance monitoring

Language	Compiler options
C	`-sym on` or `-sym full`; `-trace on` or bracket code to be monitored with `#pragma trace on` and `#pragma trace off`.
C++	`-sym on` or `-sym full`; `-trace on` or bracket code to be monitored with `#pragma trace on` and `#pragma trace off`.
Pascal	`-sym on` or `-sym full`; bracket code to be monitored with `$D++` and `$D--`.

Required Linker Options

The ILink options described in this section are required to make sure that the monitoring code remains resident while the target program is running and, optionally, to make sure that the resident code (the main or resident module plus the code contained in `Proff.o`) can safely exceed the 32 KB segment size limit.

`Proff.o` is built as a single stand-alone segment, `ProffSeg`, that must stay resident while the target program runs. When you link with `Proff.o`, you must merge this segment with another resident segment. For the general case, use the `ILink -sn` option to merge the `ProffSeg` segment with the target program's main segment, as in this example:

```
ILink -sn ProffSeg=main -sym full
```

If you use Proff to monitor code that executes when A5 doesn't belong to the target program (for example, ROM patch code), merge the `ProffSeg` segment with the resident segment containing the patch code. For example,

```
ILink -sn ProffSeg=myPatch -sym full
```

In all cases, specify `-sym full` on the `ILink` command line.

If you suspect that enlarging the program segment by adding `ProffSeg` would violate the 32 KB default size for code segments, you can specify the `-br on` and `-srtsg` options on the `ILink` command line to extend this limit. For example,

```
ILink -sn ProffSeg=main -sym full -br on -srtsg
ILink -sn ProffSeg=myPatch -sym full -br on -srtsg
```

Proff Output File

Executing a program that was built for monitoring with Proff creates an output file containing the performance data. This file is placed in the same directory as the target program and is given the application name with `.proff` appended as a suffix.

If you are monitoring the execution of an MPW tool, the name of the output file is `MPWShell.proff`. However, if you create an alias to ***toolName***`.proff` and then rename the alias `MPW Shell.proff`, the profiling output automatically goes to the file ***toolName***`.proff`.

Although the output file is intended to be processed by PrintProff, its format might be of interest. Listing 17-4 shows part of a sample output file, and Table 17-2 describes the contents of the columns shown in the listing. The column headings do not appear in an actual output file; they are appended here for your convenience. All numbers in the output file are given in hex.

Listing 17-4 A sample `.proff` file

```
 A      B      C      D      E        F          G
 1     64      1      c      1     2fdbc0     24ef0
 1    134      1     a6      1     31db97     9076a
 1    1de      2      e      1      1bca7       e75
 1    270      3      c      d    59076a4     8ecad
 1    29c      7      c      3      2abe7      464c
 3     50      5     10      4     b5a754      8eca
 3     56      5    25a      3    37fa85c     076a4
 3     5c      1    298      3      d2568      65b1
 3     4a      4     14a     3    1404055     5b1b6
 4     a2      4      10    40    18ecad9      9076
 4    15a      7       c     3     226bb       1404
 4    182      8      50     7     464ca       5a75
 4    1ec      4      10     7    65b1b6        85c
 4    206      7      4c      2      90f9       5a7
```

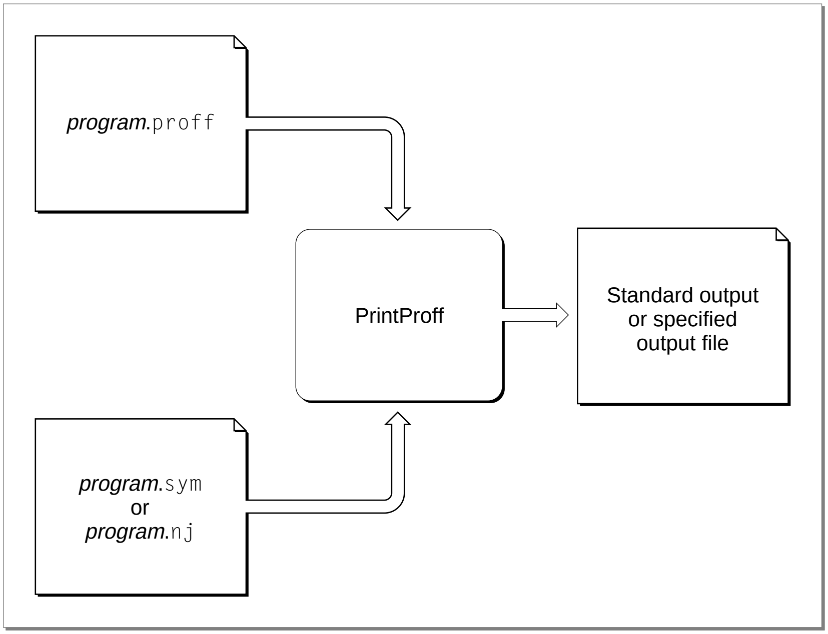
Table 17-2 Contents of the `.proff` file

Column	Meaning
A	Caller segment number
B	Byte offset within caller segment
C	Segment number of called routine
D	Byte offset of called routine within its segment
E	Number of times the arc executed
F	Cumulative hierarchical time
G	Cumulative flat time

Generating a Performance Report

PrintProff is an MPW tool that analyzes the `.proff` data file created by running a program built for profiling. PrintProff uses as input the target program's `.proff` and `.sym` files as shown in Figure 17-2. Using the information contained in these two files, PrintProff generates a performance report. The format and contents of the report are described in "Sample Performance Report" on page 17-20.

Figure 17-2 Generating a performance report



By default, the `.sym` file and the `.proff` file have the same name as the target program with the appropriate extension added. Also by default, PrintProff looks for them in the MPW Shell's current directory. The output from PrintProff goes to standard output unless redirected to an output file.

The syntax for the PrintProff command is

```
PrintProff program [-h|-f][-c][-p]
```

If the `.proff` and `.sym` files are in the current directory, *program* is just the target's terminal name. If they are in some other directory (they must both be in the same one), you must specify a complete pathname for *program*, including the terminal name.

Specifying the Sort Order of Monitored Routines

The `-h` and `-f` options of PrintProff control whether the report lists monitored routines sorted by total hierarchical time (`-h`) or total flat time (`-f`). By default, the report is sorted by hierarchical times. The time units are microseconds.

The report also gives percentage values for each routine's hierarchical and flat time. The distinction in the percentage calculation is of use when doing partial monitoring.

- If you select the hierarchical sort, then the report shows each routine's time as a percentage of the most hierarchically expensive routine's time.
- If you select the flat sort, the report shows each routine's time as a percentage of the sum of the flat times.

If recursion is present, the percentage values are misleading. If recursion is not present and if all routines are monitored, then the largest hierarchical time is that of the original caller and is exactly equal to the sum of the flat times.

If you specify the `-c` option, PrintProff adds to the information for each monitored routine a list of the routines it calls. This redundant information, shown elsewhere in the report, makes it easier to follow the call chain, account for hierarchical time, and view the program's segmentation.

Sample Performance Report

Listing 17-5 shows sample PrintProff output for one routine; the `-c` and `-h` options have been used to generate this output.

Listing 17-5 A sample PrintProff output

```
2________________________________________________________________________

SkelMain
    Total Time:     9,768,889   H       2,135,743       F/1      (95.236%H/20.821%F)
    Called by Procs:
        1  main.(51)
                    9,768,889   H       2,135,743       F/1      1/Main -> 3/TransSkel
    Calls Procs:
        3  Background
                    3,614,659   H       1,786,999       F/233    3/TransSkel -> 1/Main
        4  DoEvent
                    3,531,485   H       2,381,569       F/39
        9  LogEvent
                    448,654     H           2,432       F/39     3/TransSkel -> 1/Main
        32  DoIdle
                     38,347     H          20,059       F/806
```

The entry for the `SkelMain` routine shown in Listing 17-5 is a representative entry, except that it contains the optional section `Calls Procs`. This section is generated by the use of the `-c` option.

Information given for each routine is preceded by the name of the routine. The number shown just above the routine indicates the routine's ranking (in time elapsed) in the overall sort of the monitored routines. Thus, the number 2 above the `SkelMain` routine indicates that this routine is ranked second.

The information shown in the section `Called by Procs` specifies the name of the calling routine. If the name is followed by a number enclosed in parentheses, the number specifies the statement number at which the call occurs. Taking the example shown in Listing 17-5, the `SkelMain` routine has been called from statement number 51 from the `main` routine.

The numbers given for hierarchical times (`H`) and flat times (`F`) represent the total time for all executions of the arc. If the flat time is followed by a slash and a number, the integer following the slash specifies the number of executions of the arc. For example, the call to `Background` is executed 233 times for a total flat time of 1,786,999 microseconds.

If the rightmost column contains an entry like `1/Main -> 3/TransSkel`, the call is an intersegment call. The information shown specifies the number and name of both the calling and called segment.

The number preceding the name of a routine listed in the `Called by Procs` and `Calls Procs` sections is an index of that routine's ranking in the overall sort of monitored routines.

The time given for a routine listed in the `Calls Procs` section is the total time for the arc for the routine called from the monitored routine. This value is not the total time for the called routine unless the monitored routine is its only caller. If the same routine is listed twice in the `Calls Procs` section, this shows that it was called from two different locations in the monitored routine.

Implementation Issues

Please note the following limitations in the use of Proff and PrintProff:

- Although `Proff.o` adds only 29 KB to the monitored program, it requires memory roughly equivalent to the size of the target program's `'CODE'` resources to run. By default, `Proff.o` allocates its memory from temporary memory. If temporary memory is not available or is insufficient, then `Proff.o` allocates its memory from the heap of the target application. If this is also insufficient, `Proff.o` drops into the debugger and asks you to increase the application heap size. `Proff.o` allocates all its memory the first time that a monitored routine executes and does not move memory while the target program executes.
- The times reported for a recursive routine, as reached from each of its callers other than itself, are correct. However, the total hierarchical time reported for a directly or indirectly recursive call is larger than the actual value because of the way in which direct or indirect calls of a routine from itself are summed and included in the total.
- The code in `Proff.o` assumes that routines being profiled and their call sites are located in resources of type `'CODE'` in the resource map of the target application with which `Proff.o` is linked. Any resources added to the map after `Proff.o` first executes are not monitored.
- The procedure `ExitToShell` does not call any of the exit routines that have been installed by `atExit`. Such an exit routine is used by `Proff.o` to write its output file. Therefore, if the host program calls `ExitToShell`, the output file is not generated.

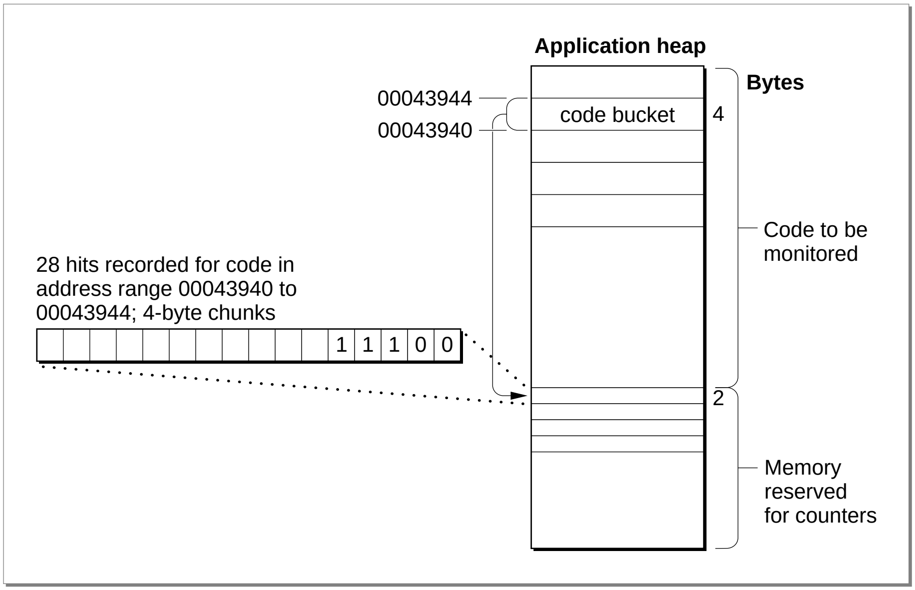
- An explicit `getCaller` function is included in the program being monitored. This function is used by `%_BP` (Proff entry) to determine the return address in the monitored routine's caller. You can override the default `getCaller` function by linking in a custom function. This is useful if the program uses a patched `LoadSeg` trap that alters the return address of the caller.
- The global variable `gStackMax` defines the size of Proff's internal stack. This variable has a default value of 400, which specifies the depth of subroutine calls that can be accommodated. You can set this variable to any desired value prior to initializing Proff (the first `#pragma trace` or `{$D+}`).
- The global variable `gNumBlockRecs` is used to control the number of memory blocks that can be allocated. This variable has a default value of 128. You can increase this number when all the allocated blocks have been used.

MPW Perf (Classic 68K Only)

The MPW Perf performance-monitoring tool samples the program counter (PC) register often enough to obtain a statistically accurate estimate of the program's actual use of time. The code to be measured is divided into "buckets" of 2 or more bytes, and a count of sampled PC values for each bucket during the program's execution is output to a text file. You can then analyze these results by running the report generator `PerformReport`. `PerformReport` merges the output file with a link map of the measured code resources to produce a list of procedures, sorted by the number of PC samples found within the procedure.

When you initialize MPW Perf, a block of memory is allocated in the program's application heap; recorded hits for each bucket of your program's code are stored in 2-byte chunks in this block. Figure 17-3 illustrates the mapping between program buckets and the block of memory allocated to record hits.

Figure 17-3 Recording bucket hits



A **bucket** represents an arbitrary chunk of code whose size you specify when you run MPW Perf. When, at some stated interval, MPW Perf samples the program counter, it records a hit for the bucket if the absolute address of the current instruction falls within the address range of the bucket. For additional information about specifying the frequency of the sampling interval and the size of the bucket counts, see "How MPW Perf Measures Performance" on page 17-25.

▲ **WARNING**

If your application is tuned to ask for only as much heap as it thinks it needs, the allocation of additional memory by MPW Perf is likely to impact your application's memory management strategy. ▲

Components of MPW Perf

MPW Perf consist of the following components:

- A library file (`PerformLib.o`). This file is in the `{Libraries}` folder. Link with this file.
- An interface file (`Perf.h`). This file is in the`{CIncludes}` folder, and it depends only on the standard Mac OS memory-types file `Types.h`.

  An assembly-language interface has not been provided for the performance tools. Assembly-language programmers can use the C interface, which goes directly to the Pascal and assembly-language implementation in `PerformLib`.
- ROM map files. The folder `{MPW}`'ROM Maps' contains the ROM map that is appropriate for your machine. These files are combined with the link map file for your application in order to add location information for the Macintosh Operating System and Toolbox routines to the location information for monitored programs. You usually append the appropriate ROM map to your application's link map for input to the tool PerformReport.
- Sample programs, makefiles, and instructions for execution. These files are in `{CExamples}`Examples:CExamples. Instructions for running the sample programs with performance tools are included in this folder.
- The PerformReport tool. This tool, which is used for analyzing performance data and producing reports, is found in the `{MPW}`Tools folder. For detailed information about the tool, see the command pages in the *MPW Command Reference*. For detailed instructions on how to run this tool, see the instructions in the Examples folder. Examples of the output from this tool are discussed in "Generating a Performance Report" on page 17-36.

How MPW Perf Measures Performance

MPW Perf is designed to give you useful information about the performance of a program without severely altering the program's responsiveness to the user or the program's memory requirements—that is, without changing the characteristics of what is being measured. However, the act of measurement necessarily alters what is being measured in the ways summarized below.

Program Counter Sampling

The frequency with which MPW Perf samples the program counter (PC) register is specified by you. You want to sample frequently enough to obtain a statistically accurate estimate of the actual program performance, but infrequently enough so that overall performance is not affected. The performance-measurement tools use the vertical blanking interrupt (VBL) on 64 KB ROMs and the Time Manager on 128 KB and larger ROMS.

- The Time Manager allows 1 ms (millisecond) resolution in sampling, but this imposes about a 20 percent performance degradation. A value of 4 ms to 10 ms reduces the performance degradation by 4 percent to 10 percent.
- Use of the VBL signal on 64 KB ROMs imposes a sampling rate of approximately 60 times per second (16 ms).

If your application directly uses the VIA timer (or some software that uses it, such as the Time Manager), then you might not be able to use MPW Perf.

In the case of 64 KB ROMs, MPW Perf might not work correctly with programs that make use of VBL tasks.

▲ **WARNING**

If you set the sampling interval too low for your machine, the performance tools might crash or cause your program to run very slowly. It is best to start with a high sampling interval, such as 10 ms or 20 ms, and decrease it only after experience allows you to predict the effect of the shorter interval. For example, if measurements taken with a sampling rate of 10 ms cause your program to run 10 percent slower, then it is probably safe to increase the sampling rate to every 5 ms at a cost of having the program run 20 percent slower. ▲

Bucket Counts

MPW Perf requires 2 bytes of memory for a counter for each “bucket” of code that is measured. For instance, for a 100 KB tool or application, using a bucket size of 16 bytes, about 12,800 bytes are required for the counters. If the ROM is measured, an additional 8 KB, 16 KB, or 32 KB (for 64 KB, 128 KB, or 256 KB ROMs) is required.

If your program spends a substantial amount of time outside code segments and ROM, then you may want to measure RAM “misses.” A *RAM miss* is a PC sample that is not contained in any of the user code segments or the ROM. Because RAM can be quite large, you can specify a second (generally larger) bucket size for RAM "misses." And you can control the range of RAM to be measured by using a low starting address for the first bucket and a high ending address for the last bucket. If the RAM misses are measured, additional memory is required.

The sum of all memory required for counters is allocated as a single contiguous block at the time `InitPerf` is called. For this reason, you should call `InitPerf` fairly early in your initialization, before memory becomes fragmented.

In addition to the memory for bucket counters, MPW Perf uses one master pointer for a handle to some information and allocates a few small structures with `NewPtr` calls.

Using MPW Perf

To use MPW Perf, you need to add calls to the routines included in the `PerformLib.o` file in your application, MPW tool, or stand-alone program and to include `PerformLib.o` in your link. Table 17-3 offers a brief summary of these routines.

Table 17-3 MPW Perf routines

Routine	Effect
`InitPerf`	Specifies the types of measurements to be performed and allocates storage. This function should be called once, near the beginning of your code.
`PerfControl`	Starts and stops measurements. `PerfControl` must be called once (after `InitPerf`) to start measurements. Use `PerfControl` to avoid taking measurements in idle loops, dialog boxes, alert boxes, and other places where the user response time determines performance.
`PerfDump`	Stops measurements (if active) and writes the performance data to an output file. You should call `PerfDump` after measurements are collected for reporting.
`TermPerf`	Stops measurements (if active) and frees the storage. `TermPerf` must be called once after `InitPerf` succeeds and measurement is finished.

Listing 17-6 shows a skeleton Pascal program containing calls to MPW Perf routines. Steps 1 through 7, following the listing, provide additional information about the use of these calls. The section “MPW Perf Routines” on page 17-32 describes each of these routines in detail.

This section presents a detailed explanation for each of the seven steps necessary to install the performance-measurement routines into your code. You need make only a few changes to install these tools. The changes are basically the same, whether you are developing an application, an MPW tool, or a stand-alone program (such as a driver). It is even possible to install performance tools in ROM.

Listing 17-6 A skeleton program with MPW Perf calls

```
                                                {Conditional compilation: Step 1}
$SETC DoPerform := true         {false to exclude }           perf. monitoring}

UNIT ResidentOrMain;{Include interface and allocate pointer to}
        {memory for MPW Perf in main or resident }
        {part of your code}

INTERFACE
USES                                            {Include the interface: Step 2}
$IFC DoPerform
    MemTypes,
    Perf;

    VAR thePerfGlobals: TP2PerfGlobals;         {Pointer to variables: Step 3}
    VAR OldState: boolean;                      {PerfControl result: Step 5}
    VAR MPWPerfErr: OSErr                       {PerfDump result: Step 6}
$ENDC
    {USES and VARs used by your code}

IMPLEMENTATION
$IFC DoPerform
    thePerfGlobals := NIL;                      {Initialize MPW Perf: Step 4}
    IF NOT InitPerf(thePerfGlobals, other params) THEN
        BEGIN
        {report error and terminate}
    END;
$ENDC
```

```
                                        {Turn on measurements: Step 5}
$IFC DoPerform
    OldState := PerfControl (thePerfGlobals, true);
$ENDC

    {CODE TO BE MONITORED}

                                        {Turn off measurements: Step 5}
$IFC DoPerform
    OldState := PerfControl (thePerfGlobals, false);
$ENDC

    {CODE NOT TO BE MONITORED}

                                        {Turn on measurements: Step 5}
$IFC DoPerform
    OldState := PerfControl (thePerfGlobals, true);
$ENDC

    {CODE TO BE MONITORED}

                                        {Dump data to a text file: Step 6}
$IFC DoPerform
    MPWPerfErr := PerfDump(thePerfGlobals, 'Perform.out', other params);
    IF err <> noErr
    THEN
        {report errors during dump};

                                    {Deallocate memory; terminate: Step 7}
    TermPerf (thePerfGlobals);
$ENDC
```

Step 1—Install Under Conditional Compilation

After measuring the performance of your program, you will probably want to make changes, test the changes for correctness, and then repeat the measurements to verify the performance improvements. While making and testing changes, it is very important not to include the performance tools, unless you are confident that the changes do not introduce any new bugs. If your code terminates early for any reason, then the normal system recovery techniques (calls such as `G` in MacsBug, `SysRecover` in the MPW Shell, or `ES` from an application) do not work. In such a case, within a few milliseconds after the Mac OS tries to reuse the memory occupied by the performance tools, a timer interrupt occurs and a system error or crash results. You will probably have to reboot. For this reason, it is advisable to include performance monitoring calls under a conditional flag, as follows:

To enable the conditional compilation of MPW Perf routines in your C program, use directives like the following:

```
/*
    #define PERFORMANCE to turn on the measuring tools.
    #undef PERFORMANCE to turn off the measuring tools.
*/
    #define PERFORMANCE
```

You can then enclose calls to MPW Perf routines by the following conditional compilation statements:

```
#ifdef PERFORMANCE
...
#endif PERFORMANCE
```

Step 2—Include the Interface

In the main body of your MPW C code, you need to include the header file for the performance tools, like this:

```
#include <Perf.h>
```

Step 3—Provide a Pointer to a Block of Variables

How you provide a pointer to the global variables used by MPW Perf depends on the type of code you're measuring. For an application or MPW tool, you can declare a global variable. If you are developing a driver, ROM patch, or other stand-alone code that does not have global variables, then you need to be somewhat creative in finding a location for the pointer. The choices include a local variable on the stack (assuming the stack frame persists long enough), a field of a block allocated and locked down in the heap, or a low-memory location. In any event, the address of the location allocated for the pointer must be passed to the performance routines. In C, write

```
TP2PerfGlobals ThePGlobals;
```

Step 4—Initialize MPW Perf

Somewhere near the beginning of your code's execution and when large chunks of memory are available, you need to call the `InitPerf` function to initialize MPW Perf.

In C, the function `InitPerf` allocates a block on the heap for the performance global variables if `ThePGlobals` is `nil`. If the `ThePGlobals` is not `nil`, `InitPerf` assumes the block is already allocated.

```
ThePGlobals = nil;
if (!InitPerf(&ThePGlobals, ... other parameters ...)) {
        /* report error in initialization and terminate */
    };
```

▲ **WARNING**

Once your code has initialized the performance routines successfully, it is important that you call the termination routine described in "Step 7—Terminate Cleanly" on page 17-32 before your code terminates. Failure to do so almost always results in a fatal system crash. ▲

Step 5—Turn On the Measurements

After initialization succeeds, you can start measurements at any point in your code. The function that starts (and stops) measurements, `PerfControl`, returns the current on-off state as a Boolean value.

In C, write

```
(void)PerfControl(ThePGlobals, true);
```

You can call `PerfControl` with a second argument of `false` to turn performance measurements off. This is useful for disabling sections of code that you don't want to measure, such as the event loop of an application, a dialog box where user response time dominates the compute time, parts of the application that rely on the VIA timer, and so on.

Step 6—Dump the Results

When you reach the end of the code to be measured, you use the `PerfDump` function to have the performance counters written to a text file. If this function encounters any I/O, memory-management, or other system errors, it returns a nonzero return code. You can examine this code to determine the nature of the problem.

The `PerfDump` function takes the output filename as a Pascal string. If the empty string is passed, the name defaults to `Perform.out`.

In C, write

```
OSErr err;
…
err = PerfDump(ThePGlobals, "\pPerform.out", ...other params);
if (err != noErr)
                            /* Code to report errors during dump */
```

Step 7—Terminate Cleanly

After dumping the counters to a text file, you must use the `TermPerf` function to terminate the performance-measurement tools cleanly. `TermPerf` removes the interrupt routine and frees memory associated with the performance global variables and counters.

In C, write

```
TermPerf(ThePGlobals);
```

MPW Perf Routines

This section gives detailed information about parameters to the performance tools routines.

The InitPerf Function

The function `InitPerf` sets up the performance-monitoring interrupt handler and allocates memory for counters. The function returns `true` if initialization is successful, and `false` if it encounters errors.

Here is the MPW C declaration for `InitPerf`:

```
pascal Boolean InitPerf(
    TP2PerfGlobals          *thePerfGlobals,
    short                   timerCount,
    short                   codeAndROMBucketSize,
    Boolean                 doROM,
```

```
    Boolean             doAppCode,
    const               Str255 appCodeType,
    short               romID,
    const               Str255 romName,
    Boolean             doRAM,
    long                ramLow,
    long                ramHigh,
    short               ramBucketSize
);
```

The function `InitPerf` takes a number of parameters, which are described in Table 17-4.

Table 17-4 Parameters to `InitPerf`

Parameter	Setting and effect
`thePerfGlobals`	Contains the address of the pointer to the global variable area. This pointer should be initialized to `nil`; a block will be automatically allocated on the heap for the global variable area.
`timerCount`	Specifies the number of milliseconds between PC samples. For 64 KB ROMs, `timerCount` is the number of VBL events (16 ms each) between PC samples.
`codeAndROMBucketSize`	Sets the bucket size for user code (and the ROM, if ROM measurement is requested). The size of the bucket specifies the number of bytes of code covered by one counter. A separate parameter sets the bucket size for RAM, as described in the last entry in this table. The bucket size may be any integer greater than or equal to 2. MPW Perf forces the bucket size to be a power of 2 by rounding this parameter up to the nearest power of 2.

continued

Table 17-4 Parameters to `InitPerf` (continued)

Parameter	Setting and effect
`codeAndROMBucketSize` *(continued)*	The minimal value for this parameter (2 bytes) can cause individual instructions to be measured. (68K instructions can exceed 2 bytes.) However, this requires an amount of memory equal to the amount of code (and ROM) measured. A better value is 8, which requires only 25 percent of the memory being measured. You can specify larger bucket sizes if memory is scarce, keeping in mind that increasing the bucket size lowers the resolution.
`doROM`	Determines whether the ROM code as well as user code is measured. A value of `true` causes the ROM code to be measured.
`doAppCode`	Determines whether or not user code is measured. A value of `true` causes user code to be measured.
`appCodeType`	A Pascal string that specifies the resource type of user code to be measured. For C application programs this should be `"\pCODE"`; for drivers, it should be `"\pDRVR"`. Resources of the specified type are obtained from the current (top-level) resource file at `InitPerf` time.
`romID`	Indicates ROM types. You'll normally pass a `romID` of 0, indicating the use of one of the predefined ROMs.
`romName`	A Pascal string that specifies a ROM name. You'll normally pass the empty string, indicating the use of a predefined ROM name.
`doRAM`	Determines whether RAM misses are measured. A value of `true` invokes measurement. A *RAM miss* is a PC sample that is not contained in any of the user code segments or the ROM.
`ramLow`	Specifies the lower limits of RAM to measure for misses. This parameter has no effect unless `doRAM` is `true`.

continued

Table 17-4 Parameters to `InitPerf` (continued)

Parameter	Setting and effect
`ramHigh`	Specifies the upper limit of RAM to measure for misses. This parameter has no effect unless `doRAM` is `true`.
`ramBucketSize`	Specifies the bucket size to use for measuring RAM misses. This parameter has no effect unless `doRAM` is `true`.

The PerfControl Function

The `PerfControl` function returns the previous state. You must call `PerfControl` once to begin performance measurements. You can call it more frequently to turn monitoring off and on, thus avoiding measuring uninteresting areas of code, such as idle loops or dialog boxes.

The `PerfControl` function takes two parameters. The parameter `thePerfGlobals` points to the global variable area, initialized by a successful call to `InitPerf`. The `turnOn` parameter turns measurements on (`true`) and off (`false`).

Here is the MPW C declaration for `PerfControl`:

```
pascal Boolean PerfControl(
        TP2PerfGlobals thePerfGlobals,
        Boolean turnOn
);
```

The PerfDump Function

The function `PerfDump` dumps the statistics gathered by the performance tools into a text file suitable either for direct analysis or for processing by `PerformReport`. `PerfDump` calls `PerfControl` to turn off measurements and accepts the following parameters:

`thePerfGlobals` Points to the global variable area, initialized by a successful call to `InitPerf`.

`reportFile` Pascal string that specifies the name of the report file. If this is the empty string, the default name `Perform.out` is used.

`doHistogram` — If `true`, places a histogram (bar graph) after the bucket counts in the data file. The histogram consists of a number of asterisks for each bucket, normalized so that the bucket with the largest number of hits receives a line of asterisks out to `rptFileColumns`.

`rptFileColumns` — Controls the number of columns in the report file. It has no effect unless `doHistogram` is `true`.

Here is the MPW C declaration of `PerfDump`:

```
pascal short PerfDump(
        TP2PerfGlobals              thePerfGlobals,
        const Str255                reportFile,
        Boolean                     doHistogram,
        short                       rptFileColumns
);
```

The TermPerf Procedure

If the call to `InitPerf` succeeds, then you must call the `TermPerf` procedure before terminating the program. Otherwise, a system crash results because the timer interrupt, which is still enabled, jumps to points unknown. The `TermPerf` procedure removes the interrupt handler and frees the storage used by the counters with the parameter `thePerfGlobals`. This parameter points to the global variable area, initialized by a successful call to `InitPerf`.

Here is the MPW C declaration of `TermPerf`:

```
pascal void TermPerf(TP2PerfGlobals thePerfGlobals);
```

Generating a Performance Report

When your code has completed its execution, you call `PerfDump` to generate a performance output file showing the results of the bucket counts. You can analyze this data by using the tool PerformReport. Examples of both the performance output and report files appear in this section. See the *MPW Command Reference* for a detailed description of the tool PerformReport.

Performance Data File

When you call the `PerfDump` function, the results of the performance tests are output to a performance data file. This file is a text file containing the bucket locations and counts. You should call `PerfDump` at the very end of the test so that no interference with program I/O occurs. MPW Perf does not open the performance output file until `PerfDump` is called.

Listing 17-7 shows a sample performance output file as generated by a call to `PerfDump`. Some repeated lines have been omitted, as indicated by ellipses (...).

Notice that the performance data is arranged on a per-segment basis. Only nonzero buckets are reported; in other words, missing buckets had a hit count of zero. `PerfDump` has an option to produce a histogram (bar graph) to the right of the `Hits` column. That option was not exercised in this example.

Listing 17-7 Sample performance output file

```
Performance Parameters
======================

Bytes per bucket, Code and ROM: 8
Bytes per bucket, RAM: 4
Sampling Interval: 4 ms

Performance Summary
===================

Total hits outside of the sampled segments: 2
Maximum hits in one bucket: 872
Total hits in all buckets: 3222

Performance Data
================

Offset Hits | Segment 117  size  20000
=================================================================
 52F8     1 |
 5300     1 |
 53C8    12 |
 53E8     1 |
```

```
 53F0     2 |
 5400     1 |
 5428     1 |
 5728   872 |
...
1B830     9 |
1B838    53 |
1B840    41 |
1B848    61 |
1B850    41 |

Offset Hits | Segment 253  size 1FFFFF
=================================================================
 D6A0     1 |
287D0    40 |
1E6134    1 |
1E8990    1 |
1FB20C  101 |

Offset Hits | Segment 13  size    B68  name STDIO
=================================================================

Offset Hits | Segment 12  size    71E  name SACONSOL
=================================================================


...
Offset Hits | Segment 5  size     D0  name ROMSEG2
=================================================================
   70     2 |
   88     5 |
   90    10 |
...
   B0     4 |
   B8     2 |
   C0     3 |

Offset Hits | Segment 4  size    136  name ROMSEG1
==================================================================
   10     9 |
   18     3 |
   20     5 |
```

```
...
  110    19 |
  118    58 |
  120    46 |

Offset Hits | Segment 3  size     8C  name SEG2
=================================================================
   50     3 |
   58     3 |
   60     9 |
   68     1 |
   70    14 |
   78     1 |

Offset Hits | Segment 2  size     D0  name SEG1
=================================================================
   10     2 |
   18    18 |
   20     4 |
...
   A8     7 |
   B0    12 |
   B8    18 |

Offset Hits | Segment 1  size   101C  name Main
=================================================================
  F38    43 |
  F40   116 |
  F48    78 |
  F50    19 |
  F58    77 |
  F60    66 |
  F68    56 |
```

Generating a Performance Report With PerformReport

Once the performance data file has been generated, you are ready to run the report generator, a tool called PerformReport. This tool merges the performance output file with a link map of the measured code resources to produce a list of procedures, sorted by the number of PC samples found within the procedure.

If your call to `InitPerf` had the parameter `doROM` set to `true`, then you need to append the correct ROM map file to your application's link map before running PerformReport, as in this example:

```
Link -o YourApp -l > LinkMap YourApp.p.o ...etc...
YourApp                         # run your application,
                                # generate Perform.Out

Catenate {MPW}'ROM Maps':romName.Map >> LinkMap
PerformReport -l LinkMap -m Perform.Out
```

Interpreting the Performance Report

PerformReport translates the bucket hit information into routine-based information. Because routines can span buckets, there might be some uncertainty about how bucket hits are related to routine hits. PerformReport attempts to deal with this uncertainty by classifying hits into several categories:

- Definite. When a bucket is completely within a single routine, all hits in the bucket are counted as definite hits in that routine.
- Possible/Probable. When a bucket spans several routines, all hits in the bucket are counted as possible hits in each routine; in addition, the hits in the bucket are counted as probable hits in a particular routine, based on the amount of the bucket that is covered by the particular routine.

The concept of probable hits is not intended to give an accurate statistical picture of the situation. What happens in practice is that buckets are frequently covered by two routines, and almost all of the hits occur in one routine or the other. The intent behind recording possible and probable hits is to give you some feeling for the accuracy of the resulting data.

If the Pascal example `TestPerf.p` is modified to have a bucket size of 8, then the possible hits will be few relative to the definite hits. The exception is the `%I_DIV4` procedure, which will have no definite hits, but shares a bucket with `%I_MUL4`. In fact, there are no divide operations in the sample program; therefore, all hits apparently belonging to `%I_DIV4` really belong to the multiply operations.

If the percentage of definite hits becomes too low, you should consider reducing the requested bucket size.

A sample performance report is shown in Listing 17-8.

Listing 17-8 A sample output of the PerformReport tool

```
PerformReport -- Merges Linker Output and Performance Dump
January 30, 1989

Reading Link Map file: "LinkMap"

Reading Performance Measurements file: "Perform.Out"

PerformReport Parameters:

      8 bytes per bucket, ROM and CODE.
      4 bytes per bucket, RAM.
      2 hits outside code measured.
   3224 hits total,      0.0% outside the segments.
    872 maximum hits in one bucket.

Procedures by possible hits (showing Probable % of time):
Num Segment  : Procedure         Def   Prob   Poss  Prob%

117     Main :      ATRAP68020    497    436    872    28.9%
117     Main :         CHKSLOT      0    436    872    13.5%
117     Main :         DSPATCH      0      0    872     0.0%
117     Main :           RSECT    474     13     26    15.1%
  1     Main :         %I_MUL4    399     14     56    12.8%
...
..1     Main :         %I_DIV4      0      3      5     0.6%
  4  ROMSEG1 :         ROMW100      5      0      0     0.1%
117     Main :         LVL1INT      1      0      1     0.0%
117     Main :     TFSDISPATCH      1      0      0     0.0%
117     Main :         LVL2INT      0      0      1     0.0%

           Total Reported =     67.6%    32.1%         99.8%

PerformReport: That's All Folks!
```

Adding Identification Lines to a Data File

After displaying a title line and giving the names of the files being read, PerformReport has an option (`-e`) to echo lines from the head of the measurements file until the phrase "Performance Data" is encountered. You can use this option to add identification lines at the head of performance-measurement files. Various parameters are gathered from lines that begin with special keywords. Here are the keywords with the phrases they head:

`Bytes`	Bytes per bucket
`Maximum`	Maximum hits
`Performance`	Performance Data
`Total`	Total hits

You are free to add comment lines at the head of a data file, as long as the comment lines do not begin with these keywords.

Implementation Issues

The performance tools have been designed to work "as is" for most common application and stand-alone program runtime environments. However, because the Macintosh has an open architecture, it is possible that actions taken or assumptions made by application code will conflict with the needs of the performance tools. This section discusses possible conflicts and how to resolve them.

Locking the Interrupt Handler

You must lock down both code and data for the performance tools while taking performance measurements. Code for the trap handler must be locked down because the timer interrupts occur asynchronously. Data for the counters must be locked down because handles cannot be assumed to be valid during interrupt processing. The data area for counters cannot be increased at interrupt time because the heap may be inconsistent.

Segmentation

The code that must be locked down at execution time has been placed in segment `Main` and occupies about 1 KB of space. This is because the segment `Main` is usually guaranteed not to be unloaded at run time.

If your application's main segment is too full to allow the performance tools to be linked correctly, then you may retarget the code in `PerformLib.o` by using the Lib tool. However, your application must not have an "unload all segments" routine in its idle procedure. One good segment to retarget to is `PerfMain`, because this segment contains some of the other pieces of the performance tools.

The following MPW commands illustrate how to retarget the code in `PerformLib.o`:

```
Duplicate {Libraries}PerformLib.o temp
Lib -o {Libraries}PerformLib.o -sn Main=PerfMain temp
Delete temp
```

The first command line creates a copy of `PerformLib.o` in `temp`. The second line replaces the original `PerformLib.o` with the output of Lib. The `-sn` option causes all code originally placed in segment `Main` to be in segment `PerfMain`. The third line deletes the file `temp`.

Dirty Code Segments

Because A5 is not valid at interrupt time and there are no low-memory globals assigned to performance measurement, the interrupt routine stores some data values in its code space, including the pointer to the locked-down data. Thus, if your application uses checksums to detect code segments attacked by errors, the performance tools will cause erroneous checksum failures. The easiest fix is simply not to checksum the main segment (or whichever segment you choose).

Movable Code Resources

You must lock down the code for the trap handler and the data area for the counters during performance measurement.

In counting hits in code resource segments, the performance interrupt routine checks that the handle to a measured resource is locked. If it is not locked, the resource is assumed to be unloaded, and PC values are not checked for being within the resource.

The performance tools call `StripAddress`, among other A-traps. If you are using A-trap breaks in MacsBug (as with the `ATHC` command), you may get an A-trap from within the performance tool's interrupt handler, and MacsBug may state that the heap is corrupt. The heap might not actually be corrupt, but simply inconsistent at interrupt time.

Appendixes

The 'ckid' Resource Format

This appendix describes the format of the `'ckid'` resource, which is used by Projector to identify files that have been checked out of the Projector database.

Although Projector is normally used to manage revisions to source code files, it can be used to manage revisions to any type of file. For example, it can be used to track revisions to documentation files as well as to source code. Thus, being aware of the existence and use of this resource might be as important for a word-processing application as it is for a development environment. If your application creates files that might be checked into and out of Projector, you need to understand how the `'ckid'` resource is used and to make sure, at least, that your application does not inadvertently delete this resource from a file.

See Chapter 16, "Managing Projects With Projector," for a description of how to use Projector.

How Projector Uses the 'ckid' Resource

Projector manages sets of files called *projects*. Users are able to check out these files, make changes, and check the files back into a project. Files can be checked out as modifiable or read-only. Only one modifiable version of a file can be checked out at one time. This ensures that two users cannot modify the same version of a file at the same time.

When a file that has been checked into the Projector database is checked out, Projector attaches a `'ckid'` resource to that file. This resource contains information about the file, which includes

- the name of the project to which the file belongs
- the date the file is checked out
- the name of the user who checked it out
- whether the file can be modified

It is important that your application does not delete this resource from its documents. The next section explains what your application must do to support Projector's use of this resource.

Application Support

Users can include your application's files in a Projector database if your application

- does not delete the `'ckid'` resource
- prevents the user from changing files that are checked out read-only

Because the `'ckid'` resource is the file's link to the Projector database, it is extremely important that applications do not delete the resource from a file. For example, when saving changes to an existing file, some applications write the modified data to a temporary file, delete the old file, and rename the temporary file to have the same name as the original. This process deletes the `'ckid'` resource and prevents the user from checking the file back into the Projector database.

Table A-1 describes the fields of the `'ckid'` resource that your application needs to be aware of in order to support Projector files.

Table A-1 `'ckid'` resource fields

Field	Size	Contents
`checkSum`	4 bytes	Stores a checksum used to validate the rest of the resource. The checksum is generated by summing the subsequent longwords in the resource handle, skipping the `checkSum` field itself and any extra bytes at the end that don't compose a longword.
`version`	2 bytes	Stores the version number of the `'ckid'` resource. The information presented in this appendix is valid for version 4 only.

continued

Table A-1 'ckid' resource fields (continued)

Field	Size	Contents
readOnly	2 bytes	Contains 0 if the file is checked out read-only. If the file can be modified, this field is nonzero and contains special version information for Projector.
modifyReadOnly	1 byte	Provides a limited override to the readOnly field. Sometimes users need to modify a file that has been checked out read-only. For example, they have taken a copy home and do not have access to the Projector database to check out a modifiable version. Under MPW, the user can execute the ModifyReadOnly command. This sets the modifyReadOnly field to nonzero, indicating that the file can be edited even though it is checked out read-only. In your application, you might want to include a feature that allows the user to modify a read-only file. To do this, you need to set the modifyReadOnly field to nonzero and recalculate the checksum.

The format of the 'ckid' resource is shown in Listing A-1. Underlined comments are for the fields you can access using the Pascal packed record or C struct declarations described in the next two sections.

Listing A-1 The 'ckid' resource format

```
type 'ckid' {
    unsigned longint;                       // checkSum
    unsigned longint LOC = 1071985200;      // location identifier
    integer version = 4;                    // ckid version number
    integer readOnly = 0;                   // checkout state; 0=modifiable
    byte noBranch = 0;                      // if modifiable & byte not 0, then
                                            // branch was made on checkout
    byte clean = 0, MODIFIED = 1;           // Did user execute “ModifyReadOnly”?
    unsigned int;                           //  1 if history pressent, 0 if not
    unsigned int;                           // length of current comment if
                                            //  history is present
    unsigned longint;                       // date and time of checkout
    unsigned longint;                       // modification date of file
```

```
    unsigned longint;                           // PID.a
    unsigned longint;                           // PID.b
    integer;                                    // user ID
    integer;                                    // file ID
    integer;                                    // revision ID
    pstring;                                    // project path
    Byte = 0;
    pstring;                                    // user name
    Byte = 0;
    pstring;                                    // revision number
    Byte = 0;
    pstring;                                    // filename
    Byte = 0;
    pstring;                                    // task
    Byte = 0;
    wstring;                                    // comment
    Byte = 0;
};
```

The following two sections explain how you provide support for the `'ckid'` resource from a C or Pascal program.

Supporting 'ckid' in Your C Application

To provide support in your C application, include the struct declaration shown in Listing A-2.

Listing A-2 The `CKIDRec` declaration in C

```
typedef unsigned long uLong;
typedef struct {
    uLong      checkSum;
    long       LOC;
    short      version;
    short      readOnly;
    char       branch;
    Boolean    modifyReadOnly;
/* There's more, but this is all we need. */
} CKIDRec, *CKIDPtr, **CKIDHandle;
```

Use the `CKIDIsModifiable` function, shown in Listing A-3, to determine whether the file is modifiable. This function takes a handle to a `'ckid'` resource, checks the value of the `readOnly` field of the resource, and returns `true` if the file is modifiable and `false` otherwise.

Listing A-3 Determining if a file is modifiable—C

```
pascal Boolean CKIDIsModifiable(CKIDHandle ckid)
{
    if (ckid == nil)
        return(true);
    else
        return( ((**ckid).readOnly != 0) ||
        (((**ckid).readOnly == 0) && (**ckid).modifyReadOnly));
}
```

If you want to allow the user to be able to modify a read-only file, you need to set the `modifyReadOnly` field to nonzero and to recalculate the checksum. You can use the `HandleCheckSum` function, shown in Listing A-4, to recalculate the checksum. This function returns the checksum, which you can assign to the `checkSum` field of the `CKIDRec` struct.

Listing A-4 Recalculating the checksum—C

```
pascal uLong HandleCheckSum(Handle h)
{
        long        size;
        uLong       sum = 0;
        uLong       *p;

        size = (GetHandleSize(h) / sizeof(long)) - 1;
        p = (uLong *) *h;
        p++;                        /* skip over checksum field */
        while (size-- > 0) {
            sum += *p++
        }
    return(sum);
}
```

Disassembler Routines

This appendix describes the disassembly routines included in the MPW ToolLibs libraries (`PPCToolLibs.o`, `ToolLibs.o`, and `NuToolLibs.o`) and the interfaces to these routines, which are defined in `Disassembler.h` (for the PowerPC disassembler) and `DisAsmLookup.h` (for the 68K disassembler).

You use the PowerPC disassembler to disassemble PowerPC code, and the 68K disassember to disassemble 68K code. Note that you can access the PowerPC disassembler from 68K-based ToolLibs libraries (`ToolLibs.o` and `NuToolLibs.o`).

The ToolLibs libraries also include routines that you use to control the rotating beach ball cursor and to communicate with the MPW Error Manager. These routines are documented in Chapter 13, "Writing and Building MPW Tools."

The PowerPC Disassembler

The PowerPC disassembler library routine generates assembly code from 4-byte POWER, PowerPC 601, 32-bit PowerPC, and 64-bit PowerPC instructions. Four bytes are passed to the disassember at a time, and the dissassembler returns a mnemonic, an operand, and comment strings that you can use in your calling application.

You can specify options such as the desired target architecture, whether to support extended mnemonics, formatting control, and how to treat invalid instruction encoding conditions. The dissassembler also supports a callback lookup function that allows the caller to substitute names for various operand objects. On successful disassembly, you can determine the properties of the instruction (optional, privileged, whether it is a POWER instruction, and so on) and also find out if the instruction bits are valid.

Note
The PowerPC disassembler contains no explicit references to any runtime library routines (for example, `strcpy`), although the C compiler may generate implicit ones. Also, except for statically declared (read-only) tables, the disassembler uses no other global data. ◆

The basic dissassembler call structure is as follows:

```
DisassemblerStatus ppcDisassembler(unsigned long *instruction,
                                   long dstAdjust,
                                   DisassemblerOptions options,
                                   char *mnemonic,
                                   char *operand,
                                   char *comment,
                                   void *refCon,
                                   DisassemblerLookups lookupRoutine);
```

Given the 4 bytes pointed to by `instruction`, the disassembler disassembles it and places the instruction information in the strings `mnemonic`, `operand`, and `comment`. You can format these strings in a way that is appropriate for your application. If any of the three strings is a `NULL` pointer, that particular information is not returned. Otherwise it is assumed that the associated string buffers are large enough to hold the disassembled output.

Returned comments begin with a `;` (or `#` if you specify the IBM option).

Invalid instructions generate `dc.l` (or `.long` if you specify the IBM option), an operand of the form 0xXXXXXXXX showing the actual instruction, and a comment message that indicates what is wrong with the instruction.

For PC-relative branches, the comment is the destination address, since the only address the disassembler knows about is the address of the code pointed to by the instruction. That address may have no relation to the actual code, so you can specify an adjustment factor, `dstAdjust`, that is *added* to the value normally placed in `comment`.

You can tell the disassembler to call a user-specified callback routine to substitute your own values for operands (such as registers, absolute and relocatable addresses, and signed and unsigned values). You do this by specifying `lookupRoutine` in your call. If you do not need a lookup routine, you should pass `NULL` for the parameter. A `refCon` value is passed to the disassembler, which is in turn passed on to the lookup routine. See the next section for information about specifying `lookupRoutine`.

Note
The disassembler library uses the convention that, with the exception of the called routine name itself (that is, `ppcDisassembler`), all externally visible names (linker symbols and macro names) begin with the letters "`dis`." You should keep this in mind to avoid possible name conflicts. ◆

The Callback Lookup Routine

You can use the optional lookup function to substitute name strings for the various objects that can appear as an operand. This function should return a pointer to a non-null string if you want to specify your own names. If the function returns `NULL` or a null string, the disassember uses its default names.

IMPORTANT
The lookup function must be a C function, not a C++ member function, since the disassembler was built with a C compiler and some compilers have different calling conventions for C and C++. In addition, C++ member functions require the "this" pointer, which is not passed to the callback routine. ▲

The lookup routine should have the form

```
typedef char *(*DisassemblerLookups)(void *refCon,
                                const unsigned long *cia,
                                const DisassemblerLookupType lookupType,
                                const DisLookupValue thingToReplace);
```

The `refCon` value is the "reference constant" that can be used as a communication link between the lookup routine and the caller of the disassembler. This is the same `refCon` value that is passed to the disassembler.

The `cia` parameter is the instruction address passed to the disassembler.

The values of `lookupType` are defined (in `Disassembler.h`) as follows:

```
enum DisassemblerLookupType {           /* types of lookup objects   */
    Disassembler_Lookup_GPRegister,     /*   general-purpose register*/
    Disassembler_Lookup_FPRegister,     /*   floating-point register */
    Disassembler_Lookup_UImmediate,     /*   unsigned immediate      */
```

```
    Disassembler_Lookup_SImmediate,     /*   signed 32-bit immediate */
    Disassembler_Lookup_AbsAddress,     /*   absolute address        */
    Disassembler_Lookup_RelAddress,     /*   relocatable address     */
    Disassembler_Lookup_RegOffset,      /*   offset from base register*/
    Disassembler_Lookup_SPRegister      /*   special-purpose register */
    };
```

The parameter `thingToReplace` holds the value of the object to be substituted. This parameter is of type `DisLookupValue`, which defines a type and form for each `DisassemblerLookupType` as follows:

```
union DisLookupValue {          /* "meaningful" name for each type: */
    unsigned long gpr;          /*   Disassembler_Lookup_GPRegister */
    unsigned long fpr;          /*   Disassembler_Lookup_FPRegister */
    unsigned long ui;           /*   Disassembler_Lookup_UImmediate */
    long si;                    /*   Disassembler_Lookup_SImmediate */
    long absAddress;            /*   Disassembler_Lookup_AbsAddress */
    long relAddress;            /*   Disassembler_Lookup_RelAddress */
    unsigned long spr;          /*   Disassembler_Lookup_SPRegister */
    struct {                    /*   Disassembler_Lookup_RegOffset  */
        short offset;
        unsigned short baseReg;
        } regOffset;
    };
typedef union DisLookupValue DisLookupValue,*DisLookupValuePtr;
```

Table B-1 shows the values for each `DisassemblerLookUpType`.

Table B-1 Values for `DisassemblerLookupType`

`DisassemblerLookUpType` **name**	**Value**
`Disassembler_Lookup_GPRegister`	0:31
`Disassembler_Lookup_FPRegister`	0:31
`Disassembler_Lookup_UImmediate`	integer
`Disassembler_Lookup_SImmediate`	integer
`Disassembler_Lookup_AbsAddress`	address
`Disassembler_Lookup_RelAddress`	address
`Disassembler_Lookup_RegOffset`	D + Ra
`Disassembler_Lookup_SPRegister`	spr

The value for `Disassembler_Lookup_AbsAddress` is an absolute target branch address (that is, the "a" bit is set in the branch instruction). The passed `absAddress` value is the address contained in the instruction.

The `Disassembler_Lookup_RelAddress` value is a relocatable target branch address (the "a" bit in the branch instruction is *not* set). The value of `relAddress` is defined by

`relAddress =` ***destinationAddress*** `+ dstAdjust + cia`

where `cia` is the value of the instruction address passed to the disassembler, and `dstAdjust` is the adjustment factor.

For the `Disassembler_Lookup_RegOffset` value, both the offset (D) and base register (Ra) are passed. The `DisLookupValue.regOffset` value defines how they are packed into `thingToReplace`. You should assign the offset to a `long` value to get its true 32-bit value. However, you can pass it as a `signed short` value since the instruction field from which it came is never more than 16 bits wide.

The lookup for the special-purpose registers (SPRs) is handled differently in that a lookup is only done when the SPR number is not one of the predefined POWER, PowerPC 601, 32-bit PowerPC, or 64-bit PowerPC SPR names. The reason for this is because each PowerPC architecture can have additional SPRs specific to that architecture. The disassembler calls the lookup routine only if the SPR is not one of the predefined numbers shown in Table B-2.

Table B-2 Predefined special-purpose register names

0 MQ	272 SPRG0	528 IBAT0U	1008 HID0
1 MXR	273 SPRG1	529 IBAT0L	1009 HID1
4 RTCU	274 SPRG2	530 IBAT1U	1010 IABR
5 RTCL	275 SPRG3	531 IBAT1L	1013 DABR
6 DEC	280 ASR	532 IBAT2U	1023 PIR
8 LR	282 EAR	533 IBAT2L	
9 CTR	284 TB	534 IBAT3U	
18 DSISR	285 TBU	535 IBAT3L	
19 DAR	287 PVR	536 DBAT0U	
22 DEC		537 DBAT0L	
25 SDR1		538 DBAT1U	
26 SRR0		539 DBAT1L	
27 SRR1		540 DBAT2U	
		541 DBAT2L	
		542 DBAT3U	
		543 DBAT3L	

Not all these SPRs are valid for all targets. The disassembler checks to see if these SPRs are valid for the specified PowerPC architecture. If they are not, the disassembler treats the SPR number as an invalid field and either returns a warning or treats it as invalid (depending on which error option you chose).

The disassembler automatically accepts SPR numbers that are not in Table B-2 and that do not have a substitute lookup name. However, since the disassembler cannot test the validity of these numbers, the `Disassembler_InvSprMaybe` return status flag is set.

Disassembler Options

The file `Disassember.h` defines disassembler option settings as shown in Listing B-1. All options are of type `DisassemblerOptions` (`unsigned long`), and you specify them in your call to the disassembler.

Listing B-1 Option settings as defined in `Disassembler.h`

```
                                       /* target architecture(one must be set):*/
#define Disassemble_Power         0x00000001UL/*    POWER                        */
#define Disassemble_PowerPC32     0x00000002UL/*      32-bit PowerPC             */
#define Disassemble_PowerPC64     0x00000004UL/*         64-bit PowerPC          */
#define Disassemble_PowerPC601    0x00000008UL/*            PowerPC 601          */
                                       /*                                      */
                                       /* error detection options              */
#define Disassemble_RsvBitsErr    0x80000000UL/* invalid reserved bits cause an error */
#define Disassemble_FieldErr      0x40000000UL/* invalid field (regs, BO, etc.)causes */
                                       /* an error                             */
                                       /*                                      */
                                       /* formatting options(reverses presets):*/
#define Disassemble_Extended      0x08000000UL/* extended mnemonics (PPC only)      */
#define Disassemble_BasicComm     0x04000000UL/* basic form in comment if extended    */
#define Disassemble_DecSI         0x02000000UL/* SI fields formatted as decimal      */
#define Disassemble_DecUI         0x01000000UL/* UI fields formatted as decimal      */
#define Disassemble_DecField      0x00800000UL/* fields shown as decimal            */
#define Disassemble_DecOffset     0x00400000UL/* D of D(RA) shown in decimal        */
#define Disassemble_DecPCRel      0x00200000UL/* $+decimal offset instead of $+hex    */
#define Disassemble_DollarHex     0x00100000UL/* $XXX... instead of 0xXXX...         */
#define Disassemble_Hex2sComp     0x00080000UL/* negative hex shown in 2s complement  */
#define Disassemble_MinHex        0x00040000UL/* min nbr of hex digits for values >= 0*/
#define Disassemble_CRBits        0x00020000UL/* crN_LT, crN_GT, crN_EQ, crN_SO       */
#define Disassemble_CRFltBits     0x00010000UL/* crN_FX, crN_FEX, crN_VX, crN_OX      */
#define Disassemble_BranchBO      0x00008000UL/* branch BO meaning if not extended    */
#define Disassemble_TrapTO        0x00004000UL/* trap TO meaning if not extended      */
#define Disassemble_IBM           0x00002000UL/* IBM assembler conventions           */
```

Table B-3 discusses the options in detail. Note that if you do not use an option, then the disassembler acts according to the default behavior.

Table B-3 PowerPC disassembler options

Option	Description

`Disassemble_Power`
`Disassemble_PowerPC32`
`Disassemble_PowerPC64`
`Disassemble_PowerPC601`
	These options specify the target architecture of the instruction being disassembled.
	You must specify one of these options when you call the disassembler.

`Disassemble_RsvBitsErr`	Invalidates the instruction if the reserved bits are incorrectly coded. By default, reserved bits in PowerPC instructions are treated as warnings, with the return status indicating whether the reserved bits were incorrectly coded (1's that should be 0's and vice versa).
`Disassemble_FieldErr`	Invalidates the instruction if the field value is not valid for the target. For example, if you attempted to use an SPR that was not supported by the target architecture, the field is considered invalid. By default, an invalid field value generates a warning as indicated by the return status. Note that if the field has *no* valid decoded value for *any* target, it is always considered an invalid instruction.
`Disassemble_Extended`	Allows extended mnemonic generation. You can only use this option for the 32-bit PowerPC and 64-bit PowerPC architectures, or for 32-bit PowerPC or 64-bit PowerPC instructions on a PowerPC 601. The default is to not generate extended mnemonics.

continued

Table B-3 PowerPC disassembler options (continued)

Option	Description
`Disassemble_BasicCom`	Places the basic form of the instruction in the comment field if an extended mnemonic is generated for it. This option is not recommended since it tends to clutter up the comment field, making it hard to see branch addresses. The default is not to place the instruction in the comment field.
`Disassemble_DecSI`	Generates signed immediate integers (SIs) in decimal form. The default SI format is hexadecimal.
`Disassemble_DecUI`	Generates unsigned immediate integers (UIs) in decimal form. The default UI format is hexadecimal.
`Disassemble_DecField`	Generates all fields (for example, shift/rotate constants) in decimal form. The default field format is hexadecimal.
`Disassemble_DecOffset`	Generates D offsets in operands of the form D(Ra) in decimal form. The default format is hexadecimal.
`Disassemble_DecPCRel`	Generates PC-relative branch addresses in decimal form. The default address form is $±*n*, where *n* is the offset in hexadecimal.
`Disassemble_DollarHex`	Generates hexadecimal values in the form $*XXX*... The default format prefixes the hexadecimal number with 0x.
`Disassemble_Hex2sComp`	Displays negative hexadecimal numbers in two’s-complement form. By default, negative hexadecimal numbers are prefixed with a minus sign (–).

continued

Table B-3 PowerPC disassembler options (continued)

Option	Description
`Disassemble_MinHex`	Specifies a 2-digit minimum when representing positive or negated negative hexadecimal digits. For example, values such as 0x01 or -0x01 are represented with 2 digits, even if they originally came from a 16-bit field. The default representation is to always suggest the size of the instruction field that produced the value or implied value size. For example, 32-bit target addresses are shown as 8 hexadecimal digits.
`Disassemble_CRBits`	Causes Condition Register bits to be referenced using the format cr*N*_*x*, where *N* is a 4-bit condition register field (0:7) and *x* is the name of the bit field (`LT`, `GT`, `EQ`, and `SO` for bits 0, 1, 2, and 3 respectively). This notation is automatically used if you specify extended mnemonics. In the default mode, the condition register bits are referenced as bit numbers 0:31.
`Disassemble_CRFltBits`	This option is identical to `Disassemble_CRBits` except that the bit field names for *x* are `FX`, `FEX`, `VX`, and `OX` for bits 0, 1, 2, and 3 respectively. You can use this option in the context of floating-point operations, but it is up to you to determine that context. In the default mode, the Condition Register bits are referenced as bit numbers 0:31.
`Disassemble_BranchBO`	References branch test BO values with meaningful names (such as `dCTR_NZERO_NOT` or `ALWAYS`). The default BO encodings are referenced as values 0:31 in the basic instruction operand forms.

continued

Table B-3 PowerPC disassembler options (continued)

Option	Description
`Disassemble_TrapTO`	Generates trap TO values in the form *x* | *y* | ... where *x* and *y* (and so on) are the meanings of each of the five TO bits (`LT`, `GT`, `EQ`, `LOW`, and `HI` for bits 0, 1, 2, 3, and 4 respectively). The default TO encodings are referenced as values 0:31 in the basic instruction operand forms.
`Disassemble_IBM`	Uses IBM assembler conventions for comments and invalid instructions. A `#` replaces the `;` as the comment character, and `.long` replaces `dc.l` for the invalid instruction directive mnemonic. The default is to use Apple assembler conventions.

The following definition (found in `Disassembler.h`) lists a set of standard options that gives acceptable results:

```
#define DisStdOptions (Disassemble_Extended  |/* permit extended mnemonics      */
                        Disassemble_DecSI     |/* decimal SIs but hex UIs       */
                        Disassemble_DecField  |/* decimal field numbers         */
                        Disassemble_BranchBO  |/* meaning of branch BO          */
                        Disassemble_TrapTO |/* meaning of trap TO               */
                        Disassemble_CRBits) /* CR bits references as crN_X      */
```

Disassembler Return Status

The disassembler returns a value of type `DisassemblerStatus`. If the value is 0, the instruction was invalid. Other return values indicate various attributes about the instruction as shown in Listing B-2. These status values are defined in `Disassembler.h`.

Listing B-2 Disassembler return status values

```
#define Disassembler_OK             0x0001U /* instruction successfully decoded     */
#define Disassembler_InvRsvBits     0x0002U /* invalidly coded reserved bits        */
#define Disassembler_InvField       0x0004U /* invalidly coded field(s)             */
#define Disassembler_InvSprMaybe    0x0008U /* possibly invalid SPR                 */
#define Disassembler_601Power       0x0010U /* power instruction used with 601      */
#define Disassembler_Privileged     0x0020U /* privileged instruction               */
#define Disassembler_Optional       0x0040U /* optional instruction                 */
#define Disassembler_Branch         0x0080U /* branch instruction                   */
#define Disassembler_601SPR         0x0100U /* SPR valid only for 601 has been used */
#define Disassembler_HasExtended    0x4000U /* possible extended mnemonic           */
#define Disassembler_ExtendedUsed   0x8000U /* the extended mnemonic was generated  */

#define DisInvalid ((DisassemblerStatus)    0x0000U)    /* invalid instruction      */
```

The `Disassembler_OK` status code is always set if the instruction was valid. Table B-4 describes the other status codes. Note that some of these values depend on certain disassembler input options.

Table B-4 Disassembler return status codes

Name	Description
`Disassembler_InvRsvBits`	The instruction had all or some of its reserved bits incorrectly coded, and the `Disassemble_RsvBitsErr` option was not set. This condition is generally considered a warning. If the `Disassemble_RsvBitsErr` option is set, this condition is considered an error and generates an invalid instruction.
`Disassembler_InvField`	The instruction had an incorrect field value for the specified target architecture, and the `Disassemble_FieldErr` option was not set. Note that this code implies that the incorrect field value is still valid for some target (for example, not valid for the PowerPC 601 but valid for the 64-bit PowerPC).

continued

Table B-4 Disassembler return status codes (continued)

Name	Description
`Disassembler_InvSprMaybe`	The `mfspr` or `mrspr` instruction references a possibly invalid special-purpose register (SPR). This condition occurs when an SPR value is not one of the predefined SPR names and either there is no lookup routine or no lookup name is substituted. Since the disassembler cannot know whether the register is valid for the architecture of interest, it sets this flag (instead of `Disassembler_InvField`) to indicate that the SPR may be invalid.
`Disassembler_601Power`	The options specified the PowerPC601 target architecture (`Disassemble_PowerPC601`) and a POWER instruction was disassembled. The PowerPC 601 architecture essentially combines the instruction sets of the POWER and 32-bit PowerPC architectures. However, you could use this flag to filter out POWER instructions to prepare the code for use on a “pure” 32-bit PowerPC or 64-bit PowerPC architecture.
`Disassembler_601SPR`	The options specified the PowerPC 601 target architecture and an `mfspr` or `mrspr` instruction references an SPR valid only for the PowerPC 601.
`Disassembler_Privileged`	The instruction is privileged.
`Disassembler_Optional`	The instruction is optional.

continued

Table B-4 Disassembler return status codes (continued)

Name	Description
Disassembler_Branch	The disassembler has processed one of the following branch instructions: bc[l][a], b[l][a], bclr[l], bcctr[l], or POWER instructions bcr[l], bcc[l]. The disassembler signals branches since you may want to do additional processing on them. For example, a debugger can use this to dynamically show which branch is taken. Note that you may still have to extract the BO and BI fields to determine the condition of the branch.
Disassembler_HasExtended	The instruction possibly has an extended mnemonic. That is, the instruction could have extended mnemonics, but not for all values of its operands. The Disassemble_Extended option does not affect this value.
Disassembler_ExtendedUsed	The instruction has an extended mnemonic, and it was used because the Disassemble_Extended option allowed it. The disassembler formats the operand appropriate to the extended mnemonic. The original basic form is placed in the comment if you specify the Disassemble_BasicCom option.

The 68K Disassembler

The ToolLibs.o and NuToolLibs.o libraries include routines that you can use to disassemble 68K-family machine code. All 68K-family instructions are supported, including 68881, 68882, and 68851 instructions. These routines are summarized in Table B-5.

Table B-5 Routines used to disassemble code

Routine	Effect
`Disassembler`	Disassembles a sequence of bytes.
`endOfModule`	Checks to see if specified address contains an `RTS`, `JMP(A0)`, or `RTD #n` instruction immediately followed by a valid MacsBug symbol.
`InitLookup`	Prepares for use of standard `Lookup` procedure.
`Lookup`	Determines the type of lookup and calls the appropriate procedure.
`LookupTrapName`	Converts a trap instruction to its corresponding trap name.
`ModifyOperand`	Scans operand string returned by `Disassembler` and modifies negative hex values to negated positive value.
`showMacsBugSymbol`	Formats a MacsBug symbol as an operand of a `DC.B` directive.
`validMacsBugSymbol`	Checks that specified bytes represent a valid MacsBug symbol.

To use the disassembler routines included in the ToolLibs libraries, you must include the interface file `DisAsmLookup.h` in your source code and link your object file with the appropriate ToolLibs library.

The `Disassembler` routine is a Pascal routine that disassembles a sequence of bytes starting at a specified address; it returns the number of bytes disassembled and pointers to three strings designating the opcode, operand, and comment. You can then format these strings in a way that is appropriate to your application.

The following sections describe the routines available for disassembly and additional formatting.

The Disassembler Routine

The `Disassembler` routine disassembles a sequence of bytes pointed to by the `FirstByte` parameter. The disassembly consumes the number of bytes specified by the `BytesUsed` parameter. The routine returns the `Opcode`, `Operand`, and `Comment` strings as null-terminated Pascal strings (for easier manipulation with C).

The C declaration for the `Disassembler` routine and for the data used by the routine is as follows:

```
typedef enum {
        _A0_,_A1_,_A2_,_A3_,_A4_,_A5_,_A6_,_A7_,_PC_,_ABS_,_TRAP_,_IMM_} LookupRegs;

pascal void Disassembler(           long DstAdjust,
                                    short *BytesUsed,
                                    Ptr FirstByte,
                                    char *Opcode,
                                    char *Operand,
                                    char *Comment,
                                    Ptr LookUpProc);
```

Interpreting the Opcode, Operand, and Comment Strings

Depending on the opcode and effective addresses (EAs) to be disassembled, the `Opcode`, `Operand`, and `Comment` strings contain the information shown in Table B-6.

Table B-6 Disassembler strings

Case	**Opcode**	**Operand**	**Comment**
Non–PC-relative EAs	`op.sz`	EAs	; `'c...'` (for immediates)
PC-relative EAs	`op.sz`	EAs	; address
Toolbox traps	`DC.W`	`$A`XXX	; `TB` XXXX
OS traps	`DC.W`	`$A`XXX	; `OS` XXXX
Invalid bytes	`DC.W`	`$`XXXX	; `????`

For valid disassembly of processor instructions, `Disassembler` generates the appropriate 68K opcode mnemonic for the `Opcode` string along with a size attribute when required. The source and destination effective addresses are generated as the `Operand` string along with a possible comment. Comments start with a semicolon (;). Traps use a `DC.W` assembler directive as the `Opcode` string, the trap word as the `Operand` string, and a comment indicating the trap number and whether the trap is a Toolbox or Operating System trap. As described later in this appendix, you can generate symbolic substitutions into effective addresses and provide names for traps.

Invalid instructions cause the string `'DC.W'` to be returned in the `Opcode` string. `Operand` is $XXXX (the invalid word) with a comment of `; ????`. `BytesUsed` is 2. This is similar to the trap call case, except for the comment.

Note

The operand effective addresses are syntactically similar to but *not compatible with* the MPW Assembler. This is because `Disassembler` generates byte hex constants as "$XX" and word hex constants as "$XXXX". Negative values (such as $FF or $FFFF) produced by the `Disassembler` procedure are treated as longword values by the MPW 68K Assembler. Thus, it is assumed that `Disassembler` output will *not* be used as MPW Assembler input. If that is the goal, you must convert strings of the form $XX or $XXXX in the `Operand` string to their decimal equivalent The `ModifyOperand` routine is provided for this purpose. ◆

Since the only address that `Disassembler` knows about is the address of the code pointed to by `FirstByte`, the PC-relative address in the comment field may have no relation to reality, that is, the actual code loaded into the buffer. Therefore, to allow the address comment to be mapped back to some actual address, you can specify an adjustment factor, specified by the `DstAdjust` parameter to the `Disassembler` procedure; this factor is *added* to the value that normally would be placed in the comment.

Symbolic Substitutions

The `Disassembler` procedure generates operand effective-address strings as a function of the effective-address mode. A special case is made for A-trap opcode strings. In places where a possible symbolic reference could be substituted for an address (or a portion of an address), the `Disassembler` procedure can call a user-specified routine to do the substitution (using the

`LookupProc` parameter described later). Table B-7 summarizes the generated effective addresses and notes where symbolic substitutions (S) can be made.

Table B-7 Disassembler effective addresses

Mode	Generated effective address	Effective address with substitution (S)
0	DD*n*	D*n*
1	A*n*	A*n*
2	(A*n*)	(A*n*)
3	(A*n*)+	(A*n*)+
4	–(A*n*)	–(A*n*)
5	∂(A*n*)	S(A*n*) or just S (if A*n*=A5, ∂≥0)
6*n*	∂(A,X*n*.Size*Scale)	S(A*n*,X*n*.Size*Scale)
6*n*	(BD,A*n*,X*n*.Size*Scale)	(S,A*n*,X*n*.Size*Scale)
6*n*	([BD,A*n*],X*m*.Size*Scale,OD)	([S,A*n*],X*m*.Size*Scale,OD)
6*n*	([BD,A*n*,X*n*.Size*Scale],OD)	([S,A*n*,X*n*.Size*Scale],OD)
70	∂	S
71	∂	S
72	*±∂	S
73	*±∂(X*n*.Size*Scale)	S(X*n*.Size*Scale)
73	(*±∂,X*n*.Size*Scale)	(S,X*n*.Size*Scale)
73	([*±∂],X*m*.Size*Scale,OD)	([S],X*m*.Size*Scale,OD)
73	([*±∂,X*n*.Size*Scale],OD)	([S,X*n*.Size*Scale],OD)
74	#data	S (#data made comment)
A-traps	$AXXX	S (as opcode, AXXX made comment)

For A-line instructions, you can substitute for the DC.W opcode string. If the substitution is made, the Disassembler procedure generates ,Sys and/or ,Immed flags as operands for Toolbox traps and ,AutoPop for Operating System traps when the bits in the trap word indicate these settings.

	Generated opcode	Operand	Comment	Substituted opcode	Operand	Comment
Toolbox	DC.W	$AXXX	; TB XXXX	S	[,Sys][,Immed]	; AXXX
OS	DC.W	$AXXX	; OS XXXX	S	[,AutoPop]	; AXXX

All displacements (∂, BD, OD) are hexadecimal values shown as a byte ($XX), word ($XXXX), or long ($XXXXXXXX) as appropriate. The *Scale is suppressed if it is 1. The Size is W or L. Note that effective address substitutions can only be made for "∂(A*n*)", "BD,A*n*", and "*±∂" cases.

User-Supplied Procedure for Symbolic Substitutions

For the effective address modes 5, 6*n*, 7*n*, and for A-traps, a co-routine (a procedure) whose address is specified by the LookupProc parameter is called by Disassembler (if the LookupProc parameter is not nil) to do the substitution (or A-trap comment) with a string returned by the procedure. It is assumed that the routine pointed to by LookupProc is a non-nested routine declared as follows in C:

```
pascal void LookUp(                 Ptr            PC,
                                    LookupRegs     BaseReg,
                                    long           Opnd,
                                    char           *S);
```

The PC parameter points to the instruction extension word or A-trap word in the buffer pointed to by the FirstByte parameter of the Disassembler routine.

The BaseReg parameter determines the meaning of the Opnd value and supplies the base register for the "∂(A*n*)", "BD,A*n*", and "*±∂" cases. BaseReg may contain any one of the values shown in Table B-8.

Table B-8 Base register values

_A0_	= 0	==>	A0
_A1_	= 1	==>	A1
_A2_	= 2	==>	A2
_A3_	= 3	==>	A3
_A4_	= 4	==>	A4
_A5_	= 5	==>	A5
_A6_	= 6	==>	A6
_A7_	= 7	==>	A7
ABS	= 9	==>	Abs addr (special case)
IMM	= 11	==>	Immediate (special case)
_PC_	= 8	==>	PC-relative (special case)
TRAP	= 10	==>	Trap word (special case)

Table B-9 shows the contents of `Opnd` depending on the `BaseReg` parameter. For absolute addressing (modes 70 and 71), `BaseReg` contains `_ABS_`. For A-traps, `BaseReg` would contain `_TRAP_`. For immediate data (mode 74), `BaseReg` would contain `_IMM_`.

Table B-9 `BaseReg` and `Opnd` values

BaseReg	Meaning	Operand contains
`_ABS_`	Absolute effective address	The (extended) 32-bit address specified by the instruction's effective address. Such addresses are generally used to reference low-memory globals on a Macintosh.
`_A`*n*`_`	Effective address with a base register	The (sign-extended) 32-bit (base) displacement from the instruction's effective address. In the Macintosh environment, a `BaseReg` specifying A5 implies either global data references or jump-table references. Positive `Opnd` values with an A5 `BaseReg` thus mean jump-table references, while a negative offset would normally mean a global data reference. Using the `-wrap` option to the ILink command can make an `-`*n*`(A5)` address a jump-table reference. Base registers of A6 or A7 would usually mean local data.
`_IMM_`	Immediate data	The (extended) 32-bit immediate data specified by the instruction.
`_PC_`	PC-relative effective address	The 32-bit address represented by "*±∂" adjusted by the `Disassembler` procedure's `DstAdjust` parameter.
`_TRAP_`	A-traps	The entire trap word. The high-order 16 bits of `Opnd` are 0.

The parameter `S` is a Pascal string returned from `Lookup` containing the effective-address substitution string or a trap name for A-traps. `S` is set to `NULL` *prior* to calling `Lookup`. If it is still null on return, the string is not used. If not null, then for A-traps, the returned string is used as an opcode string. In all other cases, the string is substituted as shown in Table B-7 on page B-18.

Depending on the application, you have three choices on how to use the `Disassembler` procedure and an associated `Lookup` procedure:

- You can call `Disassembler` and provide your own `Lookup` procedure. In that case, you must follow the calling conventions discussed above.
- You can provide `nil` for the `LookupProc` parameter, in which case, no `Lookup` procedure is called.
- You can first call the `InitLookup` routine (described in the next section) and pass the address of this unit's standard `Lookup` routine when `Disassembler` is called. In this case, all the control logic to determine the kind of substitution to be done is provided for you, and all that you need to provide are the routines to look up any or all of the following:
  - PC-relative references
  - jump-table references
  - absolute address references
  - trap names
  - references with offsets from base registers
  - immediate data names

The InitLookup Routine

The `InitLookup` routine prepares for use of a unit's `Lookup` procedure.

The C declaration for the routine is

```
pascal void InitLookup          (Ptr PCRelProc,
                                 Ptr JTOffProc,
                                 Ptr TrapProc,
                                 Ptr AbsAddrProc,
                                 Ptr IdProc,
                                 Ptr ImmDataProc);
```

When `Disassembler` is called and the address of this unit's `Lookup` routine is specified, then for PC-relative references, jump-table references, A-traps, absolute addresses, offsets from a base register, and immediate addresses, the associated non-nested procedure specified here is called (if it is not `nil`—all six addresses are preset to `nil`). The calls assume that you have declared these routine as follows (in C).

```
pascal void PCRelProc(long Address, char *S)
pascal void JTOffProc(short A5JTOffset, char *S)
pascal void TrapNameProc(unsigned short TrapWord, char *S)
pascal void AbsAddrProc(long AbsAddr, char *S)
pascal void IdProc(LookupRegs BaseReg, long Offset, char *S)
pascal void ImmDataProc(long ImmData, char *S)
```

Note
`InitLookup` contains initialized data that requires initializing at load time. This is of concern only to users with assembly-language main programs. ◆

The Lookup Routine

The `Lookup` routine performs all the logic to determine the type of lookup and is available to the caller for calls to `Disassembler`.

The routine's C declaration is

```
pascal void Lookup(Ptr PC,              /* Addr of extension/trap word */
                   LookupRegs  BaseReg,/* Base register/lookup mode */
                   long Opnd,           /* Trap word, PC addr, disp. */
                   char *S);            /* Returned substitution     */
```

If you use this procedure, then you must call `InitLookup` prior to any calls to `Disassembler`. For PC-relative references, jump-table references, A-traps, absolute addresses, offsets from a base register, and immediate addresses, the associated non-nested procedure specified in the `InitLookup` call (if not `nil`) is called.

This scheme simplifies the `Lookup` mechanism by allowing you to focus on the problems related to the application.

The LookupTrapName Routine

The `LookupTrapName` routine converts a trap instruction (in `TrapWord`) to its corresponding trap name (in `S`).

The routine's C declaration is

```
pascal void LookupTrapName (unsigned short TrapWord, char *S);
```

The `LookupTrapName` routine is provided primarily for use with `Disassembler`, and its address may be passed to `InitLookup` for use by this unit's `Lookup` routine. Alternatively, there is nothing prohibiting you from using it directly for other purposes or by some other lookup routine.

Note
The tables in this routine make the size of this routine about 9500 bytes. The trap names are fully spelled out in uppercase and lowercase. ◆

The ModifyOperand Routine

The `ModifyOperand` routine scans an operand string, that is, the null-terminated Pascal string returned by `Disassembler` (`NULL` *must* be present here), and modifies negative hex values to negated positive values. For example, $FFFF(A5) would be modified to –$0001(A5). The operand to be processed is passed as the function's parameter, which is then edited in place and returned to the caller.

The routine's C declaration is

```
pascal void ModifyOperand (char *operand);
```

This routine is essentially a pattern matcher and attempts to modify only 2-digit, 4-digit, and 8-digit hex strings in the operand that may be offsets from a base register. If the matching tests are passed, the same number of original digits are output (because that indicates a value's size: byte, word, or long).

For a hex string to be modified, the following tests must be passed:

- There must have been exactly 2, 4, or 8 digits. Only hex strings $XX, $XXXX, and $XXXXXXXX are possible candidates because that is the only way `Disassembler` generates offsets.

- The hex string must be delimited by the left-parenthesis character (`(`) or a comma (`,`). The left-parenthesis character allows offsets for `$`XXXX`(A`*n*`,...)` and `$`XX`(A`*n*`,X`*n*`)` addressing modes. The comma allows for the 68020 addressing forms.
- The hex string must *not* be preceded by the plus-or-minus sign (±). This eliminates the possibility of modifying the offset of a PC-relative addressing mode always generated in the form `*±$`XXXX.
- The hex string must *not* be preceded by the number sign (`#`). This eliminates modifying immediate data.
- The value must be negative. Negative values are the only values modified. A value $FFFF is modified to –$0001.

The validMacsBugSymbol Function

The `validMacsBugSymbol` function checks that the bytes pointed to by `symStart` represent a valid MacsBug symbol.

The function's C declaration is

```
extern char *validMacsBugSymbol (char *symStart, void *limit,
                                  char *symbol);
```

The symbol must be fully contained in the bytes starting at `symStart`, up to, but not including, the byte pointed to by the `limit` parameter.

If a valid symbol is *not* found, then `nil` is returned as the function's result. However, if a valid symbol is found, it is copied to `symbol` (if it is not `nil`) as a null-terminated Pascal string and returns a pointer to where it thinks the *following* module begins. In the "old-style" cases (see Table B-10), this will always be 8 or 16 bytes after the input `symStart`. For new-style Apple Pascal and C cases, this depends on the symbol length, existence of a pad byte, and size of the constant (literal) area. In all cases, trailing blanks are removed from the symbol.

A valid MacsBug symbol consists of _ characters, % characters, spaces, digits, and uppercase and lowercase letters in a format determined by the first 2 bytes of the symbol, as shown in Table B-10. The formats are determined by whether bit 7 is set in the first and second bytes. This bit is removed when it is found OR'ed into the first or second valid symbol characters, or into both.

Table B-10 MacsBug symbol format

1st byte range	2nd byte range	Byte length	Comments
\$20–\$7F	\$20–\$7F	8	Old-style MacsBug symbol format
\$A0–\$7F	\$20–\$7F	8	Old-style MacsBug symbol format
\$20–\$7F	\$80–\$FF	16	Old-style MacApp symbol ab==>b.a
\$A0–\$FF	\$80–\$FF	16	Old-style MacApp symbol ab==>b.a
\$80	\$01–\$FF	*n*	*n* = 2nd byte (Apple compiler symbol)
\$81–\$9F	\$00–\$FF	*m*	*m* = BAnd (1st byte & \$7F) (Apple compiler symbol)

The first two formats in Table B-10 are the basic old-style MacsBug formats. The first byte may or may not have bit 7 set if the second byte is a valid symbol character. The first byte (with bit 7 removed) and the next 7 bytes are assumed to make up the symbol.

The second pair of formats are also old-style formats, used for MacApp symbols. Bit 7 set in the second character indicates these formats. The symbol is assumed to be 16 bytes with the second 8 bytes preceding the first 8 bytes in the generated symbol. For example, 12345678abcdefgh represents the symbol abcdefgh.12345678.

The last pair of formats are reserved by Apple and generated by the MPW 68K compilers. In these cases, the value of the first byte is always between \$80 and \$9F, or with bit 7 removed, between \$00 and \$1F. For \$00, the second byte is the length of the symbol with that many bytes following the second byte (thus a maximum length of 255). Values \$01 to \$1F represent the length itself. A pad byte may follow these variable-length cases if the symbol does not end on a word boundary. Following the symbol and the possible pad byte is a word containing the size of the constants (literals) generated by the compiler.

Note
If `symStart` actually does point to a valid MacsBug symbol, then you can use `showMacsBugSymbol` to convert the MacsBug symbol bytes to a string that could be used as a `DC.B` operand for disassembly purposes. This string explicitly shows the MacsBug symbol encodings. ◆

The endOfModule Function

The `endOfModule` function checks to see if the specified memory address contains an `RTS`, `JMP (A0)`, or `RTD #n` instruction immediately followed by a valid MacsBug symbol. (These sequences are the only ones that can determine an end-of-module when MacsBug symbols are present.)

The function's C declaration is

```
extern char *endOfModule(void *address,
                         void *limit
                         char *symbol
                         void *nextModule);
```

During the check, the instruction and its following MacsBug symbol must be fully contained in the bytes starting at the specified `address` parameter, up to, but not including, the byte pointed to by the `limit` parameter.

If the end-of-module is *not* found, then `nil` is returned as the function's result. However, if an end-of-module is found, the MacsBug symbol is returned in `symbol` (if it is not `nil`) as a null-terminated Pascal string (with trailing blanks removed), and the function returns the pointer to the start of the MacsBug symbol (that is, `address`+2 for `RTS` or `JMP (A0)` and `address`+4 for `RTD #n`). This address may then be used as an input parameter to `showMacsBugSymbol` to convert the MacsBug symbol to a `Disassembler` operand string.

Also returned in `nextModule` is where the *following* module is expected to begin. In the old-style cases (see "The validMacsBugSymbol Function" beginning on page B-25), this will always be 8 or 16 bytes after the input address. For the new style this will depend on the symbol length, existence of a pad byte, and size of the constant (literal) area. See `validMacsBugSymbol` for a description of valid MacsBug symbol formats.

The showMacsBugSymbol Function

The `showMacsBugSymbol` function formats a MacsBug symbol as an operand of a `DC.B` directive.

The function's C declaration is

```
extern char *showMacsBugSymbol (char *symStart,
                                void *limit,
                                char *operand,
                                short *bytesUsed);
```

The first one or two bytes of the symbol are generated as $80+'c' if their high bits are set. All other characters are shown as characters in a string constant. The pad byte, if present, is also shown as $00.

When called, `showMacsBugSymbol` assumes that `symStart` is pointing at a valid MacsBug symbol as validated by the `validMacsBugSymbol` or `endOfModule` routine. As with `validMacsBugSymbol`, the symbol must be fully contained in the bytes starting at `symStart`, up to, but not including, the byte pointed to by the `limit` parameter.

The string is returned in the `operand` parameter as a null-terminated Pascal string. The function also returns a pointer to this string as its return value (`nil` is returned only if the byte pointed to by the `limit` parameter is reached prior to processing the entire symbol—which should not happen if properly validated). The number of bytes used for the symbol is returned in `bytesUsed`. Because of the way MacsBug symbols are encoded, `bytesUsed` may not necessarily be the same as the length of the operand string.

A valid MacsBug symbol consists of _ characters, % characters, spaces, digits, and uppercase and lowercase letters in a format determined by the first 2 bytes of the symbol as described in "The validMacsBugSymbol Function" beginning on page B-25.

The Rez Language

This appendix describes the Rez language. You use the Rez language to write resource description files, which define the resources used by your program. If you have not worked with Rez and DeRez before, you should read Chapter 6, "Creating Noncode Resources and Manipulating Resources," before you read this appendix.

A resource description file can contain preprocessor directives, resource description statements, and comments.

- You use preprocessor directives for macro substitution, conditional compilation, and formatted output statements. The section "Preprocessor Directives" beginning on page C-12 describes the syntax and use of these directives.
- You use resource description statements to specify the contents and format of resources. The section "Resource Description Statements," which begins on page C-18, provides detailed information about these statements.
- You can include comments anywhere blank space is allowed in a resource description file. Comment text is delimited using /* and */; for example,

  ```
  /* This is a comment */
  ```

  or by C++ style delimiters:

  ```
  // Also a comment; the right delimiter is a carriage return.
  ```

The Rez language also includes Rez functions, which are used in preprocessor directives and resource description statements. The section "Rez Functions" beginning on page C-5 describes the syntax and use of these functions.

If you plan to create your own resource types, you need to read "Creating Your Own Resource Types" beginning on page 6-36 before you read this chapter.

If you are writing resources based on standard types, you'll probably find Chapter 6, "Creating Noncode Resources and Manipulating Resources," sufficient for your needs. You might find this chapter useful for additional reference information.

Note that the Rez language is not case sensitive.

Syntax Notation

The syntax diagrams used to describe the Rez language follow the same convention used throughout this manual with the exception of the literal use of brackets. In this appendix, brackets shown in boldface indicate that the brackets are required. The use of brackets in plain font indicates that the enclosed item is optional. For example, a type declaration that uses the syntax

`fill` *fill-size* `[ [`*length*`] ];`

gives you the option of supplying a value for *length*, which, if you specify, must be enclosed in brackets; for example:

```
fill bit [15];
```

This appendix uses certain metasymbols so frequently that they are described in Table C-1 to avoid unnecessary repetition.

Table C-1 Common metasymbols

Metasymbol	Description
attributes	*byte-expression* (see Table C-2 on page C-4)
expression	*integer-constant* *literal-constant* *numeric-function* (see Table C-3 on page C-5) *(expression)* *label* *unary-operator expression* (see Table C-4 on page C-11) *expression operator expression*
ID	*word-expression*
identifier	Must begin with an alphabetic character or underscore (_) and can contain any number of digits, alphabetic characters, and underscores. Identifiers are not case sensitive and can be of any length.

continued

Table C-1 Common metasymbols (continued)

resource-name	*string*
resource-type	*long-expression*
string	"[*character*]..." $"[*hex-digit hex-digit*]..." *string-function* (see Table C-3)

Resource Specification

A resource specification uniquely identifies a resource and sets its attributes. The section "Identifying a Resource and Setting Its Attributes" beginning on page 6-15 provides a detailed description of the resource specification. This section summarizes that material.

The syntax of a resource specification is

resource-type (*ID* [, *resource-name*] [, *attribute* [| *attribute*]…])

resource-type A four-character literal (long expression) specifying the type of the resource. The literal specifying the type is case sensitive—that is, `'MENU'` and `'menu'` designate two different resource types.

ID An integer (word expression) specifying the ID of the resource.

resource-name A string specifying the name of the resource.

attribute One or more of the keywords or corresponding values shown in Table C-2. Use the `OR` operator (`|`) to combine keywords or values. When the resource is created, all the attribute bits are set to 0 by default. The default settings are shown in the first column of Table C-2.

Table C-2 Resource attributes

Default	Alternate	Set value	Meaning
appheap	sysheap	64	Specifies whether the resource is to be loaded into the application heap or the system heap.
nonpurgeable	purgeable	32	Specifies whether the Memory Manager can purge the resource.
unlocked	locked	16	Specifies whether the resource is locked. Locked resources cannot be moved by the Memory Manager. The `locked` attribute overrides the `purgeable` attribute because a locked resource cannot be purged.
unprotected	protected	8	Specifies whether the resource is protected. Protected resources cannot be modified by the Resource Manager.
nonpreload	preload	4	Specifies whether the resource is to be preloaded. Preloaded resources are placed in the heap as soon as the Resource Manager opens the resource file.
unchanged	changed	2	Tells the Resource Manager whether the resource has been changed. Rez does not allow you to set this bit, but DeRez displays it if it is set.

In the following example, keywords are used to set the resource's attributes:

```
Resource 'DLOG' (1000, "First Dialog", preload | purgeable)
```

In the next example, numeric values are used to set the same attributes:

```
Resource 'DLOG' (1000, "First Dialog", 4 | 32 )
```

Rez Functions

A Rez function is characterized by the character prefix `$$`. You can use Rez functions in your resource description file to return the following:

- Information (ID, name, attributes, and size) about the current resource. The current resource is the resource being generated in a `Resource` statement, being included with an `Include` statement, being changed by a `Change` statement, or being deleted by a `Delete` statement.
- Current time and date values.
- The current value of an MPW Shell variable.
- Data found at labels in your resource description file.

These functions are listed in alphabetical order in Table C-3. The sections that follow describe the use of these functions.

Table C-3 Rez functions

Function name	Type	Described in section
`$$ArrayIndex`	Numeric	"Array Information" on page C-7
`$$Attributes`	Numeric	"Resource Information" on page C-6
`$$BitField`	Numeric	"Label Information" on page C-9
`$$Byte`	Numeric	"Label Information" on page C-9
`$$CountOf`	Numeric	"Array Information" on page C-7
`$$Date`	String	"Timestamp and Version Information" on page C-8
`$$Day`	Numeric	"Timestamp and Version Information" on page C-8
`$$Format`	String	"Miscellaneous Functions" on page C-10
`$$Hour`	Numeric	"Timestamp and Version Information" on page C-8
`$$ID`	Numeric	"Resource Information" on page C-6
`$$Long`	Numeric	"Label Information" on page C-9

continued

Table C-3 Rez functions (continued)

Function name	Type	Described in section
`$$Minute`	Numeric	"Timestamp and Version Information" on page C-8
`$$Month`	Numeric	"Timestamp and Version Information" on page C-8
`$$Name`	String	"Resource Information" on page C-6
`$$PackedSize`	Numeric	"Resource Information" on page C-6
`$$Read`	String	"Miscellaneous Functions" on page C-10
`$$Resource`	String	"Miscellaneous Functions" on page C-10
`$$ResourceSize`	Numeric	"Resource Information" on page C-6
`$$Second`	Numeric	"Timestamp and Version Information" on page C-8
`$$Shell`	String	"Miscellaneous Functions" on page C-10
`$$Time`	String	"Timestamp and Version Information" on page C-8
`$$Type`	Numeric	"Resource Information" on page C-6
`$$Version`	String	"Timestamp and Version Information" on page C-8
`$$Weekday`	Numeric	"Timestamp and Version Information" on page C-8
`$$Word`	Numeric	"Label Information" on page C-9
`$$Year`	Numeric	"Timestamp and Version Information" on page C-8

Resource Information

The following functions return information about the current resource:

- `$$Type` returns the type of the current resource.
- `$$ID` returns the resource ID of the current resource.
- `$$Name` returns the name of the current resource.
- `$$Attributes` returns the attributes of the current resource.
- `$$ResourceSize` returns the size, in bytes, of the current resource. When decompiling, this value is the actual size of the resource being decompiled. When compiling, this value represents the number of bytes that have been

compiled at the point when `$$ResourceSize` is encountered. This function can only be used in `Type` statements. See the `'KCHR'` resource in `SysTypes.r` for an example of the use of this function.

- `$$PackedSize` (*start*, *RB*, *RC*) is used only for decompiling resource files. Given an offset, *start*, into the current resource and two integers, *RB* (row bytes) and *RC* (row count), this function calls the `UnpackBits` Toolbox routine *RC* (row count) times. `$$PackedSize()` returns the unpacked size of the data found at *Start*. This function can only be used in `Type` statements. See the `'PICT'` resource in the `Pict.r` file (in the RIncludes folder) for an example of the use of this function.

The first four of these functions are most commonly used with the `Change`, `Delete`, and `Include` statements. The following examples illustrate their use.

- The following `Include` statement merges `'DRVR'` resources from `MyFile`. The merged resources will have the same name and ID but will also have the `sysheap` attribute set.

```
Include "MyFile" 'DRVR' (0:40) AS
'DRVR' ($$ID, $$Name, $$Attributes | 64);
```

  The statement causes `'DRVR'` resources with ID numbers 0 through 40 from `MyFile` to be merged into the Rez output file. The ID range sets up an implicit loop; each time through the loop another `'DRVR'` resource is merged. The merged resource is identical to the resource in `MyFile` except that its attributes have been changed: the `OR` operator combines the `sysheap` attribute with the current value of `$$Attributes`.

- The following command sets the protected bit on all resources of type `'CODE'`.

```
Change 'CODE' to $$Type ($$Id,$$Name, $$Attributes | 8);
```

Array Information

The following functions are used only in `Type` statements to return information about arrays. You must name an array to use these functions.

- `$$ArrayIndex` (*array-name*) returns the index value of the element in the array. In the following example, the first field of each element of the `Sample` array stores the index value of the array element. `$$ArrayIndex` returns 1 for the first element of the array, 2 for the second element, and so forth. In this example, 1 is subtracted from the value returned by `$$ArrayIndex` to create a zero-based array. An error occurs if this function is used outside the scope of an array.

```
wide array Sample {
                    integer = $$ArrayIndex(Sample) - 1;
                    unsigned integer;
                    unsigned integer;
};
```

- `$$CountOf` (*array-name*) returns the number of elements in the array. This allows the program using the resource to determine the number of elements in the array and makes it easier for DeRez to decompile the resource. This function is used to determine the size of variable-size arrays.

  In the following example, the `Fonts` array declares three fields for each element of the array. The first word of the resource contains the number of elements in the array.

```
type 'finf' {
        integer = $$CountOf(Fonts);                    // # of fonts
        array Fonts {
            integer;                                   // Font Number
            unsigned hex integer            plain;     // Font Style
            integer;                                   // Font Size
        };
};
```

Timestamp and Version Information

The following functions are used to return information about the current time and date—that is, the time and date when the resource is compiled. One additional function, `$$Version`, returns the version number of the Rez compiler used to do the compilation.

- `$$Date` returns the current date; for example, "Wednesday, August 30, 1995". The format is generated through the Toolbox procedure `IUDateString`.
- `$$Day` returns the current day in the range 1–31.
- `$$Hour` returns the current hour in the range 0–23.
- `$$Minute` returns the current minute in the range 0–59.
- `$$Month` returns the current month in the range 1–12.
- `$$Second` returns the current second in the range 0–59.

- `$$Time` returns the current time, for example, `23:45:35`. The format is generated through the Toolbox procedure `IUTimeString`.
- `$$Version` returns a string specifying the version of the Rez compiler.
- `$$Weekday` returns the current day of the week in the range 1 (Sunday) through 7 (Saturday).
- `$$Year` returns the current year.

Label Information

The following functions, used only in `Type` statements, allow you to access data at labels. The label must occur before the expression containing any one of the following functions; otherwise, an error is generated.

- `$$BitField` (*label*, *offset*, *length*) returns the *length* (maximum 32) bitstring found at *offset* number of bits from *label*.
- `$$Byte` (*label*) returns the byte found at *label*.
- `$$Long` (*label*) returns the longword found at *label*.
- `$$Word` (*label*) returns the word found at *label*.

The following example shows how you could use the `$$Byte` function to determine the length of a Pascal-style string. With this information you could redefine an `'STR '` type resource without using a Pascal-style string. Here is the definition of `'STR '` from `Types.r`:

```
type 'STR ' {
        pstring;
};
```

Here is a redefinition of `'STR '` using labels and the `$$Byte` function:

```
type 'STR ' {
len:        byte = (stop - len) / 8 - 1;
            string[$$Byte(len)];
stop:       ;
};
```

Miscellaneous Functions

The following functions are used as follows:

- `$$Format` (*fmtstring*, *arguments*...) returns a string. You specify *fmtstring* and *arguments* as you would for the `#printf` directive. `$$Format` works just like `#printf` except that it returns a string rather than printing to standard output. See "Print Directive" on page C-17 for additional information.
- `$$Read` ("*filename*") reads the data fork of the specified file and inserts the data as a hex string into the resource that you are currently building. If *filename* cannot be found in the current directory, `$$Read` searches in directories specified with the `-s` Rez option.
- `$$Resource` ("*filename*", '*type*', *ID*, "*resource-name*") reads the specified resource from *filename* and returns a copy of the contents of the resource as a hex string.
- `$$Shell` ("*string-expression*") returns the current value of the exported Shell variable {*string-expression*}. In specifying *string-expression*, do not include the braces.

  In the following example, the `$$Shell` function is used to return the value of the `{MPW}` Shell variable:

  ```
  #include $$Shell("MPW") "MyProject:MyTypes.r"
  ```

  In this case, `$$Shell("MPW")` returns the name of the directory in which MPW resides. This name is concatenated with the string `MyProject:MyTypes.r` to provide a full pathname for the file to be included. Note that `MPW` is not enclosed in braces.

Expressions

Expressions may consist of simply a number or literal. Expressions may also consist of numeric functions and labels. The syntax of an expression is shown below:

expression	*integer-constant*
	literal-constant
	numeric-function
	(expression)
	label
	unary-operator expression
	expression operator expression

Table C-4 describes the operators you can use in forming arithmetic and logical expressions. Operators are listed in order of precedence from high to low. Groupings indicate equal precedence.

Table C-4 Arithmetic and logical operators

<table>
<tr><th>Precedence</th><th>Operator</th><th>Meaning</th></tr>
<tr><td>1</td><td>( )</td><td>Expression delimiter; forces precedence in expression calculation</td></tr>
<tr><td>2</td><td>–
!
~</td><td>Unary negation (two’s complement)
Unary logical NOT
Unary bitwise NOT (one’s complement)</td></tr>
<tr><td>3</td><td>*
/
%</td><td>Multiplication
Integer division
Modulo (integer division remainder)</td></tr>
<tr><td>4</td><td>+
–</td><td>Plus
Minus</td></tr>
<tr><td>5</td><td><<
>></td><td>Bitwise shift left
Bitwise shift right</td></tr>
<tr><td>6</td><td><
>
<=
>=</td><td>Less than
Greater than
Less than or equal to
Greater than or equal to</td></tr>
<tr><td>7</td><td>==
!=</td><td>Equal to
Not equal to</td></tr>
<tr><td>8</td><td>&</td><td>Bitwise AND</td></tr>
<tr><td>9</td><td>^</td><td>Bitwise XOR</td></tr>
<tr><td>10</td><td>|</td><td>Bitwise OR</td></tr>
<tr><td>11</td><td>&&</td><td>Logical AND</td></tr>
<tr><td>12</td><td>||</td><td>Logical OR</td></tr>
</table>

Preprocessor Directives

You can use preprocessor directives in resource description files to control the compilation of your resources. These directives, listed in Table C-5, allow you to

- include other source files in the compilation
- substitute data for identifier names (macro substitution)
- perform conditional compilations
- output formatted strings

Table C-5 Preprocessor directives

Directive	Meaning
`#include` "*filename*"	Includes specified source file in compilation.
`#define` *identifier* [*value*]	Defines the specified identifier and assigns it the specified value.
`#undef` *identifier*	Deletes definition of the specified identifier.
`#if` *expression*	Includes text that follows in compilation if the specified expression is true.
`#else`	Includes text that follows in compilation if the preceding `#if`, `#ifdef`, or `#ifndef` clause is not true.
`#ifdef` *identifier*	Includes text that follows in compilation if the specified identifier is defined.
`#ifndef` *identifier*	Includes text that follows in compilation if the specified identifier is not defined.
`#elif` *expression*	Includes text that follows in compilation if the specified expression is true.
`#endif`	Terminates `#if`, `#ifdef`, or `#ifndef` construct.
`#printf` (*format*, *arg*, ...)	Generates a message defined by the specified format and arguments.

Syntax of Preprocessor Directives

Observe the following rules in writing preprocessor directives:

- The number sign (#) must be the first character on the line of the preprocessor directive (except for spaces and tabs).
- A preprocessor directive must be expressed on a single line, must begin on a new line, and must be terminated by a return character. If parameters to a directive extend beyond one line, use the backslash (\) escape character before the return to continue the line. In the following example, the two physical lines constitute only one line because the return is escaped.

```
    #define password "I want to be a noble\
rider, like my father was before me."
```

The following four sections describe the directives listed in Table C-5 (grouped according to function) and illustrate some of their uses through examples.

The Include Directive

The `#include` directive takes one parameter, the name of the source file that you want included in the compilation. The `#include` directive can be nested up to ten levels—that is, the included file can itself contain `#include` directives.

Although the `#include` directive and the `Include` statement share the same name, they are not interchangeable. The `#include` directive merges another source file into the compilation; the `Include` statement merges resources that have already been compiled into the output file.

The syntax of the `#include` directive is

`#include "`***filename***`"`

In the following example, the `#include` directive merges the file `MyTypes.r` into the file to be compiled.

```
#include "Athena:MPW:MyProject:MyTypes.r"
```

You can use the `#include` directive to include type declaration files in your resource description file. This saves you the trouble of specifying them with the Rez command, as shown in Figure C-1.

Figure C-1 Using the `#include` directive

menus.r

```
#include "Types.r"
Resource ('MENU', 128) (
.
/*body of resource*/
.
);
```

```
Rez -o menus menus.r
```

menus.r

```

Resource ('MENU', 128) (
.
/*body of resource*/
.
);
```

```
Rez -o menus Types.r menus.r
```

Macro Substitution

The `#define` directive defines an identifier and assigns a value to it. Using the `-d` option of the Rez command to define an identifier has the same effect as using the `#define` directive within the source file.

The four identifiers shown in Table C-6 are predefined. You should not assign different values to these identifiers.

Table C-6 Predefined identifiers

Identifier	Value
`true`	1
`false`	0
`rez`	1 if Rez is running 0 if DeRez is running
`derez`	1 if DeRez is running 0 if Rez is running

The syntax of the `#define` directive is

`#define` ***identifier*** [***value***]

The value you specify for the *identifier* parameter cannot start with a digit, can contain any letter or digit or the underscore character, can be of any length, and is not case sensitive.

If you do not specify a value, the identifier is set to a nonzero value. All characters following the specified identifier are substituted at compile time for the identifier. Thus, *value* can be anything that can be legally substituted for *identifier*. The following examples are taken from the `Types.r` file:

```
#define whiteRGB $FFFF, $FFFF, $FFFF
#define blackRGB 0, 0, 0
#define beepStages { beepStage; beepStage; beepStage; beepStage; }
```

The `#undef` directive deletes the definition of an identifier. The syntax of the `#undef` directive is

`#undef` ***identifier***

If you use more than one `#define` for the same identifier, the last `#define` encountered is the one used, but it is better programming practice to use the `#undef` directive rather than to override previous definitions.

There are many uses for defines. One common use allows you to trigger a conditional compilation by using a `#define` directive. If a resource description file contains an `#if` or `#ifdef` construct like the following example, the statements defined within the construct are compiled only if the identifier has been defined:

```
#ifdef identifier
        resource description statements
    .
    .
    .
#endif
```

For example, several resources in the `Types.r` file append fields that are compiled only if the identifier `SystemSevenOrLater` has a nonzero value. To trigger the compilation of these fields, you must use the `#define` directive before the `#include` directive that merges the `Types.r` file into the input file, as in this example:

```
#define SystemSevenOrLater
#include Types.r
resource ('ALRT', 231) {
/* body of resource*/
};
```

Notice that in the previous example it is not necessary to assign a value to `SystemSevenOrLater`, although you could as long as it is a nonzero value.

Several resources in the `Types.r` file illustrate other uses for `#define` directives. The `'SIZE'` resource uses `#define` directives to ensure compatibility with older `'SIZE'` resources. The `'CNTL'` resource uses `#define` directives to avoid a long enumeration list for the `ProcID` field of the resource.

Conditional Compilation

The Rez compiler recognizes six directives that control conditional compilation: `#if`, `#elif`, `#else`, `#ifdef`, `#ifndef`, and `#endif`. The syntax of conditional constructs using these directives is as follows:

```
#if expression | #ifdef identifier | #ifndef identifier
        /* text to be compiled */
[ #elif expression
        /* text to be compiled */ ]
[ #else ]
        /* text to be compiled */
#endif
```

When used with the `#if` and `#elif` directives, *expression* can take one of the following two forms:

```
defined identifier | defined (identifier)
```

In the following example, comments indicate which message would be compiled. The appropriate `#define` directive needs to precede the conditional construct in the input to the compilation.

```
Resource 'STR ' (199) {
#ifdef English
        "Hello"                 /* use if #define English */
#elif defined French
        "Bonjour"               /* use if #define French */
#elif defined (Japanese)
        "Konnichiwa"            /* use if #define Japanese */
#endif
};
```

Print Directive

The `#printf` directive outputs a formatted string. The `#printf` directive has a format that is exactly the same as that of the `printf` statement in the C language as defined in the MPW C library, with the following exceptions:

- The `#printf` directive cannot contain more than 20 arguments.
- The `#printf` directive cannot end with a semicolon (;).

You can use the `#printf` directive to generate error messages based on your choice of arguments and formats. The Rez `$$Format` function works just like the `#printf` directive except that it returns a string rather than printing to standard output.

The following example shows the use of the `#printf` directive.

```
#define         Tuesday         3
#ifdef Monday
#printf("The day is Monday, day #%d\n", Monday)
#elif defined(Tuesday)
#printf("The day is Tuesday, day #%d\n", Tuesday)
#elif defined(Wednesday)
#printf("The day is Wednesday, day #%d\n", Wednesday)
#else
#printf("DON'T KNOW WHAT DAY IT IS!\n")
#endif
```

The preceding file generates this text:

```
The day is Tuesday, day #3
```

Resource Description Statements

The following sections describe, in alphabetical order, the syntax and use of the seven types of resource description statements used in resource description files. These statements are summarized in Table C-7.

Although the syntax of resource description statements varies for each statement, the following rules apply to the format of all resource description statements:

- Resource description statements begin with the statement name and end with a semicolon (;).
- The Rez language is not case sensitive.
- Tokens in resource description statements can be separated by spaces, tabs, returns, or comments.

Please see the section "Resource Specification" on page C-3 for a detailed discussion of how you specify the type, ID, name, and attributes of a resource.

Table C-7 Resource description statements

Statement	Purpose
`Change`	Changes the name, ID, type, or attributes of the specified resources.
`Data`	Defines a resource and specifies the data for it as a sequence of hexadecimal bytes without any formatting.
`Delete`	Deletes the specified resources.
`Include`	Includes previously compiled resources from the specified file and optionally changes the name, ID, type, or attributes of the resources.

continued

Table C-7 Resource description statements (continued)

Statement	Purpose
`Read`	Defines a resource and reads the data fork of a file as the data for the resource.
`Resource`	Defines a resource and specifies the data for it as a sequence of formatted fields.
`Type`	Declares a resource type and specifies the format of the data.

Change Statement

DESCRIPTION

The `Change` statement allows you to change the vital information (the resource type, ID, name, attributes, or any combination of these at once) for one or more resources.

The `Change` statement is valid only when you use the `-a` (append) option with the Rez command. It makes no sense to change resources while creating a new resource file from scratch.

For information on using Rez functions with the `Change` statement, see the section "Resource Information" on page C-6.

SYNTAX

```
Change resource-type1 [ ( resource-name| ID[:ID] ) ]
    to resource-type2 (ID[, resource-name] [,attributes] );
```

resource-type1 [(*resource-name* | *ID*[:*ID*])]
: Identifies the resource or resources to be changed.

resource-type2 (*ID*[, *resource-name*] [,*attributes*])
: Specifies the new resource type, ID, name, and attributes of the changed resource or resources. If *resource-name* is not specified, the resource name is set to the null string. If *attributes* is not specified, the resource attributes are set to 0. For information on resource attributes, see Table C-2 on page C-4.

EXAMPLE

This command changes the ID of the `'DLOG'` resource from 120 to 1000:

```
Change 'DLOG' (120) to 'DLOG' (1000);
```

This command sets the protected bit on all `'CODE'` resources.

```
Change 'CODE' to $$Type ($$Id, $$Name, $$Attributes | 8);
```

Data Statement

DESCRIPTION

The `Data` statement defines a resource and specifies the data for it as a sequence of hexadecimal bytes without any formatting.

SYNTAX

`Data` *resource-type* (*ID* [,*resource-name*] [,*attributes*]) {
data-string
};

resource-type (*ID* [, *resource-name*] [, *attributes*])
: Identifies the resource being defined and optionally specifies the resource name and attributes. If *resource-name* is not specified, the name is set to the null string. If *attributes* is not specified, the attributes are set to 0.

data-string
: A string of hexadecimal bytes specifying the data for the resource.

CONSIDERATIONS AND USE

When DeRez generates a resource definition, it uses the `Data` statement for any resource that doesn't have a corresponding `Type` declaration or cannot be decompiled for some other reason.

Decompiling a resource without specifying a type declaration for the resource is a handy way to check for field alignment in your resources. See “Creating Your Own Resource Types” on page 6-36 for additional information.

EXAMPLE

In the following example, raw data is used to define a `'PICT'` resource.

```
data 'PICT' (128) {
                $"4F35FF8790000000"
                $"FF234F35FF790000"
};
```

Delete Statement

DESCRIPTION

The `Delete` statement deletes one or more resources from the output file. This statement is useful in situations where you want to maintain the source file, but want to delete certain resources from the output file.

The `Delete` statement is valid only when you use the `-a` (append) option with the Rez command. There’s no reason to delete resources when creating a resource file from scratch.

SYNTAX

`Delete` *resource-type* [(*resource-name* | *ID* [: *ID*])];

resource-type [(*resource-name* | *ID* [: *ID*])]
: Identifies the resources being deleted. If you omit *resource-name* or *ID*, all resources of the specified type are deleted.

CONSIDERATIONS AND USE

You can delete resources that have the protected bit set by using the `-ov` option in the Rez command line.

EXAMPLES

```
Delete 'MENU' (1:123);            /* delete MENU resources 1-123 */

Delete '#STR'  ("English Messages");

Delete 'DLOG';                    /* delete all DLOG resources*/
```

Include Statement

DESCRIPTION

The `Include` statement reads one or more (compiled) resources from the specified file and includes them into the Rez output file. Syntax variations allow you to change the type, ID, name, and attributes of the included resources.

The `Include` statement is not the same as the `#include` directive. The `#include` directive merges (uncompiled) resource description files into the input to Rez.

SYNTAX

`Include` *file* [*type* [(*resource-name* | *ID*[:*ID*])]];

Includes the *type* resource or resources with the specified name, ID, or ID range from the specified file. If the resource name or ID is omitted, all *type* resources are included. If *type* is omitted, all resources from the specified file are included.

`Include` *file* `not` *type;*

Includes all resources from the specified file that are not of the specified type.

`Include` *file type1* `as` *type2;*

Includes all *type1* resources from the specified file as resources of *type2*.

`Include` *file type1* (*resource-name* | *ID*[:*ID*])
`as` *type2* (*ID*[, *resource-name*] [, *attributes*]) ;
Includes the *type1* resource or resources with the specified name, ID, or ID range from the specified file as resources of *type2*. You can optionally specify a resource name and attributes. If you don't specify a resource name, the name is set to the null string. If you don't specify *attributes*, they are set to 0. For additional information on resource attributes, see Table C-2 on page C-4.

EXAMPLES

In this example, all resources from `MyFile` are included:

```
Include "MyFile";
```

In this example, all resources from `MyFile` are included except for `'STR '` resources:

```
Include "MyFile" not 'STR ';
```

In the next example, `'DRVR'` resources are being included from `MyFile`. The type, ID, and name of the included resources remain the same, but the attributes are changed to set the `sysheap` attribute.

```
Include "MyFile" 'DRVR' (0:40) AS
     'DRVR' ($$ID, $$Name, $$Attributes | 64);
```

This statement causes `'DRVR'` resources with ID numbers 0 through 40 from `MyFile` to be included into the Rez output file. The ID range sets up an implicit loop; each time through the loop another `'DRVR'` resource is included.

For additional information, see "Manipulating Resources" on page 6-31 and the section "Resource Information" on page C-6.

Read Statement

DESCRIPTION

The `Read` statement defines a resource and reads the data fork of a file as the data for the resource.

SYNTAX

`Read` *resource-type* (*ID* [, *resource-name*] [, *attributes*]) *file* ;

resource-type (*ID* [, *resource-name*] [, *attributes*])
: Identifies the resource being defined and optionally specifies the resource name and attributes. If *resource-name* is not specified, the name is set to the null string. If *attributes* is not specified, the attributes are set to 0.

file
: Specifies the name of the file whose data fork is being read.

CONSIDERATIONS AND USE

The type of the resource being defined should be appropriate for the data being read: `'TEXT'` or `'STR '` for text, `'PICT'` or `'ICON'` for bitmap data, and so forth.

EXAMPLE

This example defines a resource with ID 101 and resource name About MyApp. The data fork of the file `AppMessage` is used to specify the data for the resource.

```
Read 'STR ' (101, "About MyApp") "AppMessage";
```

Resource Statement

DESCRIPTION

The `Resource` statement defines a resource and specifies the data for it as a sequence of formatted fields.

The `Resource` statement causes an actual resource to be generated. A `Resource` statement must appear after its corresponding `Type` statement. If more than one corresponding `Type` statement appears before the `Resource` statement, the last one read is used. This allows you to override type declarations.

A `Resource` statement can appear anywhere in the resource description file or even in a separate file specified in the Rez command line, or as an `#include` file, as long as it comes after the corresponding `Type` statement.

SYNTAX

`Resource` *resource-type* (*ID* [, *resource-name*] [, *attributes*]) {
    [*data-definition* [, *data-definition*] ...] };

resource-type (*ID* [, *resource-name*] [, *attributes*])
: Identifies the resource being defined and optionally specifies the resource name and attributes. If *resource-name* is not specified, the name is set to the null string. If *attributes* is not specified, the attributes are set to 0.

data-definition
: Specifies the data for a field of the resource. The *data-definitions* in the `Resource` statement correspond (in type and size) to the data declarations in the `Type` statement.

The data definition syntax is as follows:

data-definition
    expression | *symbolic-name* | *string* |
    point-constant
    rect-constant | *switch-data* | *array-data*

symbolic-name
    identifier

point-constant is
{*expression* , *expression*}

rect-constant {*expression* , *expression* , *expression* , *expression*}

switch-data *case-name* { [*data-definition* [, *data-definition*]...]}

array-data {[*array-elemen*t [,*array-element*]...]}

array-element
data-definition [, *data-definition*]...

EXAMPLE

The following example shows the correspondence between the data definitions in the `Resource` statement and the data declarations in the `Type` statement.

```
Type 'TEST' {                               Resource 'TEST' (101) {
    boolean;                                    true,
    fill byte;                                  // no entry needed
    integer;                                    -3087,
    integer = 4;                                // value already defined
    pstring;                                    "My Name",
    align word;                                 // no entry needed
    point;                                      {0, 10},
    rect;                                       {40, 80, 120, 300},
    byte yes, no, maybe=$FF;                    yes,
    switch{                                     two {563421, 7},
        case one:
            key integer = 1;
            byte;
        case two:
            key integer = 2;
            longint;
            unsigned byte;
    };
    bitstring [3];                              1,
    fill bit [13];                              // no entry needed
    char;                                       "q",
    hex string [5];                                 $"12 34 FF 78 9A",
    array [2]{                                  {6, {0, 20}, 15, {50, 100}},
        byte;
        point;
```

```
    };                                                  'APPL',
    literal longint;                                    $FFFF
    hex integer;                                    };
};
```

The following sections provide additional data about providing numeric and string data. For additional information about creating `Resource` statements, see "Creating a Resource Based on a Standard Type" on page 6-20.

Specifying Numeric Data

You specify numeric data for fields declared as `bitstring`, `byte`, `integer`, `longint`, `boolean`, `rect`, and `point`.

You can specify values for numeric data using any one of the formats described in Table C-8. The valid ranges for different formats are shown in the table; however, the value you specify for a numeric data field in a resource will also depend upon the declared size of the field (as shown in Table C-9).

Table C-8 Formats for numeric data

Format	Form	Meaning	Range min/max for `longint`
Decimal	*nnn*...	Signed decimal constant	–2,147,483,648 through 4,294,967,295
Hex	0x*hhh*...	Signed hexadecimal constant	0x7FFFFFFF through 0x80000000
	$*hhh*...	Alternate hexadecimal form	
Octal	0*ooo*...	Signed octal constant	017777777777 through 020000000000
Binary	0B*bbb*...	Signed binary constant	0B1111111111111111111111111111111 through 0B10000000000000000000000000000000
Literal	`'aaaa'`	See the next section, "Specifying Literals"	Long expression

The possible specifications for the size of a numeric field are shown in Table C-9. An error is generated if a value won't fit in the number of bits allocated for the specified type.

Table C-9 Numeric types

Type	Range	Bitstring equivalent
`bitstring`[*length*]	The maximum value for *length* is 32. The brackets are required.	
`byte`	–128 through 255.	`bitsring`[8]
`integer`	–32,768 through 65,535.	`bitstring`[16]
`longint`	–2,147,483,648 through 4,294,967,295.	`bitstring`[32]

Specifying Literals

A literal is a long expression that can contain one to four characters. Characters are printable ASCII characters or escape characters. Characters that are not in the printable character set and are not the double quotation (") or backslash (\) characters can be escaped according to the character escape rules. See the next section, "Specifying String Data," for additional information.

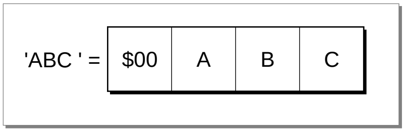
Literals and numbers are treated in the same way by the resource compiler. A literal is a value within single quotation marks; for instance, `'A'` is a number with the value 65; on the other hand, `"A"` is the character A expressed as a string. Both are represented in memory by the bitstring 01000001. (Note, however, that `"A"` is not a valid number and `'A'` is not a valid string.)

Because a literal has a numeric value, you can specify a literal using a numeric expression. The following numeric expressions are all equivalent:

```
'B'
66
'A'+1
```

If there are fewer than four characters in the literal, it is padded with nulls on the left side so that the literal `'ABC'` is stored as shown in Figure C-2.

Figure C-2 The padding of literals



Specifying String Data

You specify string data for fields declared as `char`, `string`, `cstring`, `pstring`, and `wstring`. The length of the specified string depends on the size allowed by its declaration. For additional information about string data size and the internal representation of string data types, see the section “String Type” on page C-40.

You can specify string data in one of two ways:

- A text string. This is a text string:

  ```
  "Man is in love, and love’s what vanishes."
  ```

  The string can contain any printable character except `"` and `\`. However, you can include these characters in your string by using the appropriate escape sequence shown in Table C-10 on page C-30. The string "" is a valid string of length 0.

- A hexadecimal string. This is a hexadecimal string:

  ```
  $"294D 616E 2069 7320 696E 206C 6F76 652C"
  ```

  Spaces and tabs inside a hexadecimal string are ignored. There must be an even number of hexadecimal digits. The string `$""` is a valid hexadecimal string of length 0.

In the `Resource` statement, you enclose string data in double quotation marks (`"`). Any two strings (hexadecimal or text) are concatenated if they are placed next to each other with only white space in between. In this case, returns and comments are considered white space. For example:

```
"Tomorrow,"                         // comments and returns
"and tomorrow,"                     // don’t affect
"and tomorrow"                      // concatenation
```

A side effect of string continuation is that a sequence of two double quotation marks ("") is simply ignored. For example,

```
"Hickory " "Dickory "
"Dock"
```

is the same string as

```
"Hickory Dickory Dock"
```

A separating token (for example, a comma or brace) signifies the end of the string data.

To include a quotation mark ("), backslash (\), or nonprintable character within a string, you need to use the backslash escape character, as explained in the next section, "Escape Characters."

Escape Characters

The backslash character (\) is an escape character that allows you to insert nonprintable characters in a string. For example, to include a newline character in a string, use the escape sequence `\n`.

```
"This is just about the point \n where I want the return."
```

Valid escape sequences are shown in Table C-10.

Table C-10 Resource compiler escape sequences

Escape sequence	Name	Hex value	Printable equivalent
`\t`	Tab	$09	None
`\b`	Backspace	$08	None
`\r`	Return	$0A	None
`\n`	Newline	$0D	None
`\f`	Form feed	$0C	None

continued

Table C-10 Resource compiler escape sequences (continued)

Escape sequence	Name	Hex value	Printable equivalent
`\v`	Vertical tab	$0B	None
`\?`	Rubout	$7F	None
`\\`	Backslash	$5C	\
`\'`	Single quotation mark	$3A	'
`\"`	Double quotation mark	$22	"

You can also use octal, hexadecimal, decimal, and binary escape sequences to specify characters that do not have predefined escape equivalents. The forms are shown in Table C-11.

Table C-11 Numeric escape sequences

Base	Number form	Digits	Example
2	`\0B`*bbbbbbbb*	8	`\0B01000001`
8	`\`*ooo*	3	`\101`
10	`\0D`*ddd*	3	`\0D065`
16	`\0X`*hh*	2	`\0X41`
16	`\$`*hh*	2	`\$41`

Here are some more examples:

```
\077                                  /* 3 octal digits           */
\0xFF                                 /* '0x' plus 2 hex digits   */
\$F1\$F2\$F3                          /* '$' plus 2 hex digits    */
\0d099                                /* '0d' plus 3 decimal digits  */
```

Note
An octal escape code consists of exactly three digits. For instance, to place an octal escape code with a value of 7 in the middle of an alphabetic string, write AB\007CD, not AB\7CD. ◆

Printing Escaped Characters

You can use the DeRez command-line option `-e` to print characters that would otherwise be escaped (characters preceded by a backslash, for example). Normally, only characters with values between $20 and $FF are printed as Macintosh characters. With this option, however, all characters (except null, newline, tab, backspace, form feed, vertical tab, and rubout) are printed as characters, not as escape sequences. See the description of the DeRez tool in the *MPW Command Reference* for additional information.

Type Statement

DESCRIPTION

The `Type` statement declares a resource type and specifies the format of the resource data. The `Type` statement can also be used to specify the data format for individual resources of a specific type.

The `Type` statement does not cause an actual resource to be generated. It simply specifies a template consisting of formatted fields. The template is then used by all subsequent corresponding `Resource` statements. A `Type` statement can appear anywhere in the resource description file, or even in a separate file specified on the command line, or as an `#include` file, as long as it comes before any corresponding `Resource` statements. You can have more than one `Type` statement for a specific resource type. The last one read before a corresponding `Resource` statement is the one used. This allows you to override type declarations.

SYNTAX

`Type` *resource-type* [(*ID*[:*ID*])] {
    [[*label* :] [*data-declaration*]]...};

resource-type [(*ID*[:*ID*])]
Identifies the resource type being declared and optionally restricts the type declaration to a specific resource ID or range of IDs.

label: Specifies an identifier used to calculate the offset of a field in the resource. See “Labels” on page C-47 for additional information.

data-declaration
Declares a resource field and specifies the type and size of the data that it can contain. The display format of the data can also be specified. The data declarations in the `Type` statement correspond to the data definitions in the `Resource` statement.

The syntax for *data-declaration* is as follows:

data-declaration
boolean-type | *numeric-type* |
char-type | *string-type* |
point-type | *rect-type* |
array-type | *switch-type* |
fill-type | *align-type*

boolean-type
`boolean` [= *expression* | *symbolic-value*
[, *symbolic-value*] ...];
symbolic-value
symbolic-name[= *value*]
symbolic-name
identifier

numeric-type
[`unsigned`] [*radix*] *numeric-size*
[= *expression* | *symbolic-value* [, *symbolic-value*]...];
radix
`hex` | `decimal` | `octal` | `binary` | `literal`
numeric-size
`bitstring` [[*length*]] | `byte` | `integer` | `longint`
symbolic-value
symbolic-name[= *value*]
symbolic-name
identifier

char-type

`char` [= *string* | *symbolic-value* [, *symbolic-value*]...];
symbolic-value
symbolic-name[= *value*]
symbolic-name
identifier

string-type

string-specifier [[*length*]]
[= *string* | *symbolic-value* [, *symbolic-value*]...];
string-specifier
[`hex`]`string` | `pstring` | `wstring` | `cstring`
symbolic-value
symbolic-name[= *value*]
symbolic-name
identifier

point-type

`point` [= *point-constant* | *symbolic-value*
[, *symbolic-value*]...];
point-constant
{*expression*, *expression*}
symbolic-value
symbolic-name[= *value*]
symbolic-name
identifier

rect-type

`rect` [= *rect-constant* | *symbolic-value*
[, *symbolic-value*]...];
rect-constant
{*expression*, *expression*, *expression*,
expression}
symbolic-value
symbolic-name[= *value*]
symbolic-name
identifier

array-type

[`wide`] `array` [*array-name* |[*length*]] {*array-list*};
array-list
data-declaration ...

switch-type

`switch` {*case-statement* ...};
case-statement
`case` *case-name*: [*case-body*]
case-body
[*data-declaration*...]
`key` *key-type* = *constant*;
[*data-declaration*...]
key-type
`boolean` | `char` | `point` | `rect` |
[`unsigned`] [*radix*] *numeric-size* |
string-specifier [[*length*]]

fill-type

`fill` *fill-size* [[*length*]];
fill-size
`bit` | `nibble` | `byte` | `word` | `long`

align-type

`align` *align-size*;
align-size
`nibble` | `byte` | `word` | `long`

Note
You can also use the `Type` statement to declare a resource type that uses another resource's type declaration. The syntax for this use is as follows:

`Type` *resource-type1* [(*ID*[:*ID*])] `as` *resource-type2* [(*ID*)]; ◆

Symbolic Values and Constant Values

A data declaration declares a resource field of a given data type. It can also associate symbolic values or constant values with the field. The data declaration can take three forms, as shown in this example:

```
type 'MINE' {                       // declare resource type
    byte;
    byte     off=0, on=1;           // symbolic values
    byte    = 2;                    // constant value
};
```

The first declaration declares a byte field; you supply the data for this field in a subsequent `Resource` statement.

The second byte declaration is identical to the first, except that the two symbolic names `off` and `on` are associated with the values 0 and 1. These symbolic names can be used in the data definition in the `Resource` statement.

The third byte declaration specifies that the value for this field is always 2. In this case, no corresponding data definition would appear in the `Resource` statement.

Symbolic values simplify the reading and writing of resource definitions. Symbol definitions have the form

symbolic-name[= *value*]

where *symbolic-name* is an *identifier*. For numeric data, the = *value* part of the statement can be omitted. If a sequence of values consists of consecutive numbers, the explicit assignment can be left out. If *value* is omitted, it's assumed to be one greater than the previous value. The first value in the list is assumed to be 0. This is true for bitstrings (and their derivatives `byte`, `integer`, and `longint`). In the following example, the symbolic names `documentProc`, `dBoxProc`, `plainDBox`, `altDBoxProc`, and `noGrowDocProc` are automatically assigned the numeric values 0, 1, 2, 3, and 4, respectively:

```
integer     documentProc, dBoxProc, plainDBox, altDBoxProc,
            noGrowDocProc, zoomProc=8, rDocProc=16;
```

Memory is the only limit to the number of symbolic values that you can declare for a single field. There is also no limit to the number of names you can assign to a given value. In the following example, several of the names specified are assigned identical values.

```
integer     documentProc=0, dBoxProc=1, plainDBox=2, altDBoxProc=3,
            rDocProc=16, DOcument=0, Dialog=1, DialogNoShadow=2,
            ModelessDialog=3, DeskAccessory=16;
```

Note

Symbolic values and constant values cannot be used with `array`, `switch`, `fill`, and `align` data declarations. ◆

Boolean Type

A `boolean` data type is a single bit with two possible states: 0 (or the predefined identifier `false`) and 1 (or the predefined identifier `true`). The syntax of the Boolean data declaration is

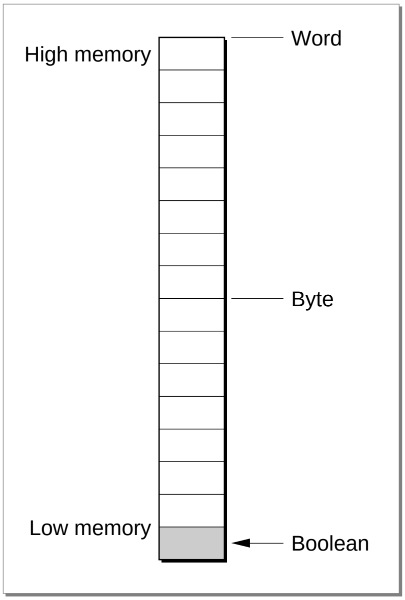
`boolean` [= *expression* | *symbolic-value* [, *symbolic-value*]...] ;

In the following statement the symbolic value `notHighLevelEventAware` is equivalent to 0 (`false`) because that is its ordinal value in the enumeration:

```
boolean            notHighLevelEventAware,
                   isHighLevelEventAware;
```

The `boolean` type declares a 1-bit field because `boolean` is equivalent to `unsigned decimal bitstring [1]`. Figure C-3 illustrates how a Boolean value is stored in memory.

Figure C-3 Boolean value in memory



As Figure C-3 shows, a Boolean value is stored in the most significant bit of a byte. Because of this, you must use either a `fill` or `align` data declaration to fill up the remaining bits and have the next data field start on a word boundary.

Enumerated Types

The syntax for enumerated type declarations is as follows:

enum [*tag*] (*identifier* [=*value*] [,*identifier* [=*value*]...])

The initial tag identifier is optional and is ignored. The members of `enum` declarations are handled internally as though defined by `#define` statements. Otherwise, enumerated types in Rez correspond to `enum` statements in C.

Numeric Types

The syntax for numeric data declarations is as follows:

[unsigned] [*radix*] *numeric-size* [= *expression* | *symbolic-value* [, *symbolic-value*]...];

radix = hex | decimal | octal | binary | literal

numeric-size = bitstring [[*length*]] | byte | integer | longint

The `unsigned` prefix signals DeRez that the number should be displayed without a sign—that is, the high-order bit can be used for data and the numeric value of the integer cannot be negative. The `unsigned` prefix is ignored by Rez but is needed by DeRez to correctly represent a decompiled number.

Rez uses a sign if it is specified in the corresponding resource data. Precede a signed negative constant with a minus sign (–). The constants $FFFFFF85, –$7B, and –123 are equivalent in value.

The *radix* argument specifies the display of numeric values by DeRez. If you do not specify *radix*, `decimal` is used by default. The *radix* argument is ignored by Rez.

You can use the `literal` specification to have DeRez display a number as a series of ASCII bytes. This comes in handy for fields specifying resource types, file types, or application signatures. The following type declaration

```
type 'MyRs' {
    literal longint;                /* resource type */
    integer;                        /* resource id */
};
```

could be used to compile the following resource definition:

```
resource ('MyRs', 1 ) { 'APPL', 0};
```

Valid ranges for *numeric-size* are listed in Table C-12. An error is generated if a value won't fit in the number of bits defined for the specified type.

Rez uses integer arithmetic and stores numeric values as integer numbers. Rez translates Booleans, bytes, integers, and long integers to bitstring equivalents. All computations are done in 32 bits and truncated.

Table C-12 Numeric types

Numeric type	Description	Bitstring equivalent
`bitstring` [*length*]	Declares a bitstring of *length* bits. The maximum value for *length* is 32. The brackets are required. The bits are allocated from the most significant to least significant in successive bytes.	
`byte`	Declares a byte field. The valid range for `byte` is –128 through 255.	`bitstring [8]`
`integer`	Declares an integer field. The valid range for `integer` is –32,768 through 65,535.	`bitstring [16]`
`longint`	Declares a long integer field. The valid range for `longint` is –2,147,483,648 through 4,294,967,295.	`bitstring [32]`

Char Type

The syntax for `char` type declarations is as follows:

`char [=` *string* `|` *symbolic-value* `[,` *symbolic-value*`]...];`

Type `char` declares an 8-bit field; this is equivalent to `string[1]`.

The following example shows a `char` type declaration and its corresponding data definition:

```
Type 'SYMB' {
    char dollar = "$", percent = "%";
};

Resource 'SYMB' (128){
    dollar
};
```

String Type

You declare a `string` type using the following syntax:

string-specifier`[ [`*length* `] ] [ =` *string* `|` *symbolic-value* `[,` *symbolic-value* `]...];`

The *string-specifier* argument is one of the keywords listed in Table C-13. Note that if you specify *length,* it must be enclosed in brackets.

Table C-13 String specifiers

String specifier	Description	Length min/max
`[hex]string`	Plain string (no length indicator or termination character). The `hex` prefix tells DeRez to display `string` as a hex string. `string` **[** *length* **]** contains *length* characters and is *length* bytes long.	1 through 2,147,483,647
`pstring`	Pascal string. A leading byte containing the length of the string is generated. `pstring` **[** *length* **]** contains *length* characters and is *length* + 1 bytes long.	1 through 255

continued

Table C-13 String specifiers

String specifier	Description	Length min/max
`wstring`	Pascal string. A leading 2-byte value is generated containing the length of the string. `wstring` **[** *length* **]** contains *length* characters and is *length* + 2 bytes long.	1 through 65,535
`cstring`	C string. A trailing null byte ('\0') is generated. `cstring` **[** *length* **]** contains *length* – 1 characters and is *length* bytes long. A C string of length 1 can be assigned only the value `""` because `cstring [1]` has room only for the terminating null.	1 through 2,147,483,647

If you do not specify a value for *length*, the length for a `pstring`, `cstring`, or `wstring` is equal to the number of characters in the corresponding data definition.

If you specify *length* and the corresponding data is shorter, the string is padded on the right (with null characters). If the corresponding data is larger, the string is truncated on the right and a warning message is given.

▲ **WARNING**
Fields that follow string data fields are not automatically aligned to word boundaries. ▲

▲ **WARNING**
A null byte within a `cstring` is a termination indicator and might confuse DeRez and C programs. However, the full string, including the explicit null and any text that follows it, will be stored by Rez as input. ▲

Point and Rect Types

Because points and rectangles appear so frequently in resources, they have their own simplified syntax:

`point` [= *point-constant* | *symbolic-value* [, *symbolic-value*]...];
point-constant = {*expression*, *expression*}

`rect` [= *rect-constant* | *symbolic-value* [, *symbolic-value*]...] ;
rect-constant = {*expression*, *expression*, *expression*, *expression*}

A `point` is defined by two 16-bit integers. For example,

```
point = {12,14 };
point       offcenter = {20,30}, center = {100,100};
```

A `rect` (rectangle) is defined by its upper-left and lower-right points. For example,

```
rect = {12,14,33,64};
rect        small = {0,0,30,30}, medium = {0,0,100,100};
```

Array Types

The syntax of the `array` declaration is

`[wide] array` [*array-name* | [*length*]] {*array-list*};
    array-list
        data-declaration ...

The `wide` prefix specifies that DeRez should display the array fields separated by a comma and space rather than by a comma, return, and tab.

Either *array-name* or **[***length***]** may be specified.

- The *length* argument is an expression that specifies the number of elements in the array. The array must contain exactly that number of elements. The enclosing brackets are required.
- The *array-name* argument is an identifier. If the array is named, then a preceding data declaration should refer to the array in a constant expression with the Rez `$$CountOf` function; otherwise, DeRez treats the array as an open-ended array. The Rez `$$CountOf` function returns the number of array elements from the resource data.

The *array-list* argument defines the size and format of each field of the array.

Arrays can be nested.

In the following example, the `Fonts` array declares three fields for each element of the array. The preceding data declaration (`integer = $$CountOf(Fonts);`) calculates the number of elements in the array.

```
type 'finf' {
        integer = $$CountOf(Fonts);                /* # of fonts   */
            array Fonts {
            integer;                               /* font number  */
            unsigned hex integer            plain; /* font style   */
            integer;                               /* font size    */
        };
};
```

A special problem is posed by the following `Type` statement; it declares an array of constant elements. Note how semicolons are used in the subsequent `Resource` statements to generate array elements.

```
Type 'xyzy' {
    array Increment {
        integer = $$ArrayIndex(Increment);
    };
};

Resource 'xyzy' (128) {
    {       // 0 elements
    }
};

Resource 'xyzy' (129) {
    {       // 1 element
    ;
    }
};
```

```
Resource 'xyzy' (130) {
    {       // 2 elements
    ;;
    }
};
```

For additional information about Rez functions that are used with array type declarations, see “Array Information” on page C-7.

Switch Type

The `switch` type specifies a list of `case` statements for one or more fields of a resource. The syntax for the `switch` type declaration is

`switch {`*case-statement…*`};`

case-statement

    `case` *case-name*`: [`*case-body*`]`

    case-body

        `[`*data-declaration…*`]`
        `key` *key-type* `=` *constant*`;`
        `[`*data-declaration…*`]`

        key-type

            `boolean | char | point | rect |`
            `[unsigned] [`*radix*`]` *numeric-size* `|`*string-specifier* `[ [`*length*`] ]`

Case-name is an identifier.

Case-body can contain any number of data declarations and must include one constant declaration per case, in the form

`key` *key-type* `= constant;`

This key definition is compiled by Rez into the resource to identify the case and is then used by DeRez to decode the resource. In the following example, the `Switch` statement is used to declare two cases: `Number` and `Address`.

```
switch{
case Number:
    key integer = 1;
    byte;
```

```
case Address:
    key integer = 2;
    longint;
    unsigned byte;
};
```

Note that `case` statements can declare different types of fields and a varying number of fields. Each key definition must uniquely identify a case and must be the same data type for each case of the same `Switch` statement. The following declaration is invalid:

```
switch{                                 // this declaration won't work
case Number:
    key integer = 1;                    // integer data type here
    byte;
case Address:
    key string = "2";                   // but string data type here!
    longint;
    unsigned byte;
};
```

Although the `key` definition can occur anywhere inside the `case` body, the data type you declare for the `key` might depend on its placement. For example, you could take advantage of the data type declaration for `key` to solve alignment problems. In this `case` statement, taken from the `'DITL'` type declaration, the `key` definition follows the first field in the `case` body and serves to define the key value as well as to provide byte alignment.

```
case Button:
        boolean         enabled,disabled;           // enable flag
        key bitstring[7] = 4;
        pstring;                                    // title
```

To prevent DeRez from becoming confused, observe the following restrictions in the placement and declaration of `key` definitions:

- Make all `key` definitions the same size. If you use a string to define a key, all other strings used to define keys within that `switch` type declaration must be the same length.
- Place all `key` definitions at the same offset from the beginning of the `case` statement.

Fill and Align Types

Although resources created by a resource definition always start on an even boundary, no implicit alignment is provided for the data fields defined in the resource. The resource is treated as a bit stream; integers and strings can start at any bit. The `fill` and `align` types described in this section allow you to pad fields to byte, word, or longword boundaries.

You use `align` to pad fields to the boundary you specify; you use `fill` to add a specified number of bits to the data stream. Both `fill` and `align` generate zero-filled fields. DeRez does not supply any values for `fill` or `align` declarations; it just skips the specified number of bits or aligns the data as specified.

The `fill` type causes Rez to add the specified number of bits to the data stream. The bits are always 0. Declare the `fill` type as follows:

`fill` *fill-size* `[ [` *length* `] ];`

fill-size
`bit | nibble | byte | word | long`

The keyword you use to specify *fill-size* declares a fill of 1, 4, 8, 16, or 32 bits (multiplied by *length* if specified). The *length* parameter is an expression whose value can range from 1 through 2,147,483,647.

The following `fill` declarations are equivalent:

```
fill word [2];
fill long;
fill bit [32];
```

The `align` type causes Rez to add fill bits of zero value until the data is aligned at the specified boundary. The syntax of the `align` type is

`align` *align-size* ;

align-size
`nibble | byte |word | long.`

The `align` type pads with zeros until data is aligned on a 4-bit, 8-bit, 16-bit, or 32-bit boundary.

Labels

Labels are used only in `Type` statements and allow you to calculate offsets and to access data at labels. (Labels are used internally to support some of the more complicated resources such as `'NFNT'` and Color QuickDraw resources.)

If your resource includes only fixed-size fields, it is easy to determine the starting address of each field. If your resource includes a variable-size field and that field is not the last field in the resource, calculating the starting address of the next field is much trickier. Placing labels in your resource allows you to calculate the beginning address of a field that follows a variable-size field.

The label declaration takes the following form:

identifier :

The identifier used to specify a label must be terminated with a colon (:). The following are examples of valid label declarations:

```
_StartString:
EndString:
Label_12:
```

Labels are local to each `Type` declaration. More than one label can appear on a data field.

The value of a label is always the offset, in bits, from the beginning of the resource to the position where the label occurs when mapped to the resource data. In this example,

```
type 'cool' {
    cstring;
endOfString:
    integer = endOfString;
};
resource 'cool' (8) {
    "Neato"
}
```

the value of `endOfString`, stored in the integer field following `cstring`, would be calculated as follows:

$$(\underbrace{5}_{\text{length of "Neato"}} + \underbrace{1}_{\text{null terminator}}) * \underbrace{8}_{\text{bits per byte}} = 48$$

Suppose that your program needs to access a field of a resource that follows a variable-size string, for example, the `cstring` declared in the next example:

```
type 'quiz'
        integer;
        longint = EndOfCstring;     // corresponds to QuizRec^^.offset
        cstring;
        align word;
        EndOfCstring:
        byte;                       // corresponds to QuizRec^^.myByte
```

This resource corresponds to a record declared as follows:

```
TYPE
    QuizRec = RECORD
        myInt: Integer;
        offset: Longint;
        myString: String;
        myByte: Char;
    END;
```

You can calculate the address of `mybyte`, following `cstring`, as follows:

@(QuizRec^^.myByte) = QuizRec^ + BitShift (QuizRec^^.offset, –3)

The fields that allow you to calculate the beginning address of the byte declaration are underlined in the resource declaration.

Labels in Expressions

Labels may be used in expressions. In expressions, use only the identifier portion of the label (that is, everything up to, but excluding, the colon). See "Declaring Labels Within Arrays" on page C-49 for more information.

Rez Functions to Access Resource Data

In some cases, it is desirable to access the actual resource data that a label points to. Several Rez functions allow access to that data. These functions are described in “Label Information” on page C-9.

Declaring Labels Within Arrays

Labels declared within arrays may have many values. For every element in the array, there is a corresponding value for each label defined within the array. Use array subscripts to access the individual values of these labels. The subscript values range from 1 to *n*, where *n* is the number of elements in the array. Labels within arrays that are nested in other arrays require multidimensional subscripts. Each level of nesting adds another subscript. The rightmost subscript varies most quickly. Here is an example:

```
type 'test' {
        integer = $$CountOf(array1);
        array array1 {
                integer = $$CountOf(array2);
                array array2 {
mooLabel:                       integer;                    // label
                };
        };
};

resource 'test' (128) {
        {
                {1,2,3},
                {4,5}
        }
};
```

In the above example, the label `mooLabel` takes on these values:

```
    mooLabel[1,1] = 32      $$Word(mooLabel[1,1]) = 1
    mooLabel[1,2] = 48      $$Word(mooLabel[1,2]) = 2
    mooLabel[1,3] = 64      $$Word(mooLabel[1,3]) = 3
    mooLabel[2,1] = 96      $$Word(mooLabel[2,1]) = 4
    mooLabel[2,2] = 112     $$Word(mooLabel[2,2]) = 5
```

The `$$ArrayIndex` function is helpful when using labels within arrays. For additional information, see "Array Information" on page C-7.

Label Limitations

Keep in mind that Rez and DeRez are basically one-pass compilers. This will help you understand some of the limitations of labels.

To decompile a given resource type, the type declaration must not contain any expressions with more than one undefined label. An undefined label is a label that occurs lexically after the expression. To define a label, use it in an expression.

This example demonstrates how expressions can have only one undefined label and how labels can be defined by their use in expressions:

```
Type 'test' {
/* In this expression, start is defined, next is undefined.*/
    start:integer = next - start;

/* In this expression, next is defined because it was used */
/* in a previous expression, but final is undefined.*/
    middle:integer = final -next;
    next:integer;
    final:
};
```

Actually, Rez can compile resource types that have expressions containing more than one undefined label, but `DeRez` cannot decompile those resources and simply generates `Data` statements.

Note
The label specified in the Rez `$$BitField`, `$$Byte`, `$$Word`, and `$$Long` functions must occur lexically before the expressions containing the function; otherwise, an error is generated. ◆

Apple Event Support

Version 3.3 or later of the MPW Shell supports the Apple events summarized in Table D-1.

Table D-1 Apple events supported by the MPW Shell

ID	Class	Effect
'oapp'	'aevt'	The Finder sends the Open Application ('oapp') event when the MPW Shell is launched and no documents are selected to be executed or printed. The reply contains 'errn' == noErr. This event is included because it is a required event.
'odoc'	'aevt'	The Open Documents ('odoc') event contains a list of files that the MPW Shell is to open. The MPW Shell opens these documents as scripts and ToolServer executes them.
'pdoc'	'aevt'	The Print Documents ('pdoc') event causes the MPW Shell to execute the Print tool to print the specified documents.
'quit'	'aevt'	The Quit Application ('quit') event terminates the MPW Shell. If the MPW Shell is executing a tool or script, the Quit Application event is queued until the script or tool completes. As required by the Apple Event Manager, the MPW Shell replies to the event and then proceeds with the quit. Because quitting involves a number of steps, including the execution of a script, it is possible for the quit to fail even though a good status has been returned.
'dosc'	'misc'	The Do Script ('dosc') event sends an MPW command line to be executed by the MPW Shell.
'scpt'	'MPS '	The Script ('scpt') event sends an MPW command line to be executed by the MPW Shell. This event returns a series of events containing the text written to Dev:Output, Dev:Error, or Dev:Console.
'outp'	'MPS '	The Output ('outp') event returns standard output to the client application.
'diag'	'MPS '	The Diagnostic ('diag') event returns diagnostic output (standard error) to the client application.

continued

Table D-1 Apple events supported by the MPW Shell (continued)

ID	Class	Effect
`'stat'`	`'MPS '`	The Status (`'stat'`) event is used by the client application to determine the current status of the MPW Shell.
`'abrt'`	`'MPS '`	The client application uses the Abort (`'abrt'`) event to abort the currently executing script.

Because of the interactive nature of the MPW Shell and the design of the MPW Shell and the editor, this support is somewhat limited compared to the support provided by ToolServer. The differences and limitations are as follows:

- The MPW Shell's reply to an Apple event is merely an acknowledgment that the event was received properly. It does not indicate the success or failure of the request, because the MPW Shell replies to the event before it processes the event.
- Because the MPW Shell replies immediately to script events, the Abort and Status events function only when the transaction ID is not set (specified as `kAnyTransactionID`). The MPW Shell does not match the sender of this event with the sender of the script.
- Because the MPW Shell has a console, it does not support the `Dev:Output` or `Dev:Error` devices or return console output to the sender of a script event.
- The MPW Shell supports the four required events (Open Application, Open Documents, Print Documents, and Quit Application). However, the Open Documents event opens the files in the editor and does not execute a script, as is the case with ToolServer.
- The MPW Shell accepts the Script and Do Script events. However, it replies immediately and then executes the script as if the user had run the script by means of a user-defined menu item. Standard output and standard error default to the console, which is generally the frontmost window. The error number in the reply indicates that the script was successfully received, not the success or failure of the script itself.

The MPW Shell can also communicate with MPW Shells on other machines and with ToolServer by using the `RShell` command that has been added to both the MPW Shell and ToolServer. This command is described in Chapter 3 of *ToolServer Reference* and in the *MPW Command Reference*.

Troubleshooting

This section contains a list of possible problems you may encounter when building Macintosh runtime applications and shared libraries. This list does not cover general programming errors.

Debugging Makefiles

When Make doesn't seem to be doing what you expect, you need to debug your makefile. The following options to the Make command are helpful in debugging makefiles:

1. Use the `-v` option to generate verbose diagnostic output. This output tells you which files don't exist, which files are up-to-date, and which files need rebuilding and why they are out-of-date. It also points out which files don't have build rules and are thus assumed to be abstract targets.
2. Use the `-s` option to show the structure of your target's dependency relations in a dependency-tree graph. This option displays the complete structure of dependencies, including those generated by default rules. A target (or subtarget) that doesn't appear or that has no prerequisites might indicate a typographical error in the dependency line for that target (or in the line for one of the targets that depend on it). A target that appears at the wrong level in the dependency graph indicates an error in your dependency specification.
3. Use the `-u` option to find unreachable targets. These might be parts of your target dependencies that did not get connected to the rest of the dependency hierarchy because of a bad or mistyped rule.

If using the options described so far does not help you debug your makefile, you can also check for the following:

- Build commands that affect files Make assumes are already up-to-date.

  Make generates commands that you must then execute to perform the actual updates. Because Make must determine what build commands to generate before any targets are actually built, the possibility of "phase errors" exists;

that is, unexpected behavior may occur when generated commands alter the assumptions that Make used to determine whether targets were out-of-date. For example, a Shell command deletes a file that Make thinks is up-to-date.

- Different pathname specifications for the same file (that is, pathnames with different degrees of volume and directory qualification). Be sure pathname specifications for the same file are identical.

If you get an error message saying that no rules were available to build a target that should have been covered by a default rule, the error might result from one of these problems:

- The default rule may not have found a match for a filename in the makefile or in the file system and was therefore not applied.
- A filename was mistyped, and the default rule could not be applied. You should be able to detect such errors by inspecting the output of the `-s` and `-v` options.
- A default rule given in the makefile was mistyped. In this case, you might not see any dependencies generated by the rule when you use the `-s` option on the Make command line.

Overriding Entry Points in PowerPC Runtime Programs

As described in "Multiple External Symbol Definitions" on page 8-6, you can "override" functions by simply adding your own object files to the command line prior to the file containing the old definition. For example, if you wanted to create and use your own version of `printf()`, you could link your `myprintf.o` file ahead of the `StdCLib.o`.

However, if you are creating a PowerPC runtime code fragment capable of being "overridden," you must be aware of the following details. First, you should compile all the source for that code fragment with the option `-shared_lib_export on`. This forces the compiler to generate the correct `nop` instructions that the linker edits to become TOC reload instructions.

Second, you must be careful with compiler optimizations. The compiler may generate optimized code that works incorrectly when linked. For example, consider the source file:

```
class mine
{
    void foo(int argc, char* argv[]);
    void bar(int argc, char* argv[]);
};

void mine::bar(int argc, char* argv[]) {}

void mine::foo(int argc, char* argv[])
{
    char local[32] = {'\0'};
    bar(argc, argv);
}
```

Certain compiler optimizations look at the code globally, and they see the call to `mine::bar()` from `mine::foo()` as a call to an empty function. As a result, the compiler does not prepare registers for passing parameters and you cannot override the function `mine::bar()`.

Linking Problems

Most linker problems result from not including the correct object file or library, or otherwise making references to routines or variables that the linker cannot find.

PPCLink Problems

The following are problems you may encounter when using PPCLink:

- PPCLink generates XCOFF files that appear to be bad in the following situation:

  A shared library exports a data item such as `myGlobal`, and a client using the shared library has a global data item with the same name as in the following example:

  ```
  long myGlobal;
  main()
  {
      ...
      myGlobal = 0xDEADBEEF;
      ...
  }
  ```

  The linker reports a duplicate symbol warning and uses the shared library's data item, but there is still a "dangling" relocation that exists in the client. If the XCOFF output format is used for the link and DumpXCOFF is then used to dump the resulting output file, the DumpXCOFF tool reports the following error:

  ```
  ### Error ### r_symndx: invalid (symbol table entry not found)!
  ```

  However, this is a benign problem, since the "bad" relocation is never used except by DumpXCOFF.

- PPCLink may generate warnings like the following:

  ```
  # Warning: Orphan BINCL/EINCL in files mooStuff.o. Will be dead stripped
  ```

  This generally occurs when the linker encounters previously built static libraries compiled with `-sym on | big`.

  To eliminate the warnings, relink your libraries.

ILink Problems

The following situations can result in errors when using ILink.

Using the State File

The state file enables ILink to do incremental linking. If you get the following error message, it probably means you have an older state file (generated by an earlier version of ILink) that is incompatible with the version of ILink you are currently using. If you experience this or any other strange error while doing an incremental link, delete the state file and relink to see if the problem goes away.

```
# Error 7014 while opening statefile “app.NJ”
# Statefile may be damaged or incorrect version. Delete statefile and
relink.
ILink - Execution terminated!
```

Using the 68K Macintosh Debugger

If you are using the 68K Macintosh Debugger to debug an application or shared library linked by ILink, you must close the Browser before relinking. Failure to do so will result in an error message similar to the following:

```
# Error -49 while opening the statefile
# The read/write permission of only one access path to a file can allow
writing (OS error -49)
ILink - Execution terminated!
```

Linking to Imported Data (CFM-68K Only)

You must indicate all imported data items using the `import` pragma in your source code, so the compiler can generate the proper references. Otherwise, ILink reports an error similar to the following:

```
# Illegal reference to "libData"
# Segment 'CODE'(1), module "mooProc"…
# Error: Reference from code to an imported symbol
```

If the compiler properly marked a data item as imported, but the library that exports the data item is not supplied to ILink at link time, ILink reports an error much like the following:

```
# Undefined entry, name: "libData"
# Referenced from "_#libData" in file "ObjFile.o"
```

The item `_#libData` is the XDataPointer to `libData` (which the linker cannot find).

Linking to Imported Functions (CFM-68K Only)

You must use the `import` pragma in your source code to indicate which functions are imported from shared libraries. Not doing so will result in a linker error such as the following:

```
# Undefined entry, name: "_$libFunc"
# Referenced from "_$mooProc" in file "ObjFile.o"
```

Marking a function as imported causes the compiler to generate references to an XVector rather than to the function itself. If ILink cannot find the imported symbol at link time, it reports an error much like the following:

```
# Undefined entry, name: "_%libFunc"
# Referenced from "_@libFunc" (a linker-generated XPointer) which is...
# Referenced from "_$mooProc" in file "ObjFile.o"
```

Other ILink Problems

- If you are linking using `-model near` or `-model far`, and you fail to link with `MacRuntime.o`, the following error results:

  ```
  # Undefined entry, name: "_A5Init3"
  # Could not find data initialization patch entry "_A5Init3"
  ILink - Execution terminated!
  ```

- If you get an error like the following, you probably have a segment larger than 32 KB:

  ```
  # Reference to "DoStuff" out of range
  # Segment 'CODE'(11), module "MyProc"…
  # Error: PC-Relative offset of reference from code does not fit into
  16 bits
  ILink - Execution terminated!
  ```

  Within `MyProc`, there is a 16-bit PC-relative reference to `DoStuff` that is out of range (greater than 32 KB).

  Remedy: Compile and link with the `-model far` option or resegment your code.

- The following usually results when you attempt to perform an incremental link on an application that was linked with the Link tool.

  ```
  # Unfaithful resource file, linked by somebody else
  # Incremental link may yield an unrunnable application
  # If application fails, delete application and try again
  ```

  Remedy: Remove the application and state file and relink.

- Object files compiled prior to MPW 3.0 are not supported. The following error results:

  ```
  # File "OldObj.o": object file is pre-3.0.  Recompile it.
  ILink - Execution terminated!
  ```

  Remedy: If you can't recompile the old object file, try running the Lib tool on it and then relink using the resulting library file. Be sure to remove all symbolic information by specifying the `-sym off` option to Lib.

- ILink does not support the linking of two different object files with the same name. You will see the following message if progress information (`-p`) is enabled:

  ```
  Note: directory id changed for objfile.o
  ```

  In general, this message appears if the object files have been moved since the previous link; in such a case, you should delete your state file and relink. However, this message is also generated if you have two identically named object files in the same link. This situation will cause ILinkToSYM to fail because it cannot distinguish between the two files.

  Remedy: Remove the state file and application, assign unique names to the object files, and relink.

Glossary

32-bit PowerPC architecture The architecture of PowerPC microprocessors that use 32-bit addressing and the full PowerPC instruction set (for example, the MPC604). Compare **64-bit PowerPC architecture, PowerPC 601 architecture.**

64-bit PowerPC architecture The architecture of PowerPC microprocessors that use 64-bit addressing and the full PowerPC instruction set (for example, the MPC620). Compare **32-bit PowerPC architecture, PowerPC 601 architecture.**

68K application An application running under the classic 68K runtime architecture. 68K applications cannot use shared libraries.

68K-based Macintosh computer Any computer containing a 680x0 central processing unit that runs Macintosh system software. See also **PowerPC-based Macintosh computer.**

68K microprocessor Any member of the Motorola 68000 family of microprocessors.

abstract target In a makefile, a target that is not actually built but represents a collection of items. A dependency rule with an abstract target has no build rules, just dependencies; the abstract target serves to trigger the dependencies for its prerequisite files. See also **dependency rule.**

accelerated resource An executable resource consisting of a routine descriptor and PowerPC code that specifically models the behavior of a 68K stand-alone code resource.

active window The frontmost window. The Shell variable `{Active}` contains the name of the current frontmost window.

A5 world In classic 68K and CFM-68K runtime applications, a memory partition that contains the QuickDraw global variables, the application global variables, the application parameters, and the jump table—all of which are accessed through the A5 register. Sometimes called the *global variable world.*

A-line instruction An instruction used to execute Toolbox and Operating System routines. The first word of an A-line instruction is binary 1010 (hexadecimal A). Also known informally as an *A-trap.*

application A program of type `'APPL'` that runs outside of the MPW Shell environment. See also **program.**

application XVector A 12-byte XVector used in the CFM-68K runtime environment. The first two fields contain the address of a function and the value to be placed in A5 when the function executes. Because applications can be segmented, the third field contains information to locate the function within a particular segment. See also **shared library XVector, XVector.**

arc In performance monitoring, the execution of a routine from a specific call site.

A-trap See **A-line instruction.**

author In Projector, with respect to a revision, the name of the person who made a revision.

basic block In performance monitoring, a contiguous section of code that contains no calls or branches and no destinations for a call or branch (except at the endpoints). That is, code in a basic block is executed entirely linearly.

blank A space or a tab character that serves to separate words in the MPW Shell command language.

branch In Projector, an additional sequence of revisions emanating from another revision and running parallel to the main trunk.

bucket In performance monitoring, the chunks into which the code to be measured is divided. The minimum bucket size is 2 bytes.

build rule In the dependency rule of a makefile, the line beginning with a space or a tab, which follows the dependency line and specifies the command or commands to be executed if the target file is out of date with respect to the prerequisite file.

built-in command An editing command, structured command, or other Shell command that is part of the MPW Shell application, as opposed to a tool or script, which is stored in a separate file on the disk.

CFM-68K runtime architecture A 68K Macintosh runtime architecture that uses the Code Fragment Manager. Its handling of fragments and the ability to use shared libraries is analogous to that of the PowerPC runtime architecture, but it differs in a number of details because of system limitations. In particular, it uses segmented application code addressed through a jump table.

checkout directory The directory into which, by default, Projector places checked-out files. Each project has a corresponding checkout directory that can be changed with the `CheckOutDir` command.

'ckid' resource The “check ID” resource that Projector maintains in the resource fork of all files belonging to a project.

classic 68K runtime architecture The runtime architecture that has been used historically for the 68K Macintosh. Its defining characteristics are the A5 world, segmented applications addressed through the jump table, and the application heap for dynamic storage allocation. See also **CFM-68K runtime architecture, PowerPC runtime architecture.**

code bucket See **bucket.**

code fragment See **fragment.**

Code Fragment Manager (CFM) The part of the Macintosh system software that loads fragments into memory and prepares them for execution. There are separate internal components to manage PowerPC code and CFM-68K runtime code, but the APIs to CFM are identical for each type of code. In general, the context determines which version of the Code Fragment Manager is being referred to.

code monitoring See **performance monitoring.**

code resource A resource created by the linker that contains the program's code. Code resources can be of many types—most commonly `'CODE'`, `'MPST'`, or `'DRVR'`.

command alias An alternate name for a command, defined with the `Alias` command.

command file See **script.**

command name The first word of a command, identifying the name of a built-in command or the name of a file (tool, script, or application) to execute.

Commando dialog box A dialog box, defined by a `'cmdo'` resource, that allows users to execute MPW tools interactively rather than by using the command line.

command script See **script.**

command substitution The replacement of a command by its output. You specify that the output of the command replace the command by enclosing the command in backquotes.

console The window where a command is entered and executed (standard input). Also, the window to which the command's output is returned (standard output).

current project The name of the current project. Projector assumes all actions pertain to this project unless you specify a different project name by using the `-project` option.

current selection The currently selected text in a window. In editing commands, the current selection in the target window is represented by the selection character § (Option-6).

data fork One of two forks of a Macintosh file. The data fork can contain text, code, or data, or it can be empty. PowerPC runtime fragments and CFM-68K shared library fragments are stored in the data fork. Compare **resource fork.**

dead code Code contained in modules that cannot be reached by references stemming from the main entry point of a program.

definition version The version of an import library used by the linker to resolve imports in the application (or other fragment) being linked. The definition version defines the external programming interface and data format of the library. Also called *link-time version*. Compare **implementation version.**

dependency file A makefile.

dependency line In a makefile, the first line of a dependency rule, which states the dependency between the target file and the prerequisite file.

dependency rule In a makefile, a rule that specifies the prerequisite files of a target file and the commands needed to build the target file.

dependent control In a Commando dialog box, a control that is enabled or disabled depending on the state of its parent control.

derez To decompile a resource file by using the MPW resource decompiler, DeRez.

desk accessory An application that is implemented either as a device driver (in system software versions prior to System 7) or as a small application in System 7.

diagnostic output Commands and tools send error and progress output to diagnostic output. By default, diagnostic output is sent to the window where the command was executed. You can redirect diagnostic output to another file, window, or selection using the ≥ and ≥≥ operators. Diagnostic output is also referred to as *standard error.*

driver A program that enables applications to communicate with hardware devices. Native PowerPC PCI drivers are implemented as code fragments.

driver reference number The means by which you identify a driver after it has been opened.

drop-in addition A shared library containing code and data that extends the capabilities of an application. Also called an *application extension*. Drop-in additions must be explicitly loaded by the client application before use.

entry point A location (offset) within a module.

escape character A character used to disable the meaning of the character following it, to continue commands over more than one line, and to insert invisible characters into command text. The MPW Shell escape character is ∂ (Option-D).

executable resource Any resource that contains executable code. See also **accelerated resource.**

export A data item or executable routine within a fragment that is made available for use by other fragments.

Extended Common Object File Format (XCOFF) An executable file format generated by some PowerPC compilers. See also **Preferred Executable Format.**

external entry point The entry point to a procedure when called indirectly or from another fragment. Typically this entry point allows inclusion of instructions to set up an A5 world for the called procedure before entering the internal entry point. Compare **internal entry point.**

external reference A reference to a routine or variable defined in a separate compilation unit or assembly.

fat application An application that contains code of two or more runtime architectures. For example, a fat application may contain both CFM-68K and PowerPC runtime code.

fat binary program Any piece of executable code (application, shared library, code resource, trap, or trap patch) that contains code of multiple runtime architectures. See also **fat application, fat library.**

fat shared library A shared library that contains code of two or more runtime architectures. For example, a fat library may contain both CFM-68K and PowerPC versions of a shared library.

filename A sequence of up to 31 printing characters (excluding colons) that identifies a file. See also **pathname.**

file type In MPW, the type of a file, such as `'APPL'`, `'MPST'`, or `'TEXT'`, which determines how the MPW Shell runs that file when you enter the filename as a command.

flat time In performance monitoring, the amount of time spent executing a routine. See also **hierarchical time.**

fragment An executable unit of code and its associated data. A fragment is produced by the linker and loaded for execution by the Code Fragment Manager.

global variable world See **A5 world.**

gtbu A billion ticks of the time base unit (tbu). See **time base unit.**

hierarchical time In performance monitoring, the amount of time spent executing a routine and any routines called directly or indirectly from that routine. See also **flat time.**

host computer The Macintosh computer on which you build a CFM-68K runtime or PowerPC code fragment.

implementation version The version of an import library that is connected at load time to the application (or other fragment) being loaded. The implementation version provides the actual executable code and data exported by the library. Also called *runtime version*. Compare **definition version.**

import A data item or executable routine referenced by a fragment but not contained in it. An import is identified by name to the linker, but its actual address is bound at load time by the Code Fragment Manager.

import library A shared library that is bound by name to a client at link time and is automatically loaded at runtime by the Code Fragment Manager. Import libraries are a subset of shared libraries.

initialization routine A function contained in a fragment that is executed immediately after the fragment is loaded and prepared. Compare **termination routine.**

insertion point An empty selection range; that is, the character position where text will be inserted. This position is marked with a blinking vertical bar.

internal entry point The entry point to a procedure when accessed through a direct call. The internal entry point skips any A5 switching and simply enters the beginning of the actual procedure. Compare **external entry point.**

intersegment reference A reference to a routine in another segment.

intrasegment reference A reference to a routine in the same segment.

jump table A table that contains one entry for every externally referenced routine in an application or MPW tool and provides the means by which segments are loaded and unloaded.

label In Rez `Type` statements, a data type that is used to calculate the offset of data in a resource.

link map A map containing information about either code fragments or code segments (depending on the runtime architecture) and the location of modules within them. Sometimes known as *location maps.*

link-time version See **definition version.**

makefile A text file that serves as the input to the Make command. This file describes the dependencies among a program's source files and the commands that must be executed to bring all target files up to date. The file is named `MakeFile` by default.

module A contiguous region of memory that contains code or static data; the smallest unit of memory that is included or removed by the linker. See also **segment.**

mounted project In Projector, a project that is not nested within another project; similar to the root directory on a volume. You can mount several projects, just as you can mount several volumes.

MPW Shell The application that provides the environment within which the other parts of the Macintosh Programmer's Workshop (MPW) operate.

MPW tool See **tool.**

noncode resource A resource containing the data structures on which the program operates, for example, `'WIND'`, `'DLOG'`, `'DITL'`, or `'SIZE'` resources. You use the resource compiler Rez or a resource editor to create noncode resources.

obsolete file A file (belonging to a project) that has been rendered inactive using the command `ObsoleteProjectorFile`. An obsolete file cannot be checked out for modification. You can reactivate the file using the command `UnobsoleteProjectorFile`.

option A command-line switch, specifying some variation from a command's default behavior. Options always begin with a hyphen (`-`).

orphan file In Projector, a file that belongs to a project, but whose resource fork no longer contains the information that Projector needs to determine to which project it belongs. See also **`'ckid'` resource.**

parameter The words following the keyword in a simple command. MPW commands use two types of parameters: options and filenames. Certain parameters, such as I/O redirection, are interpreted by the MPW Shell and are never seen by the command itself.

parent control In a Commando dialog box, an option or control whose status determines the state of a dependent option or control.

pathname A sequence of up to 255 characters that identifies a file, directory, and volume. A full pathname contains embedded colons but no leading colon. A partial pathname either contains no colons or has a leading colon. A leafname is a partial pathname that contains no colons.

pattern A literal text pattern such as /ABCDEFG/ or a regular expression. Patterns are a case of selection and always appear between pattern delimiters (/.../ or \...\).

PCI Peripheral Component Interconnect. The standard for expansion buses on PowerPC-based Macintosh computers.

PEF See **Preferred Executable Format.**

performance monitoring The measurement of the amount of time your program spends in executing certain routines or instructions.

pipe To cause the output of one command to be used as the input for the subsequent command. The command terminator `|` is the pipe symbol.

POWER architecture The precursor to the PowerPC microprocessor architecture.

PowerPC 601 architecture The architecture of the MPC601 microprocessor. The instruction set of the PowerPC 601 is a hybrid of the POWER instruction set and the 32-bit PowerPC instruction set. Compare **32-bit PowerPC architecture, POWER architecture.**

PowerPC application An application that contains code only for a PowerPC microprocessor. See also **68K application**, **fat application.**

PowerPC-based Macintosh computer Any computer containing a PowerPC CPU that runs Macintosh system software. See also **68K-based Macintosh computer.**

PowerPC microprocessor Any member of the family of PowerPC microprocessors. Members of the PowerPC family include the MPC601, 603, and 604 CPUs.

PowerPC runtime architecture The runtime architecture for Macintosh computers using the PowerPC microprocessor. Its characteristics include storage of code and data in contiguous fragments, the absence of an A5 world, and the ability to use shared libraries.

Preferred Executable Format (PEF) The format of executable files used for PowerPC applications and shared libraries. It is also used for CFM-68K runtime import libraries that have been flattened. CFM-68K runtime applications are stored in a combination of PEF containers and `'CODE'` resources.

prerequisite file In a makefile, the file on which a target file depends. If the prerequisite file is newer than the target file or if it does not exist, the subsequent build command is executed to produce an up-to-date target file.

profiling The dynamic recording, for every routine call, of the identity of the called routine and the point from which it was called. See also **performance monitoring.**

program Code and noncode resources that are written to the resource fork of the program file. See also **code resource, noncode resource.**

project In Projector, a set of files that may include other projects (subprojects).

project directory The directory in which Projector maintains all the project management information about a given project.

project information In Projector, information maintained by Projector on a per-project basis, including author, last modification date, and command.

Projector A collection of built-in MPW commands and windows that help programmers control and account for changes to all the files associated with a software development project.

`ProjectorDB` **file** The database file in which Projector stores all information about projects, their revision trees, revisions, and branches.

project tree In Projector, the set of mounted projects.

pseudodevice See **pseudofilename.**

pseudofilename Any device name that you can use in place of a filename but that has no disk file associated with it; for example, `DEV:NULL` or `DEV:OUT`. Any command can open a pseudofilename. These are most often used for I/O redirection. Also called a *pseudodevice*.

quotes A set of characters used to literalize the characters they enclose. The quote symbols are '...', "...", \...\, and /.../. The escape character, ∂, quotes the character that follows it.

reference The location within one module that contains the address of another module or entry point.

regular expression A language for specifying text patterns, using a special set of metacharacters.

regular expression operators A special set of metacharacters used in regular expressions and filename generation.

resource A data structure used to store a program's data or code. This structure is declared and defined using the Rez language. Resources used to store code are built by the linker; resources used to store data are built by the resource compiler.

resource attributes Values associated with a resource that determine where and when the resource is loaded in memory, whether it can be changed, and whether it can be purged.

resource compiler A program that creates resources from a textual description. The MPW resource compiler is named Rez.

resource definition file A file containing the definitions for types declared in a type declaration file.

resource description file A file containing the textual description of your program's noncode resources.

resource file The resource fork of a file.

resource fork One of two forks of a Macintosh file. It can contain code resources or noncode resources, or it can be empty. 68K-based runtime applications store their code in the resource fork. Compare **data fork.**

resource specification The information used to identify a resource: the resource name, the resource type, and the values of its attributes.

revision In Projector, an instance of a file in a project. A new revision is created each time a modified file is checked in.

revision information Information maintained by Projector for every revision of a file.

revision name In Projector, a name associated with a chosen set of file revisions. Revision names can be private, public, static, or dynamic.

revision tree In Projector, the composite history of a file; that is, all the revisions and branches made to a file.

root In a makefile, a top-level target that is not a prerequisite of any other target. See also **target file.**

RTOC See **Table of Contents Register.**

runtime environment The execution environment provided by the Process Manager and other system software services. The runtime environment dictates how executable code is loaded into memory, where data is stored, and how functions call other functions and system software routines.

runtime version See **implementation version.**

script A text file containing a series of commands. You can execute a script by entering its filename.

segment A named collection of modules.

segment relocation information Part of a segment header used to store information that allows the relocation of intrasegment references for programs compiled and linked using the `-model far` option.

selection A series of characters, or a character position, at which the next editing operation will occur. A selection is used as an argument to most editing commands and can be specified using a special set of selection operators.

shared library A fragment that exports functions and global variables to other fragments. A shared library is not included with the application code at link time but is linked in dynamically at runtime. A shared library is stored in a file of type `'shlb'`. There are two types of shared libraries: import libraries and drop-in additions.

shared library XVector An 8-byte XVector in the CFM-68K runtime environment. Its two fields contain the address of a function and the value to be placed in A5 when the function executes. See also **application XVector, XVector.**

signal An interrupt generated by software rather than hardware. A program running in the MPW Shell can detect two signals: one is Command-period (`SIGINT`); the other is abnormal termination by the `Abort` function (`SIGABRT`).

simple command Any command consisting of a single keyword followed by zero or more parameters.

stand-alone code A type of program used to supplement the standard features provided by the Macintosh Toolbox and Operating System, to execute startup functions, or to control peripherals.

standard error See **diagnostic output.**

standard input Input to a command, usually typed directly into the active window.

standard output Output produced by most commands that is returned to an open file, usually the window in which the command or the script containing it was executed.

`Startup` **file** A special command file that is executed each time the MPW Shell is launched. The Startup file executes a second command file called `UserStartup`.

state file A file created and used by the ILink linker to store information such as link options, a list of object files, and data used to resolve code and data references at runtime.

static library A library whose code is included in the application at link time.

status code A code returned by commands in the Shell variable `{Status}`. Zero indicates successful completion of the previous command; other values usually indicate an error.

status panel The panel in the upper-left corner of the MPW Shell's Worksheet window. The status panel shows what command MPW is executing. Clicking in the status panel is the same as pressing the Enter key.

structured command Any built-in command that controls the order in which other commands are executed. `For` and `If` are examples of structured commands. See also **simple command.**

stub library An import library that exports symbols but does not contain any code. It is often convenient to link against a stub version of a library instead of a full, working version.

subproject In Projector, any project contained within another project. Subprojects may contain other subprojects.

Table of Contents (TOC) An area of static data in a PowerPC fragment that contains pointers to imported data and routines as well as the fragment's own static data. The TOC is similar to the A5 world in a 68K runtime environment.

Table of Contents Register (RTOC) A processor register that points to the Table of Contents of the code fragment currently being executed. On PowerPC-based computers, general-purpose register 2 (GPR2) serves as the RTOC.

target computer The Macintosh computer on which you run your CFM-68K runtime or PowerPC code fragment.

target file In a makefile, a file that is to be built or rebuilt if it has not yet been built or if it is older than its prerequisite file. See also **prerequisite file.**

target window The second window from the front. It is the default target for editing commands that you enter in the active window. The Shell variable `{Target}` contains the name of the current target window.

task In Projector, a short description of the task that a user accomplished with a revision. The `{Task}` variable contains the value of the current task.

tbu See **time base unit.**

termination routine A function contained in a fragment that is executed just before the fragment is unloaded. Compare **initialization routine.**

time base unit (tbu) A unit of time measurement used by the PowerPC microprocessors. The actual physical length of a time base unit (in seconds, for example) is dependent on the speed of the microprocessor.

TOC See **Table of Contents.**

tool A program that runs in the MPW Shell environment and has access to the facilities provided by the MPW Shell.

transition vector In the PowerPC runtime architecture, an 8-byte data structure that describes the entry point and TOC address of a routine. A PowerPC transition vector is identical to the flattened shared library XVector in the CFM-68K runtime environment. See also **shared library XVector, XVector.**

trunk In Projector, the main sequence of revisions to a file. Branches from any revision are named and numbered with respect to the trunk.

type declaration file A file used to declare the format of one or more resource types. Standard type declaration files contain declarations for resource types commonly used by Macintosh programs; for example, `'MENU'`, `'WIND'`, and `'SIZE'`.

type declaration statement A statement written in the Rez language that specifies the pattern for any associated resource data by indicating data type, alignment, size, and placement of strings.

universal procedure pointer A 68K procedure pointer or the address of a routine descriptor.

update library A shared library that contains additions or changes to an existing import library.

user In Projector, the person with access to the files of a project. The name of the user is stored in the `{User}` variable.

weak import A symbol that does not need to be present in any of the client application's import libraries at runtime.

weak library A shared library that does not need to be present at runtime for the client application to run.

Worksheet window The window displayed after MPW has been launched.

XCOFF See **Extended Common Object File Format.**

XDataPointer In the CFM-68K runtime environment, a 4-byte pointer to an imported or exported data item. XDataPointers reside in the near global data area. The XDataPointer for a data item named `libData` has the name `_#libData`.

XPointer In the CFM-68K runtime environment, a 4-byte pointer to an XVector. XPointers reside in the near global data area. The XPointer for an XVector named `_%libFunction` has the name `_@libFunction`.

XVector In the CFM-68K runtime environment, a generalized procedure pointer that contains the entry point address of a function and the value to be placed in the A5 register when the function executes. XVectors reside above the jump table. The Code Fragment Manager sets an XVector's contents at load time. The XVector for a routine named `libFunction` has the name `_%libFunction`. See also **shared library XVector, application XVector.**

Index

Symbols

Numerals

A

B

C

J, K

L

N

O

P

S

T

U

V

W

X

Y, Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Pro printer. Final pages were created on a Docutek. Line art was created using Adobe™ Illustrator and Adobe Photoshop. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITERS
Jun Suzuki and Joanna A. Bujes

DEVELOPMENTAL EDITORS
Laurel Rezeau, Sanborn Hodgkins, and George Truett

ILLUSTRATORS
Sandee Karr, Bruce Lee, and Dave Olmos

PRODUCTION EDITOR
Alexandra Solinski

Special thanks to Cheryl Ewy, Wayne Flansburg, Fred Forsman, Scott Fraser, Alex McKale, Munch Paulson, Brian Strull, and Karen Wenzel.

Acknowledgments to all the Core Tools and DTS reviewers.