APPLE
PROGRAMMER'S
AND DEVELOPER'S
ASSOCIATION

APPLE IIGS Programmers Workshop

Version 1.0

APDA# K2SAW1

Apple IIGS Programmer's Workshop Reference

APDA Draft
July 27, 1987

Apple Technical Publications

This document does not include:

- *final editorial corrections*
- *final art work*
- *an index*

APPLE COMPUTER, INC.

20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010

Apple, the Apple logo, and Macintosh are registered trademarks of Apple Computer, Inc.

Apple IIGS, Apple DeskTop Bus, and SANE are trademarks of Apple Computer, Inc.

Simultaneously published in the United States and Canada.

Contents

List of Figures

List of Tables

Preface

The Apple® IIGS Programmer's Workshop (APW) is Apple Computer's development environment for the Apple IIGS™ computer. APW is a set of programs that enable developers to create application programs on the Apple IIGS. This manual includes information about the APW Shell, Editor, Linker, and utility programs; these are the parts of the workshop that all developers need, regardless of which programming language they use. It also provides the information necessary to write an APW utility or a language compiler or assembler for APW.

In addition to the APW programs described in this book, the Apple IIGS Programmer's Workshop includes several programming languages, such as 65816 assembly language and APW C. Each compiler or assembler is described in a separate manual since each language can be added to your system independently.

This manual is intended for experienced programmers and developers; that is, it assumes that you are familiar with either assembly language or a high-level programming language such as C or Pascal. It assumes that you are familiar with the Apple IIGS computer and the Apple IIGS operating system. See the following section, "Roadmap to the Apple IIGS Technical Manual Suite," for a guide to other technical reference books on the Apple IIGS computer.

This Preface is your guide to the use of this book and the Apple IIGS technical manual suite. The contents of the Preface are as follows:

- A roadmap to the Apple IIGS technical manual suite. The roadmap includes a table that shows the other books in the suite and a figure that shows their interrelationships.
- A guide to the use of this manual, including a brief summary of the contents of each chapter and a suggested sequence in which to use the book.
- An explanation of the typographical conventions used in this book to delineate different kinds of information or to direct your attention to important facts.
- A guide to the other books in the technical manual suite, including brief descriptions of the other books in the suite and an indication of the circumstances under which each book would be helpful to you.

Roadmap to the Apple IIGS Technical Manual Suite

The Apple IIGS personal computer has many advanced features, making it more complex than earlier models of the Apple II. To describe it fully, Apple has produced a suite of technical manuals. Depending on the way you intend to use the Apple IIGS, you may need to refer only to a select few of these manuals, or you may need to refer to most of them.

The technical manuals are listed in Table P-1. Figure P-1 is a diagram showing the relationships among the different manuals. To help you decide which of these manuals you will need to develop a particular program for the Apple IIGS, the contents of each of these manuals is briefly described in the section "Other Materials You'll Need" in this preface.

Table P-1. The Apple IIGS Technical Manuals

Title	Subject
Technical Introduction to the Apple IIGS	What the Apple IIGS is
Apple IIGS Hardware Reference	Machine internals—hardware
Apple IIGS Firmware Reference	Machine internals—firmware
Programmer's Introduction to the Apple IIGS	Concepts and a sample program
Apple IIGS Toolbox Reference: Volume 1	How the tools work and some toolbox specifications
Apple IIGS Toolbox Reference: Volume 2	More toolbox specifications
Apple IIGS Programmer's Workshop Reference	**This book: the development environment**
Apple IIGS Programmer's Workshop Assembler Reference	Using the APW assembler
Apple IIGS Programmer's Workshop C Reference	Using C on the Apple IIGS
ProDOS 8 Technical Reference Manual	Standard Apple II operating system
Apple IIGS ProDOS 16 Reference	Apple IIGS operating system and loader
Human Interface Guidelines	Guidelines for the desktop interface
Apple Numerics Manual	Numerics for all Apple computers

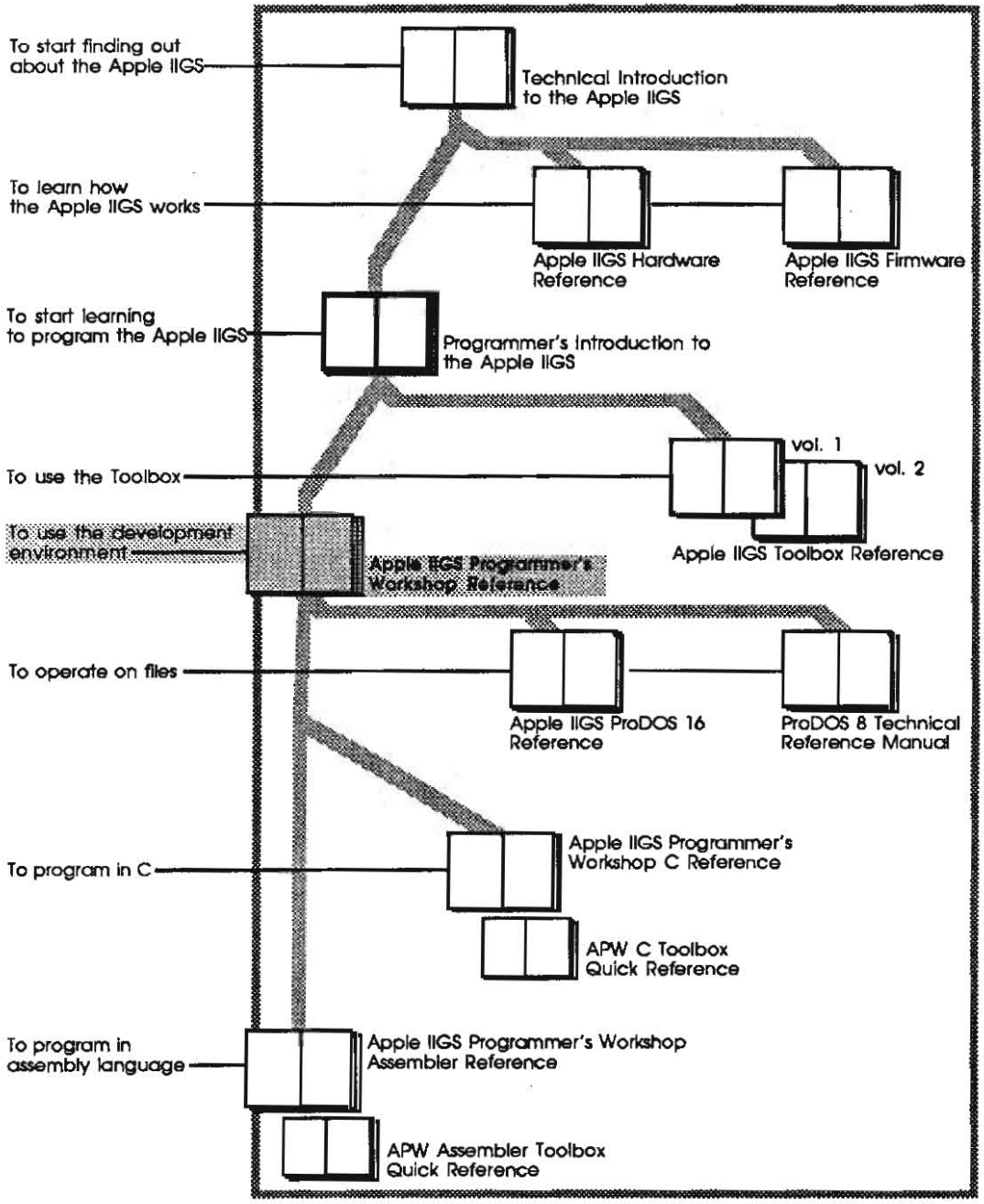


Figure P-1. Roadmap to the Technical Manuals

How to Use This Book

This section describes the contents of the *Apple IIGS Programmer's Workshop Reference* manual. Following a brief chapter-by-chapter description of this book's contents, the section "What to Read When" gives guidelines as to which sections you should read for a specific purpose. Finally, a section entitled "Visual Cues" describes the cues used in this

book to alert you to important material or to words that have special significance (for example, APW commands).

What This Manual Contains

This manual is divided into three parts: an introduction to APW containing three chapters; a four-chapter reference section describing APW programs and commands; and a three-chapter reference section to the internal workings of APW for those who want to add a program to APW. In addition, the book contains three appendixes, a glossary, and an index. Here is a brief description of each of these components:

Part I, "Getting Started," gives you the minimum information you need to get started using the Apple IIGS Programmer's Workshop.

Chapter 1, "About the Apple IIGS Programmer's Workshop," provides a general overview of APW. It defines concepts that are essential to an understanding of the APW environment and gives brief descriptions of the programs that comprise APW.

Chapter 2, "How to Use the Shell and Editor," introduces you to the abilities of APW. This chapter briefly describes how you use APW to write, compile or assemble, link, and run a program.

Part II, "Reference," provides full descriptions of APW commands, the editor, the linker, and the utility programs that are supplied with the APW system.

Chapter 3, "Shell," includes complete descriptions of every APW Shell command, along with descriptions of some APW features too advanced to be covered in Chapter 2.

Chapter 4, "Editor," includes complete descriptions of every APW Editor feature and command.

Chapter 5, "Linker," is a complete reference to the APW Linker. This chapter includes descriptions of every linker feature and function.

Part III, "Inside the Apple IIGS Programmer's Workshop," contains reference material of use to those programmers who wish to add a utility program or language compiler, assembler, or interpreter to APW. This part includes descriptions of Apple IIGS file formats and calls to internal APW functions.

Chapter 6, "Adding a Program to APW," describes the requirements that a utility or language compiler must satisfy to run under APW.

Chapter 7, "File Formats," defines and describes the standard formats for text files and object files for the Apple IIGS computer.

Chapter 8, "Shell Calls," describes several internal APW functions that utilities and compilers can (or must, in some cases) call when operating under APW, including the procedure for calling these functions from assembly language.

The appendixes summarize material for quick reference.

Appendix A, "Contents of the APW Disks," contains a list of all the files delivered on a set of APW disks.

Appendix B, "Command Summary" is a complete list, with brief descriptions, of all the shell, editor, and linker commands. This appendix also lists all the language numbers assigned so far for APW languages.

Appendix C, "Error Messages," discusses errors you can get while running the APW Shell, lists all of the error messages you can get while running the APW Linker, and briefly describes the probable cause of each type of linker error.

The Glossary defines many of the technical terms used in this book.

What to Read When

This manual provides a complete reference to the Apple IIGS Programmer's Workshop. It is not necessary for you to read the entire manual before you start using APW; in fact, depending on the kind of programming you're doing, there may be chapters or sections of chapters that you'll never have to read at all. This section makes some suggestions on how to get the most out of this manual based on your experience and needs.

First, some suggestions for everyone using APW:

- Whatever your background and experience, start with the "Other Materials You'll Need" section of this preface and all of Chapter 1. The Apple IIGS is not quite like any other computer, so even if you had helped design the Apple IIe and Macintosh, you would still have to become familiar with the peculiarities of the Apple IIGS before proceeding.
- Next, read Chapter 2. The first few sections tell you how to set up APW to run on your Apple IIGS and describe the use of the APW Shell command interpreter. The last few sections of Chapter 2 give you enough commands and instructions to get started using APW.
- Look through Appendix A to get an idea of what commands and features are available in the shell and editor.

Note: Although some APW commands are the same as Macintosh Programmer's Workshop (MPW) commands, many are entirely different. Not all MPW commands have APW equivalents.

You are now ready to begin using APW. Keep in mind that APW has many features, so even if you have used a command before, it might have options with which you are not familiar. Read the relevant sections of Chapters 4 and 5 to learn about shell and editor commands when you need to use them.

The following suggestions apply only to those with special needs:

- If you are writing large or complex programs, you will probably want to take advantage of some of the linker's advanced features. Look through Chapter 5 to get an idea of what the linker can do *before* writing your program; you can then read about the commands and options you need when you need them. For short or simple programs, you will never need this material.

- If you are writing a program to run under APW (a utility program or a compiler, for exmple), read Part III. You might find the section on the object module format interesting if you just want to find out more about how the Apple IIGS works. Unless you are actually adding programs to APW, however, these three chapters are not required reading.

Visual Cues

Look for these visual cues throughout the manual:

Note: Notes like this contain sidelights or information that you will probably find useful.

Important: "Important boxes" like this contain information that you should read before proceding.

Warning: A warning directs your attention to something that could cause loss of data or damage to the software.

Boldfaced terms are defined in the Glossary.

A special typeface is used for characters that you type or that can appear on the screen, such as commands, assembly-language instructions and directives, filenames, or system prompts:

```
It looks like this.
```

Icons are used in tables and command-input lines to indicate the arrow keys and the Apple key. If you must press two keys simultaneously, they are shown with a hyphen (-) between them. For example, the following sequence indicates you must press the Control and Y keys simultaneously, followed by the Up Arrow key:

Control-Y ↑

Apple IIe Upgrade: The Apple IIGS Apple key, indicated with the apple icon (), corresponds to the Open Apple key on the Apple IIe keyboard. The Apple IIGS Option key corresponds to the Closed Apple key () on the Apple IIe keyboard. The Clear and Enter keys on the Apple IIGS keyboard have no Apple IIe equivalents.

Important: On the Apple IIGS keyboard, the Reset key has a triangle on it rather than the word *reset*.

Italics are used in commands to indicate parameters that must be replaced with a value. For example, the word *pathname* in the following command refers to any valid ProDOS pathname:

`DELETE` *pathname*

If the file you want to delete is `/APW/MYPROGS/DONOTHING`, this command would be as follows:

```
DELETE /APW/MYPROGS/DONOTHING
```

Other Materials You'll Need

The manuals and software you need in order to develop applications that run on the Apple IIGS depend on the type of programming you are doing. For starters, you must be familiar with use of the Apple IIGS computer, including the control panel. The *Apple IIGS Owner's Guide* that came with your Apple IIGS describes routine operation of the computer.

The following sections describe the manuals in the Apple IIGS technical manual suite (other than this manual) and make recommendations about which manuals you may need based on the type of programming you are doing.

Introductory Manuals

These books are introductory manuals for developers, computer enthusiasts, and other Apple IIGS owners who need technical information. As introductory manuals, their purpose is to help you understand the features of the Apple IIGS, particularly the features that are different from other Apple computers. Having read the introductory manuals, you should refer to specific reference manuals for details about a particular aspect of the Apple IIGS.

The Technical Introduction

The *Technical Introduction to the Apple IIGS* is the first book in the suite of technical manuals about the Apple IIGS. It describes all aspects of the Apple IIGS, including its features and general design, the program environments, the Toolbox, and the development environment.

You should read this book no matter what kind of programming you intend to do, because it will help you understand the powers and limitations of the machine. If you are going to be doing assembly-language or system programming, this book is essential.

The Programmer's Introduction

When you start writing programs that use the Apple IIGS user interface (with windows, menus, and the mouse), the *Programmer's Introduction to the Apple IIGS* provides the concepts and guidelines you need. It is not a complete course in programming; rather, it is a starting point for programmers writing applications that use the Apple Desktop Interface (with windows, menus, and the mouse). It introduces the routines in the Apple IIGS Toolbox and the program environment they run under. It includes a simple **event-driven program** that demonstrates how a program uses the Toolbox and the operating system.

If you are already familiar with writing event-driven programs on the Macintosh, you can probably skim large portions of this manual. If you have never written an event-driven program, or never used the Macintosh tool sets, this manual could save you hours or days of struggling to get started.

Machine Reference Manuals

There are two reference manuals for the machine itself: the *Apple IIGS Hardware Reference* and the *Apple IIGS Firmware Reference* . These books contain detailed specifications for people who want to know exactly what's inside the machine. You don't need to read these manuals to be able to develop applications for the Apple IIGS, especially if you are using a high-level programming language such as C. These books are essential reading if you are doing system programming or writing programs that are designed to recognize whether they are running on the Apple IIGS or on an older Apple II computer. In any case, these books will give you a better understanding of the machine's features. They also explain the reasons why some of those features work the way they do.

The Hardware Reference Manual

The *Apple IIGS Hardware Reference* is required reading for hardware developers, and it will also be of interest to anyone who wants to know how the machine works. It includes the mechanical and electrical specifications of all connectors, both external and internal, and descriptions of the internal hardware.

The Firmware Reference Manual

The *Apple IIGS Firmware Reference* describes the programs and subroutines that are stored in the machine's read-only memory (ROM), with two significant exceptions: Applesoft BASIC and the Toolbox, which have their own manuals. The *Apple IIGS Firmware Reference* includes information about interrupt routines and low-level I/O subroutines for the serial ports, the disk port, and for the Apple DeskTop Bus™, which controls the keyboard and the mouse. The *Apple IIGS Firmware Reference* also describes the Monitor, a low-level programming and debugging aid for assembly-language programs.

The Toolbox Manuals

Like the Macintosh, the Apple IIGS has a set of built-in routines, known as the *Apple IIGS Toolbox*, that can be called by applications to perform many commonly needed functions. For example, there are Apple IIGS tools that you can use to draw things on the screen and tools for controlling desktop windows and menus. The toolbox serves two purposes: it makes developing new applications easier, and it supports the desktop user interface. Tools can be called from any of the Apple IIGS Programmer's Workshop languages.

The *Apple IIGS Toolbox Reference, Volume 1*, introduces concepts and terminology and tells how to use some of the tools. It also tells how to write and install your own tool set. The *Apple IIGS Toolbox Reference, Volume 2*, contains information about the rest of the tools.

You do not need to use the toolbox to write simple programs that do not use the mouse, windows, menus, or other parts of the Apple IIGS desktop user interface. For example, if all the programming you intend to do is to write short routines in C to solve mathematical problems, then you don't need the toolbox at all. If you want to use the Apple IIGS Super Hi-Res graphics display, however, or to develop an application that uses the Apple IIGS desktop and mouse, you'll find the Apple IIGS Toolbox to be indispensable.

The Apple IIGS Programming Language Manuals

The Apple IIGS does not restrict developers to a single programming language. Apple is currently providing a 65816 assembler and a C compiler. Other compilers can be used with the workshop, provided that they observe the standards defined in Chapter 6 of this book, "Adding a Program to APW." You can write different parts of a program in different APW languages and link them into a single load file using the Apple IIGS Progammer's Workshop.

There is a separate reference manual for each programming language that can be used on the Apple IIGS. Each manual includes the specifications of the language, the Apple IIGS libraries for the language, and any special compiler options for that language. The manuals for the languages Apple provides are the *Apple IIGS Programmer's Workshop Assembler Reference* and the *Apple IIGS Programmer's Workshop C Reference.*

The Operating System Manuals

There are two operating systems that run on the Apple IIGS: ProDOS® 16 and ProDOS 8. Each operating system is described in its own manual: *Apple IIGS ProDOS 16 Reference* and *ProDOS 8 Technical Reference Manual.* ProDOS 16 uses the full power of the Apple IIGS and is not compatible with earlier models of the Apple II. The *ProDOS 16 Reference* manual describes the features of ProDOS 16 and also includes information about the System Loader, which works closely with ProDOS 16 to load programs into memory. If you are writing a program that does any file manipulation or that writes to or reads from disk, you must have the *ProDOS 16 Reference* manual. It is a rare applications programmer who will not need this book at some time; for system programmers, it is essential.

ProDOS 8, previously called *ProDOS*, is the standard operating system for most Apple II computers with 8-bit CPUs. ProDOS 8 also runs on the Apple IIGS, but it cannot access certain advanced Apple IIGS features. You need the *ProDOS 8 Technical Reference* only if you are writing programs that will be able to run on 8-bit Apple II's.

All-Apple Manuals

In addition to the Apple IIGS manuals mentioned above, there are two manuals that apply to all Apple computers: *Human Interface Guidelines: The Apple Desktop Interface* and *Apple Numerics Manual.* The *Human Interface Guidelines* manual describes Apple's standards for the desktop interface of any program that runs on Apple computers. If you are writing a commercial application for the Apple IIGS, you should be familiar with the contents of this manual. The people who buy your program will expect it to work like the other programs on their computer; they will be upset if it doesn't.

The *Apple Numerics Manual* is the reference for the Standard Apple Numeric Environment (SANE™), a full implementation of the *IEEE Standard for Binary Floating-Point Arithmetic* (IEEE Std 754-1985). The functions of the Apple IIGS SANE tool set match those of the Macintosh SANE package and of the 6502 assembly language SANE software. If your application requires accurate arithmetic, you'll probably want to use the SANE routines in the Apple IIGS. Whereas the *Apple IIGS Toolbox Reference, Volume II,* tells how to use the SANE routines in your programs, the *Apple Numerics Manual* is the comprehensive reference for the SANE numerics routines. A description of the version of

the SANE routines for the 65C816 is available through the Apple Programmer's and Developer's Association (APDA), administered by the A.P.P.L.E. cooperative in Renton, Washington.

Note: The address of the Apple Programmer's and Developer's Association is 290 SW 43rd Street, Renton, WA 98055, and the telephone number is (206) 251-6548.

Part I
Getting Started

Chapter 1

About the Programmer's Workshop

The Apple IIGS Programmer's Workshop (APW) is a development environment for the Apple IIGS computer. It includes the following components:

- shell
- editor
- linker
- utility programs
- 65816 assembler
- C compiler

In order to understand the operation of these programs, you should be familiar with three other programs:

- ProDOS 16
- Apple IIGS System Loader
- Apple IIGS Memory Manager

Further support for developers is provided by the Apple IIGS Toolbox. The Apple IIGS tools consist of a variety of routines in ROM and RAM that your program can call to perform such functions as I/O control, console control, graphics generation, and mathematical computation. These tools can be used by programs written in the APW environment, but they are *not* considered to be part of the Apple IIGS Programmer's Workshop.

The Apple IIGS Programmer's Workshop, then, consists of 1) several programs that can be used by developers working with any of a variety of programming languages and 2) several programming languages. This manual describes the APW Shell, Editor, Linker, and utility programs; these are the parts of the workshop that all developers need, regardless of which programming language they use. The APW programming languages are described in separate manuals.

This chapter begins with a description of each of the programs in APW, plus ProDOS 16, the System Loader, and the Memory Manager. The next several sections briefly describe a variety of concepts that you must understand in order to program for the Apple IIGS computer.

Specific examples of programs written using APW are given in Chapters 2 and 3. See the *Programmer's Guide to the Apple IIGS* for a more thorough discussion of Apple IIGS concepts and for more extensive programming examples.

Program Descriptions

This section describes each of the programs that make up the Apple IIGS Programmer's Workshop, plus ProDOS 16, the System Loader, and the Memory Manager. Some of the terms used in this section may not be familiar to you; these terms are explained more fully in the next section, "Apple IIGS Concepts," and in the glossary at the end of this book.

Figure 1.1 illustrates the relationships between the Apple IIGS hardware and firmware, the Apple IIGS operating system, and APW. The operating system, including ProDOS, the System Loader, and the Memory Manager, provides the interface between APW and Apple IIGS hardware and firmware. The APW Shell allows you to call the other programs that constitute the Apple IIGS Programmer's Workshop and serves as the link between the APW programs and the Apple IIGS operating system. The APW Shell's command interpreter serves as the interface between you and the rest of the Apple IIGS system.

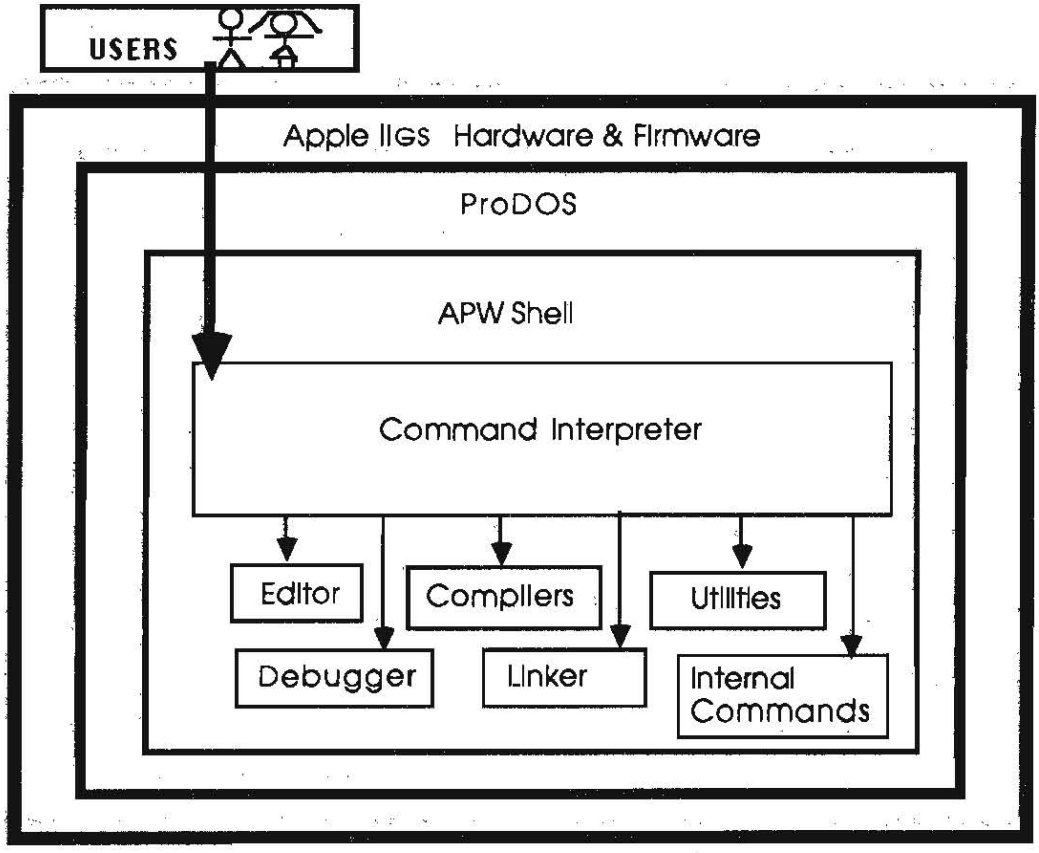


Figure 1.1. The Relationship of APW Programs to the Apple IIGS System

Shell

The shell program provides the interface that allows you to execute the desired APW command or program. It allows you to perform a variety of housekeeping functions, such as copying and deleting files, or listing a directory. The shell supports input and output redirection, as well as **pipelining** of Programmer's Workshop programs.

The shell also acts as an interface and extension to ProDOS 16, providing several functions, called **shell calls**, that can be called by programs running under the shell. Shell calls can be used by utility programs, compilers, linkers, or assemblers to perform such functions as passing parameters and operations flags between the shell and other Programmer's Workshop programs. The format for making these calls is exactly like that for making a ProDOS 16 call.

Editor

This full-screen text editor is designed for use with APW assemblers and compilers. The editor lets you enter text or source code and provides a large number of editing features, including the ability to copy, move, and delete blocks of text, search for text strings, automatically replace text strings with other text, and move quickly from one part of the file to another. Use the shell `EDIT` command to call the APW Editor.

Assembler

This full-featured assembler allows you to write 65816 assembly-language programs for the Apple IIGS computer. The Apple IIGS Programmer's Workshop Assembler includes **macros** to facilitate assembly-language programming and allows you to write your own macros and library files.

The APW Shell commands for assembling a 65816 assembly-language program are described in Chapters 2 and 4 of this manual. The APW Assembler is described in the *Apple IIGS Programmer's Workshop Assembler Reference* manual.

C Compiler

The Apple IIGS Programmer's Workshop C compiler is a complete implementation of the C programming language. It consists of a C compiler, the Standard C Library, the Apple IIGS Interface Libraries, and the C SANE Library. The object files output by the C compiler consist of relocatable code and are fully compatible with those output by the APW Assembler.

The APW Shell commands for compiling a C program are described in Chapters 2 and 4 of this manual. APW C and C compiler options are described in the *Apple IIGS Programmer's Workshop C Reference* manual.

Linker

The APW Linker takes the files (called **object files**) that have been created by the APW Assembler or any of the APW compilers and generates files that the System Loader can load into memory (**load files**). The linker **resolves external references** and creates **relocation dictionaries**, which allow the System Loader to relocate code at load time.

Although the APW Linker is a single program, conceptually there are two APW linkers:

1. Normally, the linker is called directly by a shell command (such as the `ASML` command, which assembles and links a program, or the `LINK` command, which

links object files). These commands provide a limited number of linker options on the command line; most linker options are set to default values. In this manual, this aspect of the linker is referred to as the *standard linker*.

2. Alternatively, all functions of the APW Linker can be controlled by compiling a file of linker commands. The linker command language, called **LinkEd**, allows you to do such things as place specific **object-file segments** in specific **load-file segments**, create **dynamic** load segments, set load addresses for nonrelocatable code, search libraries, and control the output printed by the linker. (Object-file segments, load-file segments, and dynamic and static segments are discussed in the section "Apple IIGS Concepts" later in this chapter.) You can append the LinkEd commands to the last file of your source code, or you can compile and execute them separately by using the `ASSEMBLE`, `COMPILE`, or `ALINK` commands of the Apple IIGS shell. In this manual, the aspect of the linker controlled by LinkEd files is referred to as the *advanced linker*.

The advanced linker is provided for programmers who require maximum flexibility from the system; for most purposes, the standard linker is completely adequate. When a statement in this book applies equally to the standard and advanced aspects of the APW Linker, the terms *APW Linker* or *linker* are used.

Since all Apple IIGS Programmer's Workshop assemblers and compilers create object code that conforms to the same format, called **object module format** (OMF), the APW Linker can link together object files written in any combination of the development-environment languages. Object module format is defined in Chapter 7, "File Formats."

> **Note:** There are currently two versions of the OMF in use for load files: Version 1.0 load files are created by the APW Linker. Version 2.0 load files are created from Version 1.0 load files by the Compact utility program. Both Version 1.0 and Version 2.0 load files can be loaded by the System Loader. See the description of the `COMPACT` command in Chapter 3 for more information on the Compact utility.

Utility Programs

The Apple IIGS Programmer's Workshop includes several programs, called APW **Utilities**, that perform functions not built in to the shell. Utilities include

- Canon, which replaces character strings in a file with other strings as specified in a dictionary file.
- Compact, which converts a load file to a more compact form.
- Crunch, which combines multiple object files created by partial assemblies or compiles into a single object file.
- DumpObj, which lists an object-module-format file to standard output (usually the screen).
- Equal, which compares two files or directories for equality of their contents, dates, and file types.
- Files, which lists the contents of a directory, including subdirectories. Files can also search for a file whose name contains a string you specify.
- Init, which initializes (formats) a disk.
- MacGen, which generates a custom macro file for your program.

- MakeBin, which creates a ProDOS 8 binary file from a ProDOS 16 load file.
- MakeLib, which creates a library file from object files.
- Search, which searches a text or source file for a string that you specify.
- Version, which displays the version number of the APW Shell that you are using.

Most utilities, referred to as *external commands*, are executed like built-in (internal) shell commands. A few utility programs might require more complex command sequences. All of the utility programs supplied with APW are described in Chapter 3. If you add a utility to your system, refer to the documentation that came with it for instructions.

Apple IIGS Debugger

To facilitate the debugging of assembly-language programs, Apple provides the Apple IIGS Debugger, which works with 65816 machine code. The Apple IIGS Debugger allows you to trace or step through a program one instruction at a time or to execute the program at full speed; in either case, you can insert breakpoints at which the debugger halts execution so that you can inspect the contents of the registers, memory, **direct page**, and **stack**. The debugger can display a variety of types of information on the screen, including a disassembly of the code being traced, the contents of memory, the normal display of the program being tested, the contents of the program's direct page, the contents of Apple IIGS registers, and the contents of the program's stack.

The debugger allows you to switch between your test program's screen display and the debugger's displays. If you switch to the debugger's display, the debugger remembers which display mode the test program was in and changes back to that mode when you switch back to the program's display.

The Apple IIGS Debugger is available from A.P.D.A. as a separate product and is described in the *Apple IIGS Debugger Reference*.

ProDOS 16

ProDOS 16 is the central part of the Apple IIGS operating system. Although other software components, such as the System Loader, may be thought of as parts of the overall operating system, ProDOS 16 is the key component. It manages the creation and modification of files, accesses the disk devices on which the files are stored, dispatches interrupt signals to interrupt handlers, and controls certain aspects of the Apple IIGS operating environment, such as pathname prefixes. Your program can call ProDOS 16 to open and close files, read data from disks, write data to disks, and perform a variety of other system functions. Most programs use the ProDOS 16 `QUIT` call to quit and pass control to another program.

ProDOS 16 is described in the *Apple IIGS ProDOS 16 Reference*.

System Loader

The System Loader is an Apple IIGS tool set that reads the files generated by the APW Linker, relocates them (if necessary), and loads them into memory. The System Loader

calls the Memory Manager as necessary to allocate blocks of memory for segments it wants to load.

Each load segment consists of two parts: a set of records that contain all of the code and data in the segment that is not location dependent (with spaces reserved for location-dependent addresses), and a relocation dictionary that provides the information necessary to patch addresses into the first part of the segment at load time. When the segment is loaded into memory, the first part is loaded very quickly; then the relocation dictionary is processed. This structure permits extremely fast loading of relocatable segments.

The System Loader is described in the *Apple IIGS ProDOS 16 Reference.*

Memory Manager

This Apple IIGS program allocates and frees blocks of memory as they are needed. It does the bookkeeping to keep track of which **blocks** of memory are being used and which program owns each block of memory. The System Loader calls the Memory Manager to reserve or release memory when loading segments; your application should also call the Memory Manager whenever it needs a block of memory. Use of the Memory Manager and the System Loader makes it possible for your application to be loaded at the same time as shell programs, memory-resident utilities, character-font data files, and so on.

The Memory Manager is described in the *Apple IIGS Toolbox Reference: Volume I.*

Apple IIGS Concepts

This section introduces a variety of features and concepts that you must understand in order to write application programs for the Apple IIGS computer. While some of these concepts may be familiar to you from work with other computers, you must still be familiar with the way in which they are implemented on the Apple IIGS to get the most out of the Apple IIGS Programmer's Workshop and to use the operating system and the memory of the Apple IIGS efficiently.

Source, Object, and Load Files

There are three main steps to developing a program in the APW environment, and each step corresponds to one of three fundamental types of files: 1) writing the program creates **source files**; 2) compiling or assembling the program creates **object files**; and 3) linking the program creates **load files**. Source files are ASCII files consisting of code and data; each source file follows the conventions of a particular programming language. Object files are binary files containing machine-language instructions rather than the directives and instructions of a higher-level programming language. There can be several object files for one program, each file containing part of the program. Object files do not contain the information needed by the System Loader to load the program into memory. Load files, on the other hand, *can* be loaded by the System Loader. The linker combines ("links") the object files into a single load file and adds the information needed by the loader to load the program into memory.

Symbolic References and Relocatable Code

A source file consists of programming-language instructions, directives, functions, and so forth, together with data needed by the program. In the source code, specific instructions, subroutines, or blocks of data are often labelled with a name. You can refer to the name in another part of the program; for example, when you want to execute a subroutine, you generally refer to the subroutine by name. A name or label of code or data used in this way is referred to as a **symbolic reference** (that is, a *symbol* that can be *referenced* or referred to). In high-level programming languages, symbolic references are often the only means available to jump from one place in a program to another; a few languages, such as BASIC, may also use source-code line numbers, which are relative to the start of the program. These line numbers also serve as symbolic references, since they are not part of the programming language, but only serve to label locations in the program.

In assembly language, in contrast, it is possible to specify actual locations in the computer's memory to which you want the program to jump. For such a program to run, the correct information—that is, the machine instruction that you want to be executed next or the data you want to use—must be present at that location in memory when the jump is made. Code whose location in memory is specified when the program is written or linked is called **absolute**, since the loader must load it at that location or not at all.

Alternatively, it is possible to write a program in which every reference to a location in the program is either relative to another location or is made through a symbolic reference. Such a program need not be loaded into a specific location in memory to run and is thus referred to as **relocatable**. Note that this term is somewhat misleading: a relocatable program can be loaded into any location in memory, but it cannot necessarily be moved once it has been loaded. (A program or block of code that can be moved from one location in memory to another while the program is running is called **movable**.) The term **relocate** in this context means the process of inserting (or **patching**) into the program in memory the actual memory addresses to which jumps must be made. Relocation on the Apple IIGS is done during program load by the System Loader.

Relocatable Load Files

The advantages of using relocatable code for the Apple IIGS are considerable. Relocatable code can be placed in memory at whatever location the Memory Manager chooses. Since desk accessories, shell programs, RAM-based tools, and so forth are placed in memory by the System Loader and Memory Manager, absolute code is likely to conflict with other code already in memory. The Apple IIGS System Loader, object module format, and Memory Manager are designed to support relocatable code. Apple IIGS Programmer's Workshop compilers generate relocatable code, and the APW Assembler is designed to work with relocatable code. Do not write absolute code unless you want to cause untold grief to yourself and the people who use your program.

When relocatable code is assembled or compiled, the assembler or compiler converts the source code into 65816 machine-language instructions, data, and symbolic references. Before the program is actually run, the symbolic references must be **resolved**; that is, the routine being referenced must be found, and the reference must be replaced with code that the loader can use to relocate the code at load time. The program that resolves the symbolic references is called the APW **Linker**. (The linker gets its name from the fact that it can combine, or link together, several object files and library files to create a single executable load file.)

The Three Steps to Program Development

As mentioned above, the conversion of a source file into data that is resident in memory is done in three main steps. Figure 1.2 illustrates these steps. The following is a more detailed account of this sequence:

1. The source code is assembled or compiled. Depending on the programming language used in the source file, the APW Assembler, C Compiler, or some other assembler or compiler processes the source file to create an object file. The object file contains 65816 machine-language instructions, data, and symbolic references to program routines. Object files, then, consist of machine-language instructions plus unresolved symbolic references.

   Your program can consist of several source files, and each source file can be in any of the APW programming languages. Each source file is converted into one or more object files by the APW Assembler and compilers.

2. The object files are input to the APW Linker, which combines all of the object files into a single load file and resolves symbolic references. The linker verifies that every routine referenced is included in the load file. If there are any routines that the linker has not found when it has finished processing all of the object files, it searches through any available library files for the missing routines. The linker removes symbolic references and replaces them with entries in special tables it creates called **relocation dictionaries**. The load file consists of blocks of machine-language code that can be loaded directly into memory (called **memory images**), plus relocation dictionaries that contain the information necessary to patch addresses into the memory images when the program is loaded into memory.

3. At program execution time, the load file is loaded into memory by the System Loader. The loader calls the Apple IIGS Memory Manager to request blocks of memory for the load file, loads the memory images, and uses the relocation dictionaries to patch the actual memory addresses into the machine-language code in memory. Because a load file can contain more than one **segment,** and because each segment can be processed independently, only part of the load file may be loaded into memory initially. OMF-file segmentation, a fundamental Apple IIGS concept, is discussed in the next section.

   The Memory Manager is the Apple IIGS tool set that allocates blocks of memory as needed and keeps track of which blocks of memory are available.

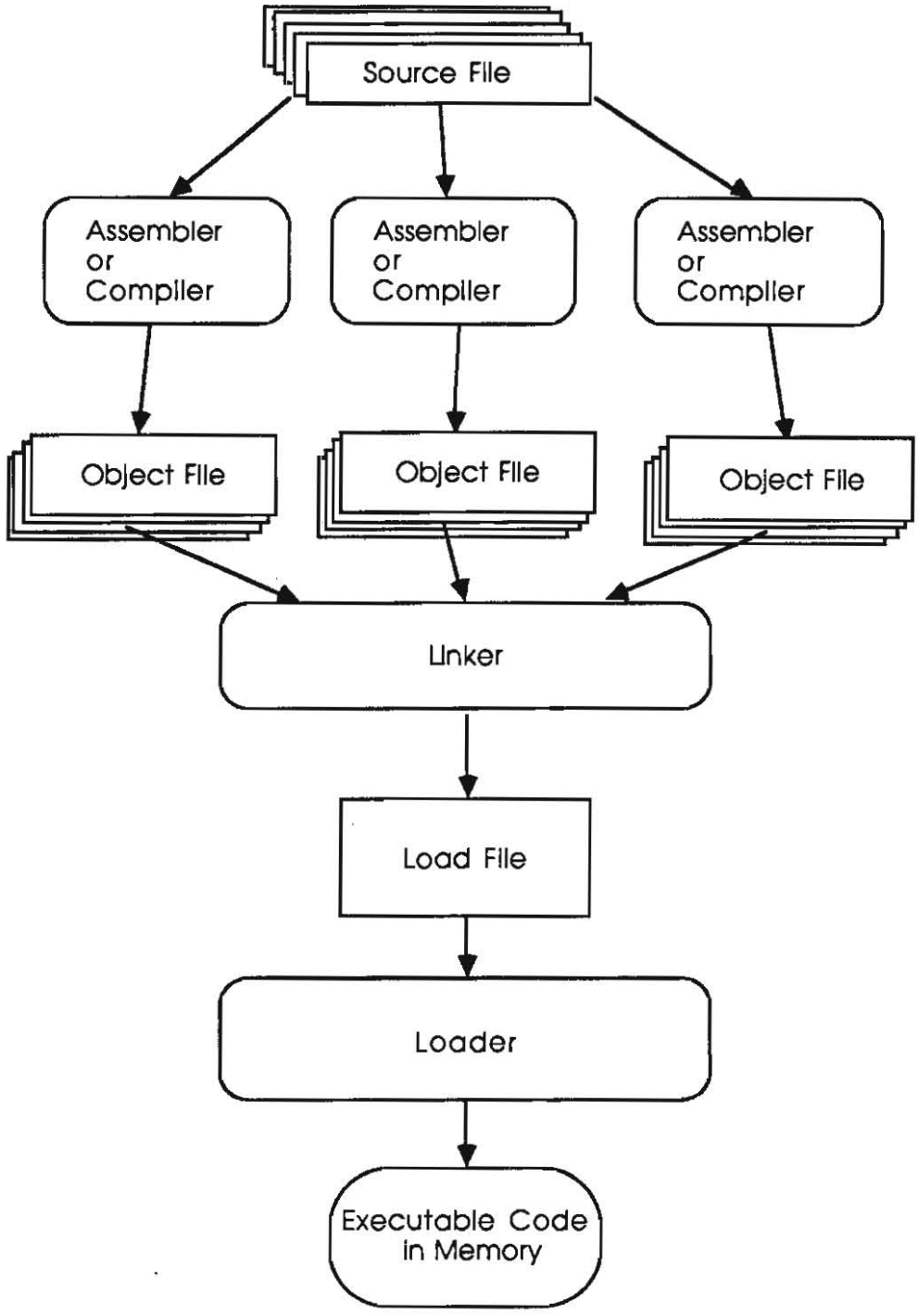


Figure 1.2. Creating an Executable Program on the Apple IIGS

Program Segmentation

In general, any computer program that consists of more than a few lines of code contains one or more subroutines; in addition, you may choose to segregate large blocks of data into

separate parts of the program. In APW, subroutines or blocks of data are given names that can be recognized by the linker; these named blocks of code are referred to as **segments**. In APW assembly-language programs, for example, you can assign labels to subroutines or blocks of data with the START—END and DATA—END directive pairs. The code between the START or DATA directive and the next END directive composes a segment. As illustrated in Figure 1.3, when you assemble or compile the program, each source-code segment becomes one **object segment.**

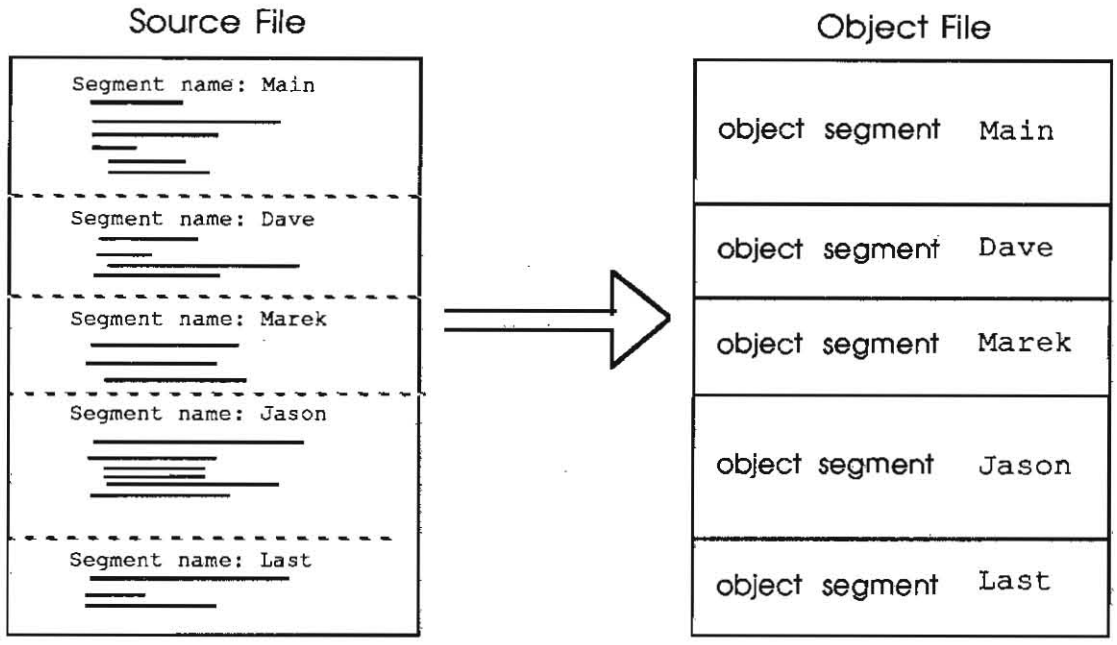


Figure 1.3. Assigning Object Segments in Your Source Code

The segmentation of source files and object files increases the flexibility and efficiency of the program development process. For example, it is not necessary to recompile an entire program each time you make a change, since each source-file segment can be compiled or assembled independently by APW compilers and assemblers in a process referred to as a **partial compile** or **partial assembly**. Object segments that are part of one program can be easily extracted for use in another program, since each object-file segment can be chosen independently for linking with a LinkEd command.

Apple IIGS load files are also segmented. The segmentation of load files allows a program to be loaded into memory in pieces rather than in one block, so that large programs can be loaded even when one large contiguous block of memory is not available. Under certain circumstances, load segments also allow parts of a program to be loaded and unloaded while the program is running so that memory can be used more efficiently; see the section "Dynamic Segments" later in this chapter for details.

It is important to understand the difference between object segments and **load segments.** Object segments generally correspond on a one-to-one basis with subroutines in the source file. Each load segment, on the other hand, can incorporate any number of object segments. Object segments are used by the linker to resolve references and to extract subroutines from library files. Load segments are used by the loader when loading a program into memory.

Object-segment names correspond to subroutine names; they are assigned in the source file. Some APW languages, such as 65816 assembly language and APW C, let you specify load-segment names in the source code as well; such load-segment names are optional, however. Each object segment must have a unique object-segment name, but any number of object segments can share the same load-segment name. You can also use LinkEd commands to assign names to load segments and to specify which object segments go in each load segment.

Source-file load segment names allow you to segment a load file without using the advanced linker. If you do not use a LinkEd file, all object segments with the same load-segment name are placed by the standard linker into the same load segment. The object-segment names are discarded by the linker; there is no record in a load segment of the object segments that went into it.

Source-file load-segment names are illustrated in Figure 1.4. The relationship of object segments to load segments is illustrated in Figure 1.5.

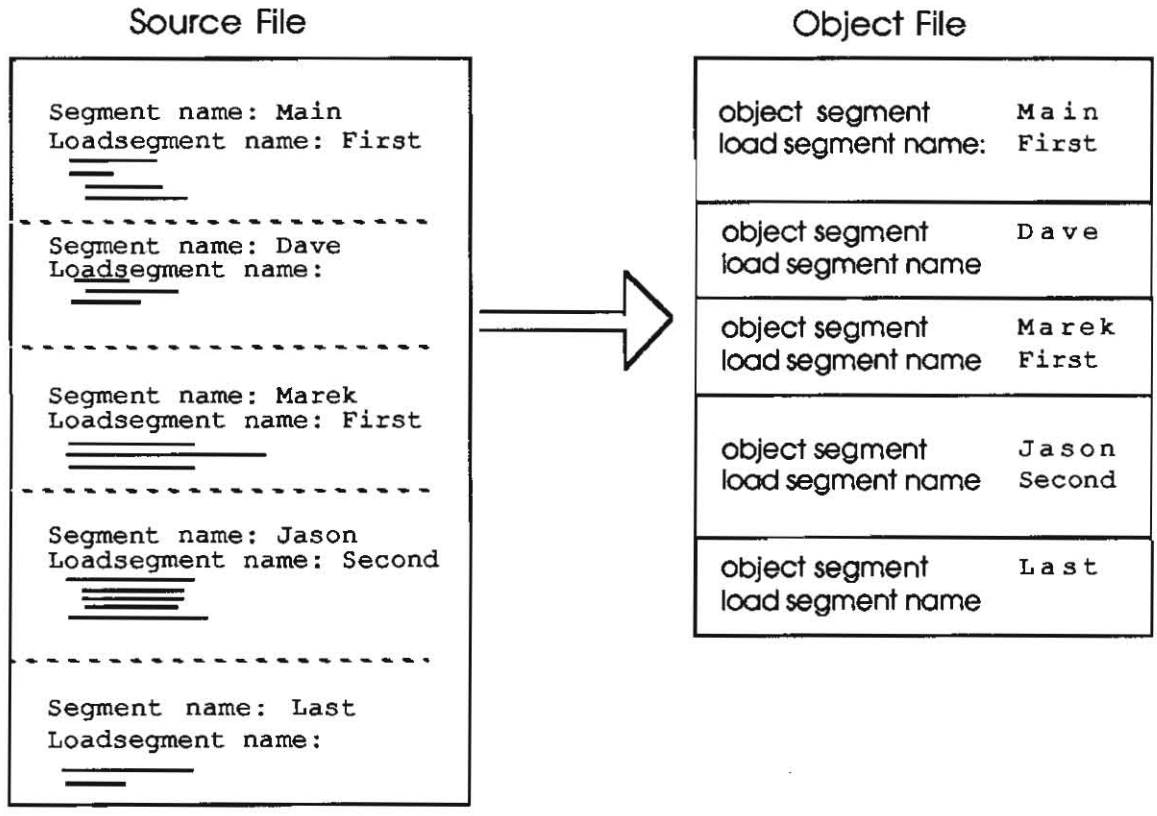


Figure 1.4. Assigning Load Segments in Your Source Code

The relationship between object segments and load segments can be made clearer if we first take a brief look at the structure of a segment in an OMF file. Each segment consists of a **segment header** and the **segment body**. The segment header is divided into **fields** containing the following information:

- The size of the segment.
- The type of segment (including code, data, static, and dynamic).
- The version number of the OMF with which this segment is compatible.

- The address in memory at which this segment is to be loaded. Normally, this field is 0, indicating that the segment is relocatable.
- The name of the segment.
- For object segments, the name of the load segment into which the standard linker should place this segment. For load segments, this header field is not used.
- Several other fields that need not concern us here (see the section "Segment Header" in Chapter 7 for a full description).

A load segment has only one name, the *name of the segment* (the *name of the load segment* field in the segment header is not used in a load segment). For object segments, however, these names are distinct; that is, both the object-segment name and the load-segment name fields are used. The object-segment name is used by a compiler when performing partial compiles (described in the section "Partial Assemblies or Compiles" in Chapter 3) and by the linker in resolving references and in extracting specific segments for linking (see the section "Linking with a LinkEd Command File" in Chapter 5). Load segment names are used by the loader when loading, unloading, and relocating segments.

Each object-segment name in an object file must be unique. In addition to the object-segment name, each object segment is also assigned a load-segment name; any number of object segments can have the same load-segment name. The standard linker places all object segments that share the same load-segment name into the same load segment. As illustrated in Figure 1.4, some programming languages (such as 65816 assembly language and APW C) let you assign your own load-segment name to an object segment; on the other hand, some compilers assign a load-segment name to the object segment for you. If no load-segment name was assigned in the source file or in a LinkEd file, the load-segment name can consist of a string of space characters.

For example, suppose your object file contains the object segments Peter, Paul, and Mary, and each of these object segments is assigned either to the load segment White or the load segment Black, as follows:

0. Object-segment name: Peter
   Load-segment name: White
1. Object-segment name: Paul
   Load-segment name: Black
2. Object-segment name: Mary
   Load-segment name: White

When the standard linker processes this file, the object-segment names Peter, Paul, and Mary are treated as references that must be resolved. Object segments Peter and Mary are placed in the same load segment, named White, and object segment Paul is placed in a separate load segment, named Black. Another example of the relationship between object segments, load-segment names, and load segments is illustrated in Figure 1.5. Note that the segments that have no load-segment name assigned in the source file are put by the standard linker into a load segment with a blank name (that is, the name consists of a string of space characters).

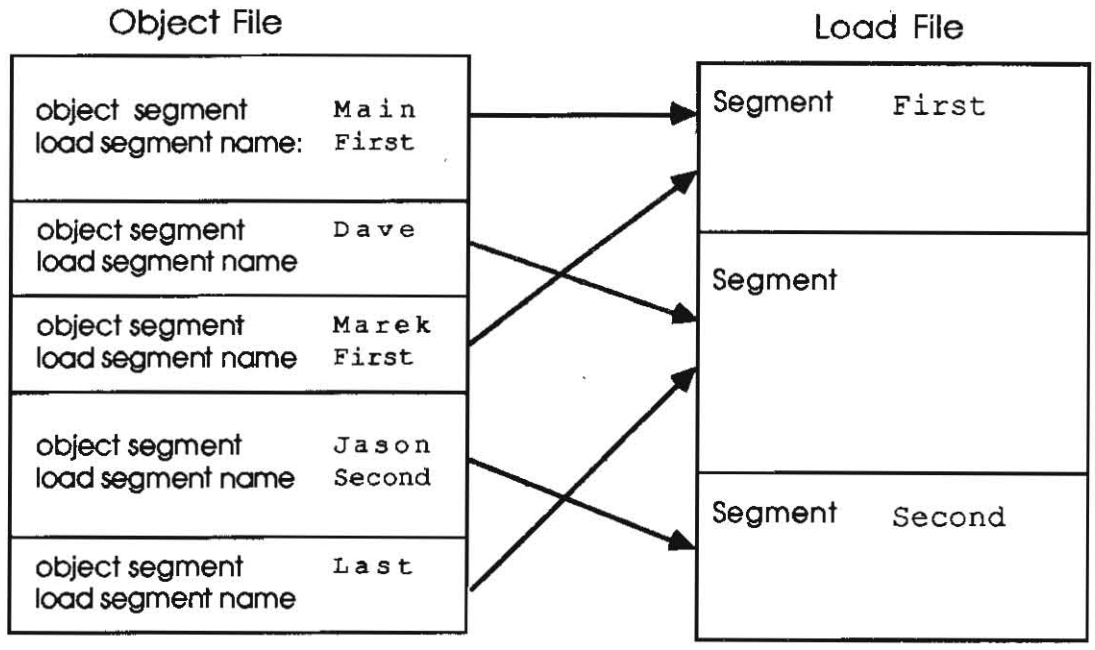


Figure 1.5. Relationship Between Object Segments and Load Segments

On the Apple IIGS computer, no single block of code can occupy more than 64K of contiguous memory. (In this manual, K is used to mean 1024 bytes.) To load a larger program than that, you must split it up into two or more load segments. When much of memory is already in use, it may be possible to load a program that is divided into several small load segments even if the same program in one or two load segments wouldn't fit. The Apple IIGS Memory Manager takes care of assigning each segment to a block of memory; the System Loader keeps track of where in memory the segment has been loaded and patches intersegment calls in each segment as it is loaded.

> **Note:** Although no single block of code can occupy more than 64K of contiguous memory, data *can* occupy more than 64K. The restriction is due to a limitation of the 65C816 microprocessor, which cannot execute code across a memory bank boundary.

Dynamic Segments

On the Apple IIGS computer, the combination of load segments together with the System Loader and Memory Manager makes possible the creation of **dynamic segments**. A dynamic segment can be loaded and unloaded automatically by the System Loader and Memory Manager during program execution. Dynamic segments can be used to fulfill the same function as overlays; that is, a dynamic segment that is not needed at a given time can be removed from memory to provide room in which to load another dynamic segment.

Dynamic segments are much more versatile than overlays, however: whereas overlays must always be loaded into the same location of memory, and that block of memory cannot be used by more than one program, dynamic segments (which, to be used effectively, should also be relocatable) can be loaded at any location in memory when needed. In addition, the System Loader and Memory Manager remove from memory a dynamic segment that is not

being used only if the memory is needed for something else; otherwise, the segment remains in memory and need not be reloaded the next time it is called.

Before the segment can be removed from memory, the application program must make the segment **purgeable** by using the System Loader's Unload Segment call or the Unload Segment by Number call. The System Loader is described in the *Apple IIGS ProDOS Reference*.

A segment that is not dynamic is referred to as **static**. A static segment is loaded at program boot time and is not unloaded or moved during execution. The first segment of any program that is loaded is static. Any other segments may be static, but (especially for large programs) the initial load of the program will be faster and the system will make more efficient use of memory if all infrequently used segments are dynamic. You can use a LinkEd command to make a segment dynamic; refer to the manual that came with the APW language you are using to see if there is also a way to assign dynamic segments in the source code.

Library Files

Library files contain routines that are useful to many different programs. On the Apple IIGS, all library files are in object module format, regardless of the language of the source file. An Apple IIGS library file (ProDOS file type $B2) can therefore be used by a program written in any source language. Some languages, such as APW C, come with a set of library files used by that language.

When the linker processes one or more object files and cannot resolve a symbolic reference, it assumes that it is a reference to a segment in a library file. If you use the standard linker, it searches any library files you name on the command line and then automatically searches all of the library files in the APW library prefix (`/APW/LIBRARIES/` on your original APW disk, for example). If you use a LinkEd command file, the advanced linker searches only the library files that you specify. Unless you are using the advanced linker, you do not even need to know the names of the library files in order to use them; the standard linker automatically finds the files and extracts the segments it needs.

You can create your own library files from one or more object files by using the MakeLib APW utility program. Figure 1.6 illustrates the process by which a library file is created. You specify one or more object files to be included in the library file. MakeLib concatenates the files and creates a special segment at the beginning of the file called the **library dictionary segment**. The library dictionary segment is the first segment of a library file; it contains the names and locations of all the **global symbols** in the file. (A global symbol is a label in one segment that can be referenced in another segment, as opposed to a **local symbol,** which can be used only within the segment in which it is defined.) The linker uses the library dictionary segment to find the segments it needs.

The library dictionary segment makes it possible for the linker to search a library file for global symbols much more rapidly than it can search an object file. Consequently, the linker will search a library dictionary segment multiple times if necessary to find segments referenced by other segments in the library file. The sequential order of the segments in a library file is therefore not important. If you were to use several library files, on the other hand, the order in which the files were searched *would* be important: if the linker first searched file `A` and then file `B`, for example, it could resolve a reference made in file `A` to a

global symbol in file B, but could not resolve a reference made in file B to a symbol in file A. It is for that reason that MakeLib allows you to include several object files in a single library file.

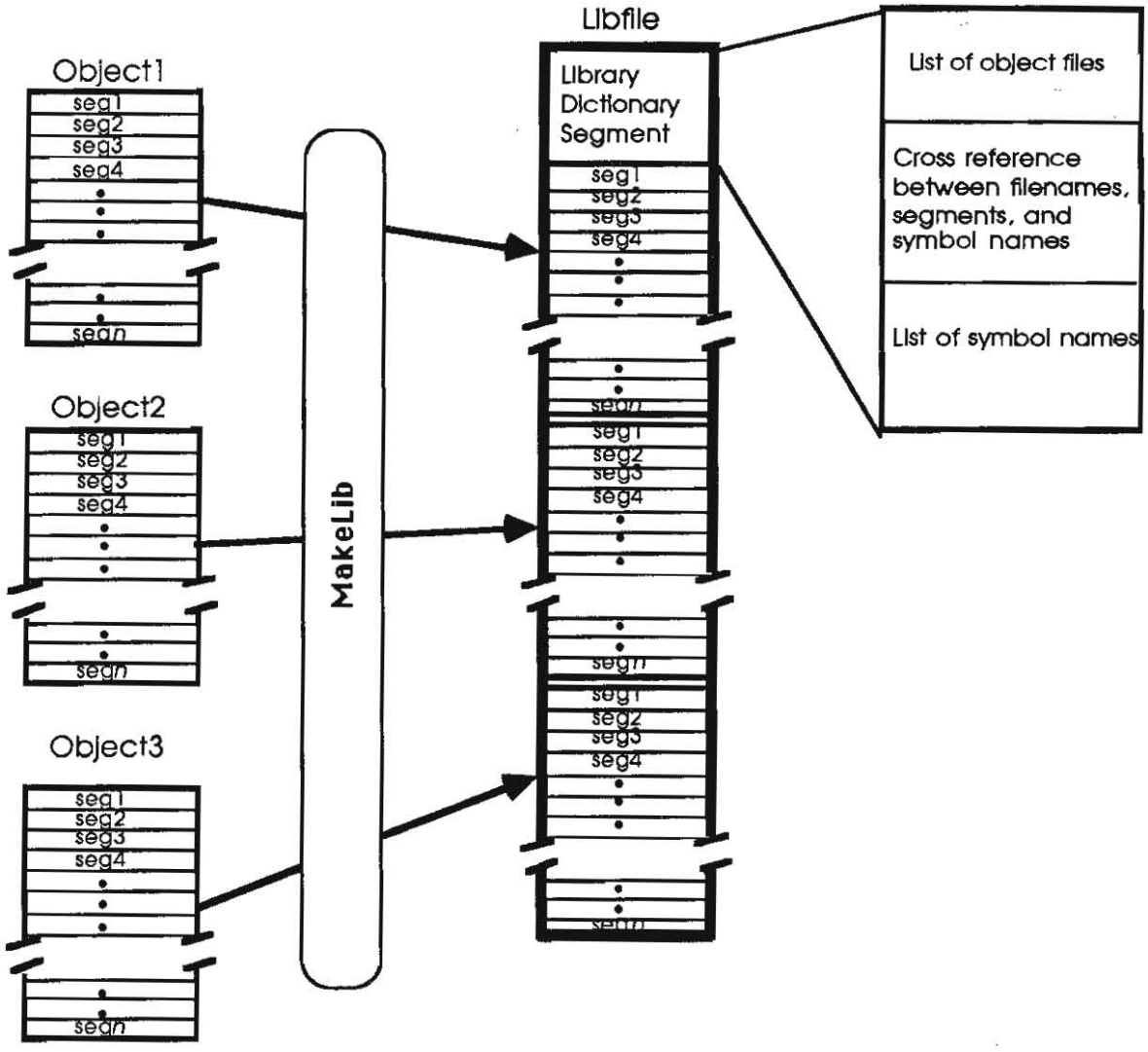


Figure 1.6. Relationship Between Object Files and Library Files

Emulation and Native Modes

The 65C816 processor of the Apple IIGS computer can run in **emulation mode** or **native mode**. In emulation mode, it behaves exactly like a 6502 processor and can run code written for the 6502 without modification. The Apple IIGS computer fully supports emulation mode by including ROM code and a memory structure that allows you to run programs written for 8-bit Apple computers, such as the Apple IIe and Apple IIc. When running in emulation mode, however, your program can use only the first 128K of Apple IIGS memory and cannot take advantage of the System Loader or Memory Manager. Native and emulation modes are discussed in the *Technical Introduction to the Apple IIGS* and described in detail in the *Apple IIGS Hardware Reference* manual.

Note: The ProDOS 8 loadable file format (called the *binary file format*), consisting of one absolute memory image along with its destination address, cannot be loaded by the Apple IIGS System Loader. You must use ProDOS 8 to load such a file.

See the description of the MakeBin utility in Chapter 3 for a way to create a ProDOS 8 binary load file with APW. Except for the section on MakeBin, this book assumes that you are writing programs to be run under ProDOS 16 in native mode on the the Apple IIGS computer.

Chapter 2

How to Use the Shell and Editor

The Apple IIGS Programmer's Workshop Shell provides your interface with APW. The shell provides a command interpreter to perform such functions as copying, moving, and deleting files, and running programs. You can assemble, compile, link, and run your programs with shell commands. In the Apple IIGS Programmer's Workshop, a single set of commands operates identically for all assemblers and compilers; you do not need to learn a new set of commands or operating sequence for each language you add to the system. APW also provides a full-featured text editor that you can use to write source code. Files written with the APW Editor are recognized by APW as language source files; the APW Shell can automatically select the correct compiler, assembler, or linker to process each source file.

This chapter introduces you to the use of the shell and editor. The following topics are covered in this chapter:

- the hardware and software needed to run APW
- how to install APW on a hard disk
- how to make your hard disk self-booting
- how to run APW on floppy disks
- how to run APW on a hard disk
- how to enter and execute APW Shell commands, including how to use wildcard characters, partial pathnames, and device names
- how to list the disk directory and read the directory listing
- how to set up and use a printer with APW
- how to launch Apple IIGS programs using APW

The following topics are too complex to be covered in this chapter in detail, but they are introduced here. For more information on each topic, see the chapters referred to in parentheses:

- how to use the editor (Chapter 4)
- how to use shell-command files, called *Exec files* (Chapter 3)
- how to compile (or assemble) and link a program (Chapter 3 and the manual that came with your compiler or assembler)
- how to use the Apple IIGS Debugger (*Apple IIGS Debugger Reference*)
- how to use APW utility programs (Chapter 3)

Only the most commonly used shell commands and features are described in this chapter; all APW Shell commands are described in detail in Chapter 3.

Important: Some commands, such as the COPY command, are used in examples and instructions given in this chapter. These examples do not show all of the ways in which the commands can be used. If you have trouble using any command, see the complete description of that command in Chapter 3.

What You Need

In order to use the Apple IIGS Programmer's Workshop, you must have the following hardware and software. A list of Apple IIGS manuals that you will find useful is given in the Preface.

- An Apple IIGS computer, or an Apple IIe computer with an installed Apple IIGS upgrade, with 256K of RAM.
- An installed Apple IIGS memory-expansion board with one megabyte (1024K) of RAM, for a total of 1280K of RAM.
- Two 3.5 inch disks containing the files shown in Appendix A.
- Two 800K disk drives or one 800K disk drive and one hard disk.
- Disks containing any other APW languages you intend to use with this system. The files on these disks must be installed on the Apple IIGS disk as described in the manuals that came with the APW language disks.

Important: APW requires one megabyte of *available* memory. That means that if you have 1280K of RAM in your Apple IIGS, you cannot assign more than 256K to a RAM disk.

A hard disk is highly recommended, especially if you intend to do multilanguage development or to develop large programs:

In addition, many developers find that an Apple II memory-expansion board is very useful in the Apple IIGS. You can use the board for a large RAM disk on which you can place library files, compilers and assemblers, the linker, and utility programs. Since many of these programs are loaded into memory from disk each time they are used, placing them on a RAM disk can speed up operation of the system during program development.

Note: See the Preface of this book for a list of the manuals you'll need to develop programs for the Apple IIGS, an explanation of the layout of this book, a description of the interrelationships of the books in the Apple IIGS technical reference suite, and a description of the typographical conventions used to describe commands in this book.

The files on the APW disks are listed in Appendix A.

Backing Up Your APW Disks

It is important to make a copy of your APW disks and to run APW from the copies only. Keep the original disks in a safe place so you can make new copies if something happens to the ones you have been using.

> **Important:** You must make copies of your APW disks even if you intend to copy APW onto a hard disk, in case something goes wrong during the installation procedure.

You can use any disk-copy utility you prefer to back up your APW disks. Remember that, if the new disk is not already formatted, you must format it before you copy APW onto it. The Apple IIGS System Disk that came with your computer includes formatting and disk-copy utilities.

> **Warning:** Formatting a disk erases all information on the disk you are formatting. Most disk duplication programs also destroy any information on the new disk (the disk being copied to). To be safe, be sure your APW disks are write-protected before you begin to copy them.

If you already have an older version of APW, you can launch your old APW and use the `INIT` and `COPY -D` commands to duplicate your new APW disks.

To run APW from the disk copies you just made, see the section "Running APW on Floppy Disks" in this chapter. To install APW on a hard disk, see the following section, "Installing APW on a Hard Disk."

> **Important:** You must give your copy of the `/APW` disk the volume name `/APW`, or the hard-disk installation procedure will not work correctly.

The Emergency Exit: Control-Reset

On occasion when you are testing a new program that runs under the APW Shell, the computer "hangs" (that is, you can neither quit the program nor get it to respond to any commands), or the program enters an infinitely repeating loop. In either case, you may be able to quit the program and return to the shell by pressing Control-Reset.

> **Warning:** Never press Control-Reset during a disk-write operation, as this can cause loss of data on the disk.

Use Control-Reset as a last resort only. Never use it instead of Apple-Period (which you can use to cancel most APW commands), as it does not cause the routine to terminate normally. If your program has written over portions of memory needed by APW, then APW may not be able to function normally after quitting the program and you might have to reboot the computer. If any files are open when you press Control-Reset, they may not be properly closed and the shell will not be able to resume operation.

If you do press Control-Reset, even if APW appears to be functioning normally, it is best to quit APW and restart it as soon as it is practical to do so.

Installing APW on a Hard Disk

If you want to install APW on a hard disk for the first time, follow the procedure in the next section, "First-Time Installation." If you want to update an already-installed APW, see the section "Updating APW," later in this chapter.

Important: Before you do anything else, you must make copies of your APW disks. If you have not already done so, use the procedure in the earlier section, "Backing Up Your APW Disks," to make copies of the disks.

First-Time Installation

Before you can install APW on your hard disk, you must have a properly formatted hard disk. If you have not already done so, follow the instructions that came with your hard disk to format it.

Use the following procedure to install APW on your hard disk.

Note: If want to make your hard disk boot directly into APW rather than into a program launcher, you must use a slightly different procedure. See the section "Booting Directly Into APW," later in this chapter, for a way to make your hard disk boot directly into APW .

1. Turn on the computer and hard disk and use the Control Panel to set the startup slot to your floppy disk drive. See the section on the Control Panel in the *Apple IIGS Owner's Guide* for instructions on setting the startup slot.
2. Insert the copy you made of the `/APW` disk in the startup disk drive and press Apple-Control-Reset to reboot the computer. The Apple IIGS Program Launcher should load from the disk.
3. Press Return twice to launch APW. Wait until the APW command-line prompt—a number sign (#)—appears at the left edge of the screen.
4. Enter the following command (substitute the volume name of your hard disk wherever you see *hardisk*). Remember to press Return after each command that you type.

   `INSTALL /APW /`*hardisk*`/APW`

   This command creates a subdirectory on your hard disk named `APW/` and copies the APW files from your `/APW` disk into the `APW/` subdirectory on the hard disk. This will take several minutes.
5. Remove the `/APW` disk from your disk drive and insert the `/APWU` disk.
6. Enter the following command:

   `INSTALL /APWU /`*hardisk*`/APW`

   This command copies the files from your `/APWU` disk into the `APW/` subdirectory on the hard disk. This will take several minutes.

You now have APW installed on your hard disk. If your hard disk is already self-booting, and if you have ProDOS 16 and the System Loader Version 1.2 or later, you can remove the `/APWU` disk from the disk drive, reset the Control Panel to boot from the hard disk, and begin using APW immediately. If you have never installed an Apple IIGS system on your hard disk, or if the version of ProDOS 16 or the System Loader are earlier than version 1.2, then follow the instructions in the next section to make your hard disk self-booting.

Important: You must have ProDOS 16 and the System Loader Version 1.2 or later to run APW. To find out what version of ProDOS 16 and the System Loader you have on your hard disk, boot the disk. The version numbers are displayed on the screen while the load process is taking place.

Making Your Hard Disk Self-Booting

The easiest way to set up your hard disk to be self-booting is to copy the Apple IIGS System Disk onto the hard disk, as described in the next section, "Copying the Apple IIGS System Disk." You must have Version 2.0 or later of this disk. If you do not have a recent version of the system disk, you can copy the system from your `/APW` disk instead. To do so, use the procedure in the section "Copying the System From the APW Disk." To make your hard disk boot directly into APW, follow the procedure in the section "Booting Directly Into APW."

Copying the Apple IIGS System Disk

To make your hard disk self-booting by copying the Apple IIGS System Disk onto your hard disk, use the following procedure:

1. Turn on the computer and hard disk and use the Control Panel to set the startup slot to your floppy disk drive. See the section on the Control Panel in the *Apple IIGS Owner's Guide* for instructions on setting the startup slot.
2. Insert the copy you made of the `/APW` disk in the startup disk drive and press Apple-Control-Reset to reboot the computer. The Apple IIGS Program Launcher should load from the disk.
3. Press Return twice to launch APW. Wait until the APW command-line prompt—a number sign (#)—appears at the left edge of the screen.
4. Remove the `/APW` disk from the disk drive and insert the Apple IIGS system disk in the drive.
5. Execute the following command to copy the files on the system disk onto your hard disk (substitute the volume name of your hard disk wherever you see *hardisk*):

   ```
   COPY -C /SYSTEM.DISK/= /hardisk
   ```

6. Use the Control Panel desk accessory to set the startup slot so that the computer will boot from your hard disk.
7. Remove the system disk from the disk drive and press Apple-Control-Reset to cause the machine to reboot from the hard disk. When you boot from the hard disk, you will get a program launcher. Select and open the APW folder, and then select and open the file `APW.SYS16` to launch APW.

Copying the System From the APW Disk

To make your hard disk self-booting by copying the system files from the `/APW` disk onto your hard disk, use the following procedure:

1. Turn on the computer and hard disk and use the Control Panel to set the startup slot to your floppy disk drive. See the section on the Control Panel in the *Apple IIGS Owner's Guide* for instructions on setting the startup slot.
2. Insert the copy you made of the /APW disk in the startup disk drive and press Apple-Control-Reset to reboot the computer. The Apple IIGS Program Launcher should load from the disk.
3. Press Return twice to launch APW. Wait until the APW command-line prompt—a number sign (#)—appears at the left edge of the screen.
4. Execute the following commands to copy the system files on the /APW disk onto your hard disk (substitute the volume name of your hard disk wherever you see *hardisk*):

   ```
   COPY -C /APW/PRODOS /hardisk
   COPY -C /APW/SYSTEM /hardisk
   ```

   None of the APW files are copied from the /APW disk to your hard disk in response to these commands, but all of the system files that you need to make your hard disk self-booting, including the Apple IIGS Program Launcher program, are copied.
5. Use the Control Panel desk accessory to set the startup slot so that the computer will boot from your hard disk.
6. Remove the /APW disk from the disk drive and press Apple-Control-Reset to cause the machine to reboot from the hard disk. When you boot from the hard disk, you will get the Apple IIGS Program Launcher. Select and open the APW folder, and then select and open the file APW.SYS16 to launch APW.

Updating APW

If you have previously installed a version of APW on your hard disk, you can use the procedure described earlier in this chapter in the section "First-Time Installation," to replace it with the latest version of APW. Before you do so, however, consider the following points:

- The installation routine assumes that you have a directory on your hard disk named */hardisk*/APW. If not, it creates one and copies APW into that subdirectory. If APW on your hard disk is in a subdirectory named something *other* than */hardisk*/APW, substitute that name in the installation procedure.
- The installation procedure replaces the files SYSCMND, LOGIN, and SYSTABS in the APW/SYSTEM subdirectory. If you have customized any of these files, you should rename them before installing the new APW, and then either edit or replace the new versions of the files as appropriate.
- You must have ProDOS 16 and System Loader Version 1.2 or later to run APW. To find out what version of ProDOS 16 and the System Loader you have on your hard disk, boot the disk. The version numbers are displayed on the screen while the load process is taking place. To replace the system files on your hard disk with more recent versions, follow either of the two procedures described earlier in the section "Making Your Hard Disk Self-Booting."

Adding Languages to APW

When you obtain a new language compiler for your APW system, follow the procedure described in the manual that came with that language to install it. Before you do so, however, consider the following points:

- There are several files that may be replaced by the installation procedure. However, the versions of these files on your disk might contain information that you do not want to lose. For example, the SYSCMND file might contain APW commands that are not included in the SYSCMND file of the new language. Before installing the new language, rename the following files so that they won't be overwritten during installation:

  ```
  APW/SYSTEM/SYSCMND
  APW/SYSTEM/SYSTABS
  APW/SYSTEM/LOGIN
  ```

  All of these files can be edited with the APW Editor. The structure and use of the SYSCMND file is discussed in the section "Command Types and the Command Table" in Chapter 3. The SYSTABS file is described in the section "Setting Editor Defaults" in Chapter 4. The LOGIN file is discussed in the section "LOGIN Files" in Chapter 3.

- If the disks that come with the language do not include a complete APW system and you want to run APW from floppy disks, you will have to prepare an APW system disk for that language. The easiest way to prepare such a disk is to copy the /APW disk, and then delete the file LANGUAGES/ASM65816 and the subdirectory LIBRARIES/AINCLUDE using the following commands:

  ```
  DELETE 5/ASM65816
  DELETE -C 2/AINCLUDE/=
  DELETE 2/AINCLUDE
  ```

  Place the new compiler in the subdirectory APW/LANGUAGES/ in place of ASM65816. Place any library files that came with the compiler in the subdirectory APW/LIBRARIES.

- You must have ProDOS 16 and System Loader Version 1.2 or later to run APW. To find out what version of ProDOS 16 and the System Loader you have on your disk, start up the computer using that disk. The version numbers are displayed on the screen while the load process is taking place. If the installation procedure replaces your system files with older versions, you have to reinstall more recent versions.

  To replace the system files on your hard disk with more recent versions, follow either of the two procedures described in the section "Making Your Hard Disk Self-Booting," earlier in this chapter. To replace the system files on a floppy disk with the ones on your /APW disk, use the following commands (substitute the volume name of your new disk wherever you see *disk*):

  ```
  COPY -C /APW/PRODOS /disk
  COPY -C /APW/SYSTEM /disk
  ```

Booting Directly Into APW

For your disk to boot directly into APW, the file `APW.SYS16` must be the first system program in the root directory of your disk and there must be no file named `START` in the `SYSTEM/` subdirectory. In this case there should be only one `SYSTEM/` subdirectory on the disk, immediately under the volume directory. Both hard disks and floppy disks can be configured to boot directly into APW. You can launch any other program you wish from APW by typing in its pathname and pressing Return.

> **Important:** For the procedures in this section to work, `APW.SYS16` must be the *first* system program (that is, program that ends in the extension `.SYS16` or `.SYSTEM`) in the root directory of your disk. If there is another such file before `APW.SYS16`, you must remove it or the disk will boot into that program instead.

Hard Disk

To make your hard disk boot directly into APW, use the following procedure:

1. Turn on the computer and hard disk and use the Control Panel to set the startup slot to your floppy disk drive. See the section on the Control Panel in the *Apple IIGS Owner's Guide* for instructions on setting the startup slot.
2. Insert the copy you made of the `/APW` disk in the startup disk drive and press Apple-Control-Reset to reboot the computer. The Apple IIGS Program Launcher should load from the disk.
3. Press Return twice to launch APW. Wait until the APW command-line prompt—a number sign (#)—appears at the left edge of the screen.
4. Enter the following command (substitute the volume name of your hard disk wherever you see *hardisk*). Remember to press Return after each command that you type.

   `INSTALL /APW /`*`hardisk`*

   This command copies the APW files from your `/APW` disk into the `volume` directory on the hard disk. This will take several minutes.
5. Remove the `/APW` disk from your disk drive and insert the `/APWU` disk.
6. Enter the following command.

   `INSTALL /APWU /`*`hardisk`*

   This command copies the files from your `/APWU` disk into the `volume` directory on the hard disk. This will take several minutes.
7. If you have not already done so, make the hard disk self-booting by using either procedure described in the earlier section "Making Your Hard Disk Self-Booting."
8. Use the following command to remove the `START` file:

   `DELETE /`*`hardisk`*`/SYSTEM/START`
9. Use the Control Panel to set the startup slot to your hard disk drive.

Your hard disk should now boot directly into APW.

Floppy Disk

To configure a floppy disk to boot directly into APW, use the following procedure:

1. Make a backup copy of your /APW disk as described in the earlier section "Backing Up Your APW Disks."
2. Format a second floppy disk and name it /APW.BOOT.
3. Insert the copy you made of the /APW disk in the startup disk drive and turn on the computer. The Apple IIGS Program Launcher should load from the disk. If it does not, make sure your Apple IIGS is set to boot from the disk drive you used (see the section on the Control Panel in the *Apple IIGS Owner's Guide*).
4. Press Return twice to launch APW. Wait until the APW command-line prompt—a number sign (#)—appears at the left edge of the screen.
5. Place the /APW.BOOT disk in the second disk drive and enter the following commands. Remember to press Return after each command that you type.

```
COPY -C /APW/PRODOS /APW.BOOT
COPY -C /APW/SYSTEM /APW.BOOT
COPY -C /APW/APW/= /APW.BOOT
DELETE /APW.BOOT/SYSTEM/START
```

The /APW.BOOT disk should now boot directly into APW. It will operate exactly as described for the /APW disk in the next section, "Running APW on Floppy Disks," except that the Program Launcher is not on the disk.

Running APW on Floppy Disks

You need at least two 800K disk drives to use APW: one to hold the /APW (or /APW.BOOT) disk, and one to hold either the /APWU disk or a disk containing only the files you are working on.

> **Important:** Do *not* run APW from the original product disks. Make copies of your APW disks for everyday use, and put the original disks in a safe place.

The /APW disk contains the Apple IIGS Program Launcher and a fully functional APW system, including the APW Assembler. This disk lacks only the help files and some of the APW utility programs. See Appendix A for a list of all the files on the /APW disk. The /APWU disk contains a full set of utility programs plus the help files for all the APW commands.

There are two ways to run APW on two 800K disk drives, as follows:

- If you need an entire 800K disk for your program files, place /APW in the startup disk drive and the disk containing your files in the second drive, and then start up the computer. The Apple IIGS Program Launcher should load from the disk. Press the Return key twice to launch APW. You can now do anything described in this manual except consult the on-line help files for APW commands or execute some of the utility programs.

Note: To prepare an APW disk that boots directly into APW, follow the procedure in the section "Booting Directly Into APW," earlier in this chapter.

- If your program files will fit on the /APWU disk, or if you need to use the help files or utility programs on that disk, then launch APW as before and place the /APWU disk in the second disk drive. To cause APW to look on the /APWU disk for the help files and utility programs, enter the following command:

  ```
  MU
  ```

  If at any time you want to remove the /APWU disk and run APW exclusively from the /APW disk, enter the following command:

  ```
  UMU
  ```

The directory that is assumed when you do not specify a prefix in a pathname is called the **current prefix.** If the /APW disk is in your first disk drive and all your program files are on the disk in the second disk drive, you may wish to set the system to use a directory on your program-file disk as the current prefix. Use the APW Shell's PREFIX command to change the current prefix. For example, if your programs are in a subdirectory called /APWU/MYPROGS/ in the second disk drive, type the following command and press Return:

```
PREFIX /APWU/MYPROGS
```

Once you have set the current prefix to that of your program disk, you need not include the prefix in pathnames when executing commands. For example, if the current prefix is /APWU/MYPROGS/, you could use the following command to obtain a directory listing of the subdirectory /APWU/MYPROGS/CSOURCE/:

```
CATALOG CSOURCE
```

Note: Do *not* include a slash (/) before the pathname when you omit the current prefix from a pathname, or APW will look for a volume by that name. For example, if you typed CATALOG /CSOURCE in the preceding example, you would get the message `Volume not found.`

Prefixes used by APW are discussed in detail in the section "Using Prefix Numbers" later in this chapter.

Keep the /APW disk in the first disk drive while you are running APW so that the system can have access to the APW programs on that disk.

Each time you start APW, it looks for a file named LOGIN in the APW system prefix (/APW/APW/SYSTEM/LOGIN on the /APW disk, for example). The LOGIN file should have an APW language type of EXEC (see the section "Listing a Directory" later in this chapter). You can include any valid APW command in this file. If it finds a LOGIN file, APW executes it before doing anything else.

You can use a LOGIN file to set system defaults (such as the printer slot), to set the current prefix, to read a command table containing command-name aliases, or even to execute commands or utility programs. Examples of LOGIN files (and the procedures for creating them) are shown in the sections "Using Prefix Numbers" and "Using a Printer" later in this chapter.

You need not have a LOGIN file in your system; if there is no LOGIN file, APW uses default settings for system parameters.

Running APW on a Hard Disk

Once you have launched APW by selecting APW.SYS16 in the APW/ subdirectory on your hard disk, all of the APW commands work exactly as they do on a floppy disk. The current directory is the APW/ subdirectory on the hard disk.

Your hard disk should have enough room on it to allow you to keep all of your APW files and your program files on the same disk. To avoid confusion with APW system files, you should create one or more new subdirectories to hold your program files. Use the CREATE command to create a new subdirectory. To create the subdirectory MYFILES/ in the current prefix, for example, use the following command:

```
CREATE MYFILES/
```

Now to change the current directory to MYFILES/, you can use the following command:

```
PREFIX MYFILES/
```

As discussed in the previous section, once you have set the current prefix to that of your program subdirectory, you need not include the prefix in pathnames when executing commands.

The LOGIN file works on the hard disk just as it does on a floppy disk. Each time you select APW from the chooser or finder program, LOGIN is executed before the APW Shell prompt (#) appears on the screen.

Shell Commands

There are two main methods of sending commands to the APW Shell command interpreter. Either

- Type in any APW command on a **shell command line** and press the Return key. The shell is ready to accept a command when a number-sign (#) prompt appears at the screen's left edge followed by a solid-block cursor.

or

- Create a file of APW commands with the language type EXEC. When you enter the name of an Exec file as a command, APW executes the commands in the file as if they were typed from the keyboard.

This section describes how to enter commands on a command line, but the rules presented here also apply to commands in Exec files. Exec files are described briefly in the section "Using Exec Files" later in this chapter and in detail in the section "Exec Files" in Chapter 3.

Entering Commands

APW requires every command to be entered in full, exactly as it appears in the list of commands you get when you type HELP and press Return (except that the command interpreter is not case sensitive). It is not necessary for you to type in the entire command, however; instead, you can type in the first letter or first few letters of the command and then press the Right Arrow key (→). The shell consults the command table and prints out the full command name of the first command it finds that matches the letters you typed. For example, suppose you type the following command:

```
CO →
```

The shell finds the first command name that begins with CO in the command table, and prints the full command name:

```
COMMANDS
```

When you press Return, the entire command line is sent to the command interpreter regardless of the location of the cursor on the command line.

If you like, you can add command aliases to the command table. For example, to make the shell recognize the command CMP as an alias for COMPILE, add CMP to the command table with the same command number as COMPILE. See the section "Command Types and the Command Table" in Chapter 3 for instructions on modifying the command table. You can also create temporary aliases for commands with the shell's ALIAS command.

You can use the line-editing commands in Table 2.1 when you are entering a command or modifying a previously entered command.

> **Note:** The APW Shell command interpreter is not case sensitive; that is, you can enter commands and filenames in any combination of uppercase and lowercase letters. Command examples are shown in uppercase letters in this book to help distinguish them from other text and because they are listed that way in the command table and help files.

Table 2.1. Line-Editing Commands

Command	Meaning
←	cursor left
→	cursor right
♁-> or ♁-.	end of line
♁-< or ♁-,	beginning of line
Delete	delete character left
♁-Y or Control-Y	delete to end of line
♁-E or Control-E	toggle insert mode
♁-Z or Control-Z	clear line and cancel command without saving changes
Esc, Clear, or Control-X	clear line and cancel command without saving changes
Return or Enter	save changes and execute command

File Not Found and Other Errors

When you type a command and press Return, APW first checks the command table to see if it is a standard command. If the command is not in the command table, APW assumes it is the name of an executable file and asks ProDOS 16 to open a file by that name in the current prefix. If ProDOS 16 does not find a file by that name, the message `ProDOS: File not found` is printed on the screen. This message indicates that ProDOS 16 could not find a file with the name of the command you typed. Check the prefix and spelling of your command and try again.

The `File not found` error can be confusing when you have also typed a pathname as a parameter for the command. For example, suppose that you want to edit the file `MYFILE`, and that you therefore enter the following command:

```
ED MYFILE
```

Unfortunately, `ED` is not a valid APW command (unless you have added it to the command table yourself or made it an alias for `EDIT`). APW looks in the command table for `ED`, doesn't find it, and calls ProDOS 16 to try to open a file named `ED`. ProDOS can't find the file, and the message `File not found` is printed on the screen. When you see this message, it is important to realize that the file that ProDOS 16 couldn't find is `ED`, *not* `MYFILE`.

The `File not found` message also appears when you attempt to execute the Paste command in the editor without first executing a Copy or Cut command. When you execute the Paste command, the Editor looks for the file `SYSTEMP` in the work prefix; this file does not exist, however, unless a Copy or Paste command has been executed first.

A similar problem can occur if you remove your APW disk from the disk drive or change a prefix used by APW (see the section "Using Prefix Numbers" in this chapter) and then try to execute an external command (such as `INIT`) or to read a help file. In this case, ProDOS 16 cannot find the directory containing the utility program or help file, and the message `Volume not found` or `Path not found` is printed to the screen. Again, it is important to realize that the volume or path that could not be found is the one containing the utility or help file, *not* one used in a parameter to the command.

For example, if you execute the `MU` command to use the utility files on the `/APWU` file, and then remove the `/APWU` disk from the disk drive and enter the command `DUMPOBJ MYFILE`, ProDOS 16 cannot find the volume `/APWU` in order to load the DumpObj utility. In this case, the message `Volume not found` appears on the screen.

Suspending Execution and Cancelling Commands

In most cases when the shell lists text on the screen, you can cause the listing to pause by pressing any key: the Spacebar key is often convenient. You can use this feature, for example, to stop long catalog listings before they scroll off the screen or to read text files that you list on the screen with the `TYPE` command. To continue the listing, press any key.

Most APW Shell commands can be cancelled by pressing Apple-Period (⌘-.). When a command prompts you for a filename, you can cancel the command by pressing Return instead of entering a filename.

Some prompts for shell commands require you to enter `Y`, `N`, or `Q` in response (see, for example, the section "Using Wildcard Characters" later in this chapter). In this case, you can type `Q` and press Return to act on the files already selected and then terminate the command, or press Esc to cancel the command immediately without acting on any files. See the section "Using Wildcard Characters" later in this chapter for an example of this feature.

Scrolling Through Commands

You can press the Up Arrow (↑) and Down Arrow (↓) keys to scroll through the last 20 commands that you have entered. You can then modify a previous command and press Return to reenter it. Each time you enter or reenter a command, that command is appended to the 20-command list.

To try out this feature, boot APW and enter the command `CATALOG` to get a directory listing of the current directory (`APW/`). The directory listing includes the subdirectory `UTILITIES/`. To obtain a listing of this subdirectory, first press the Up Arrow key. The command `CATALOG` reappears on the screen with the cursor at the end of the command line. Type a space and the word `UTILITIES`. The command line now reads `CATALOG UTILITIES`. Press Return to get the directory listing. The command sequence is as follows (the commands you type are shown in boldface):

#CATALOG

[press Return]

[directory listing printed]

#↑
#CATALOG **UTILITIES**

[press Return]

[directory listing printed]

The `UTILITIES/` subdirectory includes the subdirectory `HELP/`. Press the Up Arrow key again so that the command `CATALOG UTILITIES` reappears. Type `/HELP` (so that the command line reads `CATALOG UTILITIES/HELP`) and press Return. The `HELP/` subdirectory is listed.

Now press the Up Arrow key again. The command `CATALOG UTILITIES/HELP` reappears. Press the Up-Arrow key another time. The command `CATALOG UTILITIES` reappears. Press the Up-Arrow key once more to get the command `CATALOG` again. The command sequence is as follows:

#CATALOG

[press Return]

[directory listing printed]

```
#↑
#CATALOG UTILITIES
```

[press Return]

[directory listing printed]

```
#↑
#CATALOG UTILITIES/HELP
```

[press Return]

[directory listing printed]

```
#↑
#CATALOG UTILITIES/HELP ↑
#CATALOG UTILITIES ↑
#CATALOG
```

The 20-command list is circular; that is, once you have used the Up Arrow key or the Down Arrow key 20 times to scroll through the 20 commands, pressing the same arrow key one more time returns you to the command you started with. Experiment with the command-line scrolling and with the line-editing commands in Table 2.1 for a while. You will find that these functions can save you a lot of time and frustration in entering long or complex commands.

Entering Multiple Commands

You can enter several commands on one line by separating each command from the preceding command with a semicolon. For example, to change the name of the file WHITE to BLACK and then open the file for editing, type in the following command line and press Return:

```
RENAME WHITE BLACK ; EDIT BLACK
```

You can use this technique in Exec files as well.

Responding to Parameter Prompts

If you enter an APW command that requires one or more parameters and do not include a required parameter, then APW prompts you for it. You are *not* prompted for optional parameters. For example, the following exchange shows what happens when you enter the RENAME command without parameters. The words shown in boldface are the ones you type in:

```
RENAME
File to rename:   /APW/OLDNAME
New name:   /APW/NEWNAME
```

APW prompts you for parameters in the sequence in which they are shown in the command descriptions in Chapter 3. For example, the RENAME command requires the current name of the file followed by the new name for the file, and that is the order in which you are prompted.

If two parameters are required and you include only one, the shell always assumes that the first parameter was included and the second one was missing. For example, to change the name of OLDNAME to NEWNAME, you could use the following command sequence:

```
RENAME   /APW/OLDNAME
New name:   /APW/NEWNAME
```

If a wildcard character is allowed in the command line, you can use one in response to the prompt. Wildcard characters are described in the section "Using Wildcard Characters" in this chapter.

Since you are not prompted for optional parameters, there are some operations you cannot carry out by simply responding to prompts. For example, if you do not include any parameters after the COPY command, you are prompted for the filename of the file to copy. However, since the target pathname is not a required parameter, you are *not* prompted for it. If you do not include the target pathname on the command line (or on the same line as the source filename in response to the File Name prompt), then the current prefix is always used as the target directory (and the filename is not changed). The following example shows what happens when you include only the parameters for which you are prompted when using the COPY command:

```
COPY
Source file name:  MYFILE
File exists-replace it?
```

Since you used the current prefix for MYFILE, and the current prefix is also assumed for the target directory (because no target directory was specified), APW asks if you want to replace an existing file. To replace the file, type Y and press Return. To provide a new filename, type N and press Return. The following prompt appears:

```
New name:
```

Type the new name for the file and press Return.

You can include both the source file and the target directory in response to the prompt, as in the following example:

```
COPY
Source file name:   MYFILE  /MYPROGS/CSOURCE
```

In this case, the file named MYFILE in the current prefix is copied to the directory /MYPROGS/CSOURCE/.

Important: Some commands, such as the COPY command, are used in examples and instructions given in this chapter. These examples do not show all of the ways in which the commands can be used. If you have trouble using any command, see the complete description of that command in Chapter 3.

Pathnames

Under ProDOS 16 on the Apple IIGS, each disk (or RAM disk) has a name, called a **volume name**, and a **directory** of files on that disk (technically, a disk can contain more than one volume, but this is hardly ever the case). Among the files in the volume directory can be other directory files, which catalog the contents of **subdirectories**.

Note: Although a directory or subdirectory is actually a file on the disk, in the following discussion the word *file* refers only to the program file or text file that we are using the pathname to specify.

When you specify a file in an APW command, as when indicating which file to edit or utility to execute, you must specify the file's **pathname**. A pathname consists of a string of names, each preceded by a slash (/). The first name in a **full pathname** is the name of a volume directory. Successive names indicate the path, from the volume directory through any subdirectories to the file, that ProDOS 16 must follow to find the file. A **partial pathname** is a portion of a pathname; it must include the filename, and may include one or more subdirectory names. A partial pathname does not include the volume name and does not begin with a slash. A prefix is that part of the pathname that is left over when you remove the partial pathname: it begins with a slash and the volume name, and can include one or more subdirectories. The prefix does not include the filename.

Assume, for example, that you want to edit a file called

```
/APW/MYPROGS/C.SOURCE/GOODSTUFF
```

The filename is GOODSTUFF. There are two possible partial pathnames for this file, as follows:

```
C.SOURCE/GOODSTUFF
MYPROGS/C.SOURCE/GOODSTUFF
```

There are three possible prefixes for this file, as follows:

```
/APW/
/APW/MYPROGS/
/APW/MYPROGS/C.SOURCE/
```

As described in the following sections, you can use partial pathnames, **prefix numbers**, **device names**, and **wildcard characters** when specifying a pathname in APW.

Using Partial Pathnames

When you execute an APW command that requires a pathname, you can enter the full pathname or a partial pathname. If the pathname in the command does not begin with a slash (/), APW assumes that a partial pathname is being used and places the current prefix

in front of the pathname in the command. When you first launch APW from a chooser or finder, the current prefix is set by the chooser or finder; it is typically the subdirectory containing the chooser or finder program. If you use a self-booting disk, the current prefix is the subdirectory containing the file `APW.SYS16`. You can change the current prefix at any time with the `PREFIX` command.

For example, when you boot APW from the 3.5-inch disk that came with the system, the current prefix is set to `/APW/`, the name of the boot volume. In this case, the following two commands are equivalent:

```
CATALOG /APW
CATALOG
```

The current prefix can include as many levels of subdirectories as you wish (within the 64-character limit on the length of pathnames set by ProDOS 16). For example, if you are working on a hard disk you might set the current prefix to `/HARDISK/APW/`. In this case, the following two commands are equivalent:

```
CATALOG /HARDISK/APW/MYPROGS
CATALOG MYPROGS
```

Note: Do *not* include a slash (/) before the pathname when you omit the current prefix from a pathname, or APW will look for a volume by that name. For example, if you typed `CATALOG /MYPROGS` in the preceding example, you would get the message `Volume not found`.

You can "back up" one directory level from the current prefix by starting the partial pathname with two periods (..). For example, if the current prefix is `/HARDISK/APW/`, you could use either of the following two commands to edit `/HARDISK/MYFILE`:

```
EDIT /HARDISK/MYFILE
EDIT ../MYFILE
```

Because APW uses standard prefixes to find the APW system files it needs, APW commands and utilities continue to work correctly when you change the current prefix. For example, when you execute the `MAKELIB` command on a standard APW floppy disk, APW loads the file `/APW/UTILITIES/MAKELIB`, no matter what the current prefix is set to. The prefixes that APW searches for APW system files can also be changed with the `PREFIX` command, as discussed in the next section, the section "Standard Prefixes" in Chapter 3, and the description of the `PREFIX` command in Chapter 3.

Using Prefix Numbers

ProDOS 16 provides eight prefix numbers that can be set to specific prefixes. Prefix 0 is the current prefix. APW uses prefixes 2 through 6 to determine where to search for certain files. Program launchers, including APW, set prefix 1 to the prefix of the last program executed. The prefixes are set to the default values shown in Table 3.1 when you start APW. You can change any of the ProDOS 16 prefixes with the `PREFIX` command, as described in the section "PREFIX" in Chapter 3, and you can include `PREFIX` commands in the `LOGIN` file, as illustrated at the end of this section.

The ProDOS 16 prefix numbers can be used instead of prefixes in pathnames. For example, if you set prefix 7 to /APW/MYPROGS/, you can specify the pathname of /APW/MYPROGS/C.SOURCE/GOODSTUFF as follows:

```
7/C.SOURCE/GOODSTUFF
```

Similarly, you could get a directory listing of the subdirectory /APW/MYPROGS/ by using the following command:

```
CATALOG 7/
```

You can "back up" one directory level from any prefix by using two periods (..) after the prefix name or number. For example, if the system prefix is /HARDISK/APW/SYSTEM, you could use any of the following two commands to edit /HARDISK/APW/MYFILE:

```
EDIT /HARDISK/APW/MYFILE
EDIT /HARDISK/APW/SYSTEM/../MYFILE
EDIT 4/../MYFILE
```

Each time you restart your Apple IIGS, ProDOS 16 retains the volume name of the boot disk. You can use an asterisk (*) in a pathname to specify the boot prefix in some commands. For example, if you booted the Apple IIGS from a disk named /CHOOSER/, and then started APW, you could edit the file /CHOOSER/SYS.UTIL by using the following command:

```
EDIT */SYS.UTIL
```

You cannot change the volume name assigned to the boot prefix except by rebooting the system.

You can put prefix assignments in your LOGIN file. For example, suppose you have all of your APW languages and your program files on a disk named /APW.LANG. The following procedure creates a LOGIN file on your APW disk that sets the language prefix (prefix 5) to the LANGUAGES/ subdirectory on the /APW.LANG disk and that sets prefix 7 to the subdirectory /MYPROGS on that disk.

1. Make sure your APW backup disk is not write protected, put it in your startup disk drive, and boot APW.
2. Type the following commands (press the Return key after each command):

   ```
   EXEC

   EDIT 4/LOGIN
   ```

3. You are now in the editor. If there is already a LOGIN file on your disk, it should be open on the screen; if not, the screen should be blank except for the ruler and status lines at the bottom of the screen. Type the following lines, ending each line with a Return. You can use the arrow keys to move around in the file, and the Delete key to correct mistakes.

   ```
   PREFIX 5 /APW.LANG/LANGUAGES
   PREFIX 7 /APW.LANG/MYPROGS
   ```

4. Press ⌘-Q. When the editor's Quit menu appears, press S to save the file, then E to return to the shell.
5. To test the setup, reboot APW, and then enter the following command:

```
SHOW PREFIX
```

The response should be as follows:

```
System Prefix:

  Number   Name
  *        /APW/
  0        /APW/
  1        /APW/
  2        /APW/LIBRARIES/
  3        /APW/
  4        /APW/SYSTEM/
  5        /APW.LANG/LANGUAGES
  6        /APW/UTILITIES/
  7        /APW.LANG/MYPROGS
```

Using Device Names

ProDOS 16 assigns a device name to each I/O device currently on-line. Use the `SHOW UNITS` command to obtain a list of the device names and the ProDOS volumes currently in those devices.

For example, suppose you have a hard disk attached to a controller board in slot 7, two 800K disks attached to the built-in smart port and a RAM disk on-line. The `SHOW UNITS` command gives the following response (words shown in boldface are the ones you type in):

```
SHOW  UNITS
Units Currently On Line:

Device              Name

.D1                 /HARDISK
.D2                 /APW
.D3                 /RAM5
.D4                 /MYPROGS
.CONSOLE
.PRINTER
```

Only those devices that contain formatted ProDOS disks are shown by the `SHOW UNITS` command. For example, if you removed the disk from your first 800K disk drive and repeated the command, you would get the following response:

```
SHOW  UNITS
Units Currently On Line:

Device              Name

.D1                 /HARDISK
.D3                 /RAM5
.D4                 /MYPROGS
.CONSOLE
.PRINTER
```

You can substitute a device name anywhere you would have used a volume name. For example, to get a directory listing of the subdirectory `/MYPROGS/CSOURCE`, you could use the following command:

```
CATALOG .D4/CSOURCE
```

The names `.CONSOLE` and `.PRINTER` can also be used as device names. The device name `.CONSOLE` represents the keyboard for input and the screen for output. The device name `.PRINTER` can be used to redirect output to a printer. See the section "Redirecting Input and Output" in Chapter 3 for information on redirection.

Using Wildcard Characters

Many of the APW commands allow you to substitite a special character, called a **wildcard character**, for one or more of the characters in a filename. APW recognizes two wildcard characters: the equal sign (=), and the question mark (?). The difference between these two characters is that if you use the question mark, then each time APW finds a match for the character it pauses and asks for confirmation before carrying out the command, whereas if you use the equal sign, APW carries out the operation without asking for confirmation.

For example, suppose you want to write-protect every file in a directory called `/APW/MYFILES`. The command `DISABLE W` *pathname* write-protects the file specified by *pathname*. To write-protect these files, use the following command:

```
DISABLE W /APW/MYFILES/=
```

If you were deleting files rather than write-protecting them, on the other hand, it might be a good idea to double-check each match before letting APW delete it. To delete files in the directory `/APW/MYFILES/` that have the extension `.BKUP`, with APW asking for confirmation before deleting each file, use the following command:

```
DELETE /APW/MYFILES/?.BKUP
```

Each time APW finds a filename in the directory `/APW/MYFILES` that ends in `.BKUP`, it writes the name of the file to the screen. A cursor appears after the filename. To indicate that this file should be deleted, type `Y` (for yes) and press Return. To indicate that the file should not be deleted, type `N` (for no) and press Return. In either case, when you press Return, the shell looks for the next match. If no further matches are found, it deletes the indicated files. To delete the indicated files and quit without looking for the next match, type `Q` (for quit) and press Return. No files are deleted until all matches have been found or until you type `Q` and press Return.

Important: Typing Q does not terminate the command without acting on the selected files. When you type Q and press Return, APW stops looking for new matches to the wildcard filename and acts on all the files for which you have already responded by typing Y. To terminate the command without deleting any files, press Esc.

You can specify as many or as few characters with a wildcard character as you wish. For example, the filename specification MY=ILE would match the names MYFILE, MYBILE and MYOWNFILE. You can use more than one wildcard character in a single filename. For example, =YF?LE would match MYFILE, MARYFILE, and MYFOOLE. You can use both equal signs and question marks in a filename specification, but as long as at least one question mark is present, APW stops and waits for confirmation for every match.

You cannot use wildcard characters for pathnames of directory files or for the directory portion of a pathname (that is, the prefix). In addition, with certain commands you cannot use wildcard characters in filenames at all. For example, you cannot use wildcard characters in the ASSEMBLE command or in the second filename of a RENAME command.

Some commands accept wildcard characters but use only the first filename matched. For example, if you use a wildcard character for the first filename of a RENAME command, only the first file matched is renamed. If you use a question mark (?) in such a case, however, and respond N to the first file matched, then the next match is offered, and so forth until you accept one. The following sequence illustrates this feature. The words shown in boldface are the ones you type in:

```
RENAME  /APW/MY?ILE  /APW/YOURFILE
/APW/MYFILE                    N
/APW/MYBILE                    Y
```

In this example, the file MYFILE is left unchanged, and the filename MYBILE is changed to YOURFILE.

Using Help Files

APW includes a help file for each APW command. To obtain a listing of the APW commands, use the HELP command with no parameters. To display a help file on any command, use the HELP command with the command name as a parameter, as follows:

HELP *command*

Here *command* is the name of the command for which you need help. The help file for each command includes the command syntax, a brief command description, and a list of the required and optional parameters for the command.

The APW help files are all contained in the HELP/ subdirectory in the utilities prefix (prefix 6). Because they are standard ASCII text files, you can edit them if you wish. If you add an alias for a command to the command table, you might want to copy the help file for the command to a file with the alias command name. For example, if you create the alias CMP for the COMPILE command, use the following command to make a help file for CMP:

```
COPY /APW/UTILITIES/HELP/COMPILE /APW/UTLITIES/HELP/CMP
```

After you execute this command, there are two copies of the same help file in the HELP/ subdirectory: one named COMPILE, and one named CMP. You can then edit the CMP file to change the command name in the file from COMPILE to CMP.

Note: The HELP command does not show aliases created with the ALIAS command. Enter ALIAS to list all the aliases currently in effect.

Listing a Directory

To obtain a listing of the files in a directory, use the CATALOG command. For example, to get a listing of the contents of the /APW/ directory, enter

```
CATALOG /APW
```

Note: The CAT command is an alias for the CATLOG command; therefore the command CAT /APW also provides a listing of the contents of the /APW/ directory.

The directory listing for your program subdirectory might look something like Figure 2.1.

```
/APW/MYPROGS/=

Name            Type  Blocks   Modified            Created             Access  Subtype

MYSYSTEM        S16   30         9 NOV 86 09:14    18 SEP 86 13:12     DNB R
ABSPROG         EXE    8        12 APR 86 11:02     4 MAR 86 03:01      NBWR
ABS.SOURCE      SRC    9        13 APR 86 18:18     4 MAR 86 03:19     DNBWR   ASM65816
C.SOURCE        SRC    5        26 MAR 86 07:43    29 FEB 86 12:34     DNBWR   C
COMMAND.FILE    SRC    1         9 APR 86 19:22    31 MAR 86 04 22     DNBWR   EXE
ABS.OBJECT      OBJ    8        12 NOV 86 15:02     4 MAR 86 14:17      NBWR
TEXTFILE        TXT    1        24 DEC 85 24:59    24 DEC 85 11:14     DNBWR

Blocks Free:  1538    Blocks Used:  62    Total Blocks:  1600
```

Figure 2.1. Directory Example

The fields in the directory listing are defined as shown in Table 2.2.

Table 2.2. Fields in a Directory

Field	Meaning
Name	The name of the file. Names are not case sensitive.
Type	The ProDOS 16 file type. ProDOS 16 file types are described in the *Apple IIGS ProDOS 16 Reference* manual. The file types most commonly used in APW are as follows: `DIR` directory file (type $0F) `EXE` load file that runs under a shell program (type $B5) `LIB` library file (type $B2) `OBJ` APW object file (type $B1) `S16` load file that runs independently of any shell program (type $B3) `SRC` APW source file (ProDOS 16 file type $B0) `STR` startup load file (type $B6) `TXT` ASCII text file (type $04) A more complete list of ProDOS 16 file types is given in Table 3.4.
Blocks	The number of blocks on the disk occupied by this file, including the blocks used by the file system. A block is 512 bytes.
Modified	The last date and time at which this file was modified.
Created	The date and time at which this file was first created.
Access	Each of the letters in this list represents one of the ProDOS 16 access privileges, as follows: `D` "Delete" privileges. If you disable this attribute, the file cannot be deleted. `N` "Rename" privileges. If you disable this attribute, the file cannot be renamed. `B` "Backup required" flag. If this attribute is disabled, a backup utility assumes that the file has not been changed since the last time it was backed up. There is no APW command that disables this attribute. `W` "Write" privileges. If you disable this attribute, the file cannot be written to. `R` "Read" privileges. If you disable this attribute, the file cannot be read. Use the `ENABLE` and `DISABLE` commands to set and clear these attributes.
Subtype	For an absolute load file, this field shows the memory address at which the file is loaded when you run it. For an APW source file, this field shows the APW language type.

You can use the `CATALOG` command to get a complete listing of any subdirectory, to get catalog information on an individual file, or, with wildcard characters, to list a specific subset of files on a subdirectory. For example, to list all of the files in the current directory that begin with `MY` and end in `.PAS`, use the following command:

```
CATALOG MY=.PAS
```

You can use device names to list the directory on a volume even if you don't know the name of the volume. For example, to list the files in the second disk drive attached to your system, use the following command:

```
CATALOG .D2
```

To get information about a file named `MYFILE` in the subdirectory `/APW/MYPROGS/`, use the following command:

```
CATALOG /APW/MYPROGS/MYFILE
```

You can also use the Files utility to list directories. The Files utility can list all of the files in a directory, including the contents of all subdirectories; it can list filenames in several columns on the screen; and it can search for filenames that include a string that you specify. The Files utility is described in detail in Chapter 3.

The Editor

The Apple IIGS Programmer's Workshop Editor is a full-screen text editor, with considerable text-manipulation facilities. You can perform the following functions while in the editor:

- delete text
- copy text
- move text
- search for a text string
- search for a text string and automatically replace it with another string
- jump from one position in the file to another
- scroll the screen down or up
- set and clear tab stops
- restore accidentally deleted text
- define and use macros of editor keyboard commands

You control the editor with keyboard commands. All of the editor's features are described in detail in Chapter 4. This section provides a brief introduction to the use of the editor.

Calling the Editor

To call the editor, use the following command:

`EDIT` *pathname*

Here *pathname* is the full or partial pathname of the file you wish to edit. The file you specify in the `EDIT` command is opened; if the file does not already exist on the disk, a new file with that name is opened.

Language Types

Every APW file has an APW language type. If you open a new file, it is given the current APW language type, whereas if you open a preexisting file, APW's current language changes to match the language type of that file.

You can also change the current language by entering as a command the name of the language you wish to use. You can change the APW language type of any existing APW source file with the CHANGE command, described in Chapter 3.

Each language compiler, assembler, interpreter, text formatter, or linker you add to APW has a language name that can be assigned to a file. To get a list of the languages defined in your system, use the command SHOW LANGUAGES. Commonly used APW language types are shown in Table 2.3. A more complete list of currently assigned APW language types is given in Appendix B.

Table 2.3. Commonly Used APW Language Types

Language Type	Meaning
EXEC	An APW command file
TEXT	An ASCII text file (ProDOS 16 file type $B1)
PRODOS	An ASCII text file (ProDOS 16 file type $04)
ASM65816	APW 65816 assembly-language source code
CC	APW C source code
LINKED	APW Linker command file

Opening and Saving a File

Use the following procedure for opening and saving a new file named MYFILE:

1. Enter as a command the language type you want to use for the file by typing the name of the language and pressing Return. For example, if you want to create a C source file, enter

   ```
   CC
   ```

   Note that if the new file is the same language type as the last file edited, you can skip this step. Use the SHOW LANGUAGE command to find out what the current language is set to.

2. Type EDIT MYFILE and press Return. The editor opens a new file, named MYFILE.

3. Press Control-Q or Apple-Q. The editor's Quit menu appears on the screen. Press S to save the file and E to exit the editor.

Using the Editor

The APW Editor allows you to enter and modify source files for all APW programming languages, and to write text files with the APW TEXT language type or with the

ProDOS 16 standard text-file type. The editor provides a full range of editing functions, described in detail in Chapter 4 and summarized in Appendix B. In this section, enough commands are described to get you started using the editor.

When you press the Esc key, the editor enters a special mode called *escape mode*. You can cause a command to be repeated automatically up to 32767 times while in escape mode by typing the number of repetitions after you press the Esc key and before you execute the command. For example, to scroll down 10 lines, type Esc `10` Apple-P. If it is impossible for the editor to repeat the command as many times as you specify, it repeats it the maximum number of times possible.

To exit escape mode, press the Esc key again.

To get started using the editor, use the commands shown in Table 2.4. Note that (as shown in Chapter 4 and Appendix B) there are alternate key combinations for several of these commands; only one key combination is shown for each command here for the sake of simplicity.

> **Note:** Screen movement descriptions in this manual are based on the direction the display screen moves through the file, not the direction the lines appear to move on the screen. For example, if a command description says that the screen scrolls down one line, it means that the lines on the screen move *up* one line, and the next line in the file becomes the bottom line on the screen.

Table 2.4. Basic Editor Commands

Command	Key	Action
Help	⌘-?	Display the editor's help file. Use the Cursor Movement, Top of Screen/Page Up, and Bottom of Screen/Page Down commands to move through the help file. Press Esc, Return, or Enter to return to your file.
Cursor Movement	↑ ↓ ← →	Move the cursor. Use the arrow keys to move the cursor around on the screen.
Top of Screen /Page Up	⌘-↑	Move the cursor to the top of the screen. If the cursor is already at the top of the screen, the entire screen is scrolled up one page (that is, one screen's height).
Bottom of Screen /Page Down	⌘-↓	Move the cursor to the bottom of the screen. If the cursor is already at the bottom of the screen, the entire screen is scrolled down one page.
Toggle Insert Mode	⌘-E	Change the editor to overstrike mode if insert mode is active; change the editor to insert mode if overstrike mode is active. In insert mode, each character you type is inserted in the line of text at the cursor position; any characters to the right of the cursor are pushed to the right to make room. In overstrike mode, each new character replaces the character the cursor is on.
Tab	Tab	Press this key to move the cursor to the next tab stop. If you enter text after pressing Tab, or if you are in insert mode, spaces are inserted in the line up to the tab stop. No tab character (ASCII code $09) is inserted in the file.
Set and Clear Tabs	⌘-Tab	If there is no tab stop at the cursor position, one is added. If there is a tab stop at the cursor position, it is removed. The default locations of tab stops depend on the APW language type as described in the section "Setting Editor Defaults" in Chapter 4.
Scroll Down One Line	⌘-P	Use this command to scroll the screen down one line.
Scroll Up One Line	⌘-O	Use this command to scroll the screen up one line.
Clear	⌘-Delete	After pressing this key combination, use any of the cursor-movement or screen-scroll commands to mark a block of text (all other commands are ignored), then press Return. The selected text is deleted from the file. To cancel the Clear operation without deleting the text from the file, press Esc instead of Return.
Delete Character Left	Delete	Press this key to delete the character to the left of the cursor.
Undo Delete	⌘-Z	This command restores at the cursor position the last text deleted from the file. If the cursor has not been moved, the file is restored to its state before the delete. The undo buffer acts as a stack, so multiple Undo Delete operations are possible. This command does *not* undo a Clear operation.

Table 2.4. Basic Editor Commands (continued)

Copy	↺-C	After pressing this key combination, use cursor-movement or screen-scroll commands to mark a block of text (all other commands are ignored), then press Return. The selected text is written to the file SYSTEMP in the work prefix (prefix 3). (To cancel the Copy operation without writing the block to SYSTEMP, press Esc instead of Return.) Use the Paste command to place the copied material at another position in the file.
Cut	↺-X	After pressing this key combination, use cursor-movement or screen-scroll commands to mark a block of text (all other commands are ignored), then press Return. The selected text is written to the file SYSTEMP in the work prefix and deleted from the file you are editing. (To cancel the Cut operation without cutting the block from the file, press Esc instead of Return.) Use the Paste command to place the cut text at another location in the file.
Paste	↺-V	The contents of the SYSTEMP file are copied to the current cursor position.
Search Down	↺-L	This command allows you to search through a file for a character or string of characters. Enter the search string in response to the prompt at the bottom of the screen. Searches are not case sensitive, and they include all occurrences of the string, whether it is imbedded in a longer string or not. When you press Return, the editor looks from the cursor position toward the end of the file for the search string. If the string is found, the screen is moved so that the next occurrence of the string is on the top line with the cursor placed on the first character of the target string. The search stops at the end of the file.
Search Up	↺-K	This command operates exactly like Search Down, except that the editor looks for the search string starting at the cursor and proceeding toward the beginning of the file. The search stops at the beginning of the file.

Table 2.4. Basic Editor Commands (continued)

Search and Replace Down	⌘-J	This command allows you to search through a file for a character or string of characters and to replace the search string with a replacement string. Enter the search and replace strings in response to the prompts at the bottom of the screen. Searches are not case sensitive, and they include all occurrences of the string, whether it is embedded in a longer string or not. When you enter the Replace string and press Return, the prompt `Auto or Manual (A M Q)?` appears. Type `A` and press Return to cause all occurrences of the search string from the cursor position to the end of the file to be replaced automatically. The cursor returns to the starting point when the replacement is done. If you type `M` and press Return, then when the search string is found, it is highlighted on the top line of the screen and the prompt `Replace (Y N Q)?` appears at the bottom of the screen. Type `Y` to replace the string and search for the next occurrence; `N` to leave this occurrence of the string unchanged and search for the next occurrence; or `Q` to leave the string unchanged and terminate the Search and Replace operation. Press Return to execute the command. When the operation is finished, the cursor returns to its starting point. Type `Q` and press Return in response to the `Auto or Manual` prompt to terminate the Search and Replace operation and return to the file you are editing. When you enter a replacement string and press A or M, the editor looks from the cursor position toward the end of the file for the search string. The search stops at the end of the file.
Search and Replace Up	⌘-H	This command operates exactly like Search and Replace Down, except that the editor looks for the search string starting at the cursor and proceeding toward the beginning of the file. The search stops at the beginning of the file.
Quit	⌘-Q	This command calls the Quit menu, which allows you to save the file, save the file to a new name, open a new file, or quit the editor and return to the shell. See Chapter 4 for a complete description of all the options.

As you become familiar with APW, you can study Chapter 4 to learn the full capabilities of the editor and the fastest way to obtain results. Advanced features described in Chapter 4 include the following:

- Editor macros: a macro allows you to substitute a single keystroke for up to 128 predefined keystrokes. A macro can contain editor commands and text.
- Editing modes : the operation of the editor depends on several modes that can be toggled between different states. Each APW language has a default setting for each mode. You can toggle the modes while in the editor, and you can change the default setting for any language (see the section "Setting Editor Defaults" in Chapter 4).
- Additional commands: in addition to the commands mentioned here, there are several more commands for moving around in the file and manipulating text.

Using a Printer

You can send to a printer any APW output that would normally go to the screen. To redirect output to the printer, use the output redirection operator, >, anywhere on the command line. For example, to send a listing of the directory `/APW/MYPROGS/` to the printer, use the following command:

```
CATALOG /APW/MYPROGS  >.PRINTER
```

See the section "Redirecting Input and Output" in Chapter 3 for more information on sending output to the printer.

You can use this redirection facility together with the `TYPE` command to print out the contents of a text file, as follows:

`TYPE` *pathname* `>.PRINTER`

Here *pathname* is the full or partial pathname, including the filename, of the file you want to type.

Default Printer Settings

By default, APW

- attempts to print from a printer connected to slot 1
- sends a form-feed command to the printer after every 60 lines
- does not add a line feed after a carriage return
- does not count the characters in each line
- does not send an initialization string when you direct output to the printer.

You can use the following commands to override these defaults:

`SET PRINTERSLOT` *slotnum*

`SET PRINTERINIT` *string*

```
SET PRINTERLINES linenum

SET PRINTERLINEFEED value

SET PRINTERCOLUMNS colnum
```

Where:

slotnum The number of the slot containing your printer-driver PC board (an ASCII number from 1–7).

The default value for *slotnum* is 1, the built-in printer port on the Apple IIGS.

Important: If you specify the wrong slot number, the printer initialization string and output data are sent to the wrong slot, with consequences that depend on the device assigned to that slot. For example, the system might hang or reset.

string The initialization string to be sent to your printer each time you send text to the printer. Use this string to set the printer options you want to use, such as character pitch, print quality, line spacing, or boldfacing. Precede a character with a tilde (~) to indicate a control character. Precede a character with a number sign (#) to indicate that the next character should have the most significant bit set. Precede the tilde with a number sign to indicate a control character with the most significant bit set.

To specify the number-sign character ($23), use the sequence ~#. To specify the tilde character ($7E), use the sequence ~~. To specify the tilde character with the most significant bit set ($FE), use the sequence #~~. A space is interpreted as a space character, $20.

Important: The shell does no error checking on the initialization string. If you specify an illegal control character, the shell subtracts $40 from the character and sends it to the printer anyway. For example, if you specify ~g, the shell sends $27 to the printer.

The following command sends the string "Control-L Esc a 2" to the printer:

```
SET PRINTERINIT ~L~[a2
```

For an Apple ImageWriter™ II printer, this string feeds the paper to the next **top-of-form** position and sets the printer to near-letter-quality mode.

The following command sends the sequence $1B $44 $80 $00 to the printer:

```
SET PRINTERINIT ~[D#~@~@
```

For an Apple ImageWriter II printer, this sequence adds an automatic line feed after every carriage return.

See the manual that came with your printer for the options available and the codes necessary to set them.

Important: If you are using a parallel interface card to connect a parallel printer to your Apple IIGS, you must use the initialization string Control-I80N to set the card to 80-column mode and turn off echoing to the screen. To do so, use the following command:

```
SET PRINTERINIT ~I80N
```

linenum An ASCII number indicating the number of lines to be sent to the printer before a form-feed character ($0C) is sent. This command sets the page length. If *linenum* = 0, no form-feed characters are sent. Note that a form feed advances the paper to the next top-of-form position, which corresponds to the top of the next page only if you set up your printer correctly. See the manual that came with your printer for instructions on setting the top of form. (You can usually reset the top of form by turning off the printer, rolling the paper so that the top of the page is slightly above the print head, and turning the printer back on.)

The default for *linenum* is 60.

value If you set *value* to any value (`TRUE` would be appropriate, but any character or string of characters will do), then the printer driver automatically adds a line feed after every carriage return. To cancel this effect, use the `UNSET PRINTERLINEFEED` command.

Depending on the printer you are using and how it is set up, it may or may not automatically add a line feed at the end of each line. If no line feed is added by either the printer or APW, the printer overprints every line of text without advancing the paper. If APW adds a line feed when the printer is adding one too, the lines are double spaced. You can use the `PrinterLineFeed` variable to correct either condition without resetting your printer's DIP switches. If your output looks okay, you don't have to worry about this variable at all.

The default for *value* is null—that is, no line feed is sent.

colnum An ASCII number indicating the number of characters on a line. The printer driver assumes a new line has begun each time *colnum*+1 characters have been printed since the last carriage return. You can set this variable to cause the printer driver to count lines on a page in the case that your printer automatically inserts a carriage return and line feed to wrap lines that are too long.

If your printer stops printing at the end of the line, or returns to the start of the line and overprints the line, then the printer driver can keep track of the lines on the page by counting the number of carriage return characters in the file. In this case you can set *colnum* to 0 and the printer driver will count a new line only when a carriage return is sent.

The default for *colnum* is 0.

Important: If you are using the built-in printer port on the Apple IIGS, you can also use the Control Panel to control a variety of printer interface settings. Make sure that the Control Panel settings and APW printer settings are consistent. For example, if you use the Control Panel to set the line length, you should set *colnum* to the same value to assure that the number of lines on the page are counted correctly in case some lines wrap to the next line. Also, if you set both *value* and the Control Panel to add a line feed after every carriage return, you will get two line feeds (three if the printer is adding one too). See the *Apple IIGS Owner's Guide* for instructions on using the Control Panel.

Including Printer-Setup Commands in the LOGIN File

You can include these commands in a LOGIN file so they are executed each time you load APW. Use the following procedure to create a LOGIN file:

1. Boot APW.
2. Type the following commands (press the Return key after each command):

```
EXEC
EDIT 4/LOGIN
```

3. You are now in the editor. Type the printer-setup commands, one per line, ending each line with a Return. See Table 2.4 for a set of basic editor commands.
4. After the printer-setup commands, type an EXPORT command for each variable. The EXPORT command has the following form:

   EXPORT *variable*

   where *variable* is the name of one of the printer variables you just set. For example, if you used the SET PRINTERSLOT and SET PRINTERINIT commands, you must follow them with the following commands:

```
EXPORT PRINTERSLOT
EXPORT PRINTERINIT
```

5. Press ⌘-Q. When the editor's Quit menu appears, press S to save the file and then press E to return to the shell.
6. To test the setup, first make sure your printer is properly connected to your Apple IIGS as described in the *Apple IIGS Owner's Guide*. Then reboot APW, turn on your printer, and enter the following command:

```
TYPE 4/LOGIN >.PRINTER
```

   The contents of the LOGIN file should be sent to your printer.

Using Exec Files

The shell can accept commands from a command file, called an **Exec file**. To create an Exec file, use the following procedure:

1. Change the currrent language to EXEC by typing EXEC and pressing the Return key.

2. Type `EDIT` *filename*, where *filename* is the name you want to use for the Exec file, and press Return.
3. Type the commands in the file. You can put one command on each line, or you can put several commands on each line, separated by semicolons (;).
4. Press ⌘-Q to quit the editor. Save the file when prompted to do so.

Exec files can include conditional-execution commands (`IF` statements, for example); you can also pass parameters into Exec files. An Exec file can call other Exec files, and it can be set to terminate automatically if a routine it calls returns an error. Exec files and conditional-execution commands are described in the section "Exec Files" in Chapter 3.

To execute an Exec file, type the pathname of the file as if it were an APW command. If you need to pass parameters into the Exec file, list them after the filename, separated by spaces. (Note that the pathname is not case sensitive, but parameter values *are* case sensitive.) For example, if the Exec file had the pathname `/MYPROGS/EXEC.FILES/ANIMALS` and required two animal names as parameters, you could enter the following command to run it:

```
/MYPROGS/EXEC.FILES/ANIMALS dog alligator
```

APW executes each command in the file as if it were typed from the keyboard.

You can also place an Exec file in the `UTILITIES/` subdirectory (prefix 6) and add it to the command table as a utility program. Then you can execute the program just by typing its name on the shell's command line; in this case, the full pathname of the Exec file is not needed. The command table is discussed in the section "Command Types and the Command Table" in Chapter 3.

Exec-file variables, such as parameters passed into the file or those defined with `SET` commands, are normally local to that Exec file (that is, the definitions are not valid in any other Exec file). To use the variables in an Exec file called *by* that file, you must include the variable name in an `EXPORT` command. To use the variables in the Exec file that *calls* the file in which the variables are defined, you must execute the called Exec file with an `EXECUTE` command. The `EXECUTE` command can also be used from a command line to make the variables available at command level. The `EXPORT` and `EXECUTE` commands are described in detail in the section "Exec Files" in Chapter 3.

Compiling (or Assembling) and Linking a Program

The Apple IIGS Programmer's Workshop uses a single format for object files and a single set of commands for compiling or assembling programs written in any APW source language. Therefore, you can write different modules or routines of your program in different APW languages. Creating an executable program using APW is a three- or four-step process, as follows:

1. Write the source code. You can divide your source code among as many files as you wish and can use any combination of APW languages for your program. Each file, however, must consist of source code for only one language. If you are using more than one language, see the manuals that came with your APW languages for

instructions for passing parameters between languages and for examples of multilanguage programs.

2. Compile the source code. The compiler (or assembler) converts the source code into machine-language instructions, data, and symbolic references, and writes the result out as object files. Each source file can yield one or more object files. Because the object files contain symbolic references as opposed to actual memory addresses, they cannot be loaded by the System Loader or executed by ProDOS 16. In addition, some of the references in the object files may be to routines in library files, so that the set of object files does not necessarily represent all of the object code for the program.
3. Link the object and library files. The APW Linker replaces the symbolic references with entries in relocation dictionaries that can be used by the loader to relocate the references at load time. The linker also combines all of the object files and referenced library subroutines into a single load file. The load file still does not contain actual memory addresses, but the relocation dictionaries created by the linker contain all the information the loader needs to load the file.
4. As an optional step, if your load file contains many references that require relocation, you may be able to reduce its size significantly by running the Compact utility program. Compact is discussed in the section "Compacting Your Load File" later in this chapter.

Note: For simplicity's sake, the words *compiler* and *compile* are used in this section to include *assembler* and *assemble*.

This section begins with a short sample program that illustrates a typical sequence for writing, compiling, and linking a program, followed by a discussion of the other features of the commands for compiling and linking files.

See the discussions of the ASML and LINK commands in Chapter 3, the section "Partial Assemblies or Compiles" in Chapter 3, and the section "Using the Advanced Linker" in Chapter 5 for more information on controlling assemblies, compiles, and links.

A Sample Assembly and Link

To get some practice in the use of APW, boot your APW disk and try the following procedure:

1. Set the system language to the language type of the source code you intend to write. We are going to write a simple assembly-language file for this example, so enter the following command:

```
ASM65816
```

2. Set the current prefix to the subdirectory you want to use for your files. If your work disk is called /MYPROGS, for example, enter the following command:

```
PREFIX /MYPROGS
```

3. Open a file for editing. We will call our source file HW. To open an editor file named HW, enter the following command:

```
EDIT HW
```

4. Write the source code for your program. For our example, type in the following program:

```
            KEEP      HELLO                Output filename
            MCOPY     2/AINCLUDE/M16.UTIL       Macro file
MAIN        START                          Beginning of segment
            PHK                            Set data bank equal
            PLB                              to code bank
            WRITELN   #'Hello world!'      Macro that writes string
            LDA       #0                   Set error code to 0
            RTL                            Return to shell
            END                            End of segment
```

5. Press ⌘-Q to quit the Editor. When the Quit menu appears, press S to save the file to disk and then press E to return to the APW Shell command line.

6. To assemble, link, and execute the file HW, enter the following command:

   ```
   RUN HW
   ```

   The words `Hello world!` should be written to the screen following the diagnostic output of the assembler and linker. If not, check your source file for errors and try again.

7. You now have a file on your work disk called HELLO. To execute this program, enter HELLO from the APW Shell command line. (Because it ends in an RTL instruction rather than a ProDOS QUIT call, this program cannot be executed from a finder or chooser program.)

Specifying Names for Output Files

Before we go on to consider the use of APW commands to compile and link programs, we discuss in some detail the various ways in which you can specify the names of output files. If you do not specify the name for an object file, the compiler reads the source file and compiles it, but no object file is written to disk. Similarly, the linker can link a series of object files and library files without writing any load file to disk if you do not specify the name for the load file. There is no default filename for output files unless you set one. Therefore, it is important to understand the various options provided by APW for naming output files before attempting to use any of the commands to compile or link programs.

As noted earlier, the output of a compile or assembly consists of one or more object files. APW compilers and assemblers that are able to perform partial compiles or assemblies often create more than one object file for a given program. Each object file contains some, but often not all, of the object segments that make up that program. It is the job of the linker to extract the most recent version of each object segment from these files and combine them all into a single load file. Object filenames are constructed to make this job easier for the linker: all the object files created from a given source file begin with the same **root filename** and end with distinct filename extensions.

The program MYFILE, for example, after several partial compiles, might include the object files MYFILE.O.ROOT, MYFILE.O.A, MYFILE.O.B, and MYFILE.O.C. The object-file root filename in this case is MYFILE.O. The file with the .ROOT extension contains the first segment to be processed by the linker. The files with alphabetic extensions (.A,

.B, .C) contain other program segments; MYFILE.O.C is the last file created by a partial compile.

In addition to linking files that share the same root filename, the APW Linker can link together object files with different root filenames and library files.

After linking, each program consists of a single load file. The full pathname of the load file can be anything other than the full pathname of the source file or of one of the object files.

Depending on the assembler or compiler you are using, you have either two or three ways to specify the names of object files.

- You can use the APW Shell's command-line KEEP parameter to specify the name of the object file.
- You can specify a default object filename with the KeepName shell variable.
- If your assembler or compiler provides a way to do so, you can specify the root filename of the object files in the source file.

Note: These methods are listed in the order of priority followed by the shell. For example, if you specify different object filenames on the shell's command line and in the source file, the command-line name is used. On the other hand, any specific compiler might not support one or more of these methods.

Since there is only one load file per program, the APW Linker does not append any extensions to load filenames. As with object filenames, there are several ways to specify a load filename, as follows:

- You can use the APW Shell's command-line KEEP parameter to specify the name of the load file.
- You can specify a default load filename for the LINK command with the LinkName shell variable. You can specify a default load filename for a LinkEd file with the KeepName shell variable.
- You can specify the load filename by calling the advanced linker and specifying a KEEP command in the LinkEd file.
- You use a shell command (such as ASML) that automatically calls the linker after a successful compile. In this case, the root filename used for the first object file is also used as the load filename.

Note: These methods are listed in the order of priority followed by the shell. For example, if you specify different load filenames on the shell's command line and in the LinkEd file, the command-line name is used.

Specifying the Object Filename on a Shell Command Line

You can use the APW Shell's command-line KEEP parameter to specify the name of the object file. For example, to compile MYFILE and write the object files to files with the root filename MYFILE.O, you can use the following command:

```
ASSEMBLE MYFILE KEEP=MYFILE.O
```

In order to use the KEEP parameter when you specify multiple source filenames on the command line, you must use a wildcard character in the filename. Two wildcard characters are available for this purpose: % and $. When you use the percent sign (%) wildcard, the shell replaces it with the source filename. When you use the dollar sign ($) wildcard, the shell removes the last extension from the source filename and replaces the dollar sign with the resulting filename.

For an example of the use of the percent sign wildcard, assume you execute the following command:

```
COMPILE MYFILE YURFILE KEEP =%.O
```

The shell uses the name MYFILE.O.ROOT for the first object file created from the source file MYFILE and the name YURFILE.O.ROOT for the first object file created from the source file YURFILE.

For an example of the use of the dollar sign wildcard, assume you execute the following command:

```
COMPILE MYFILE.CC YURFILE.ASM KEEP=$
```

In this case, the shell uses the name MYFILE.ROOT for the first object file created from the source file MYFILE.CC and the name YURFILE.ROOT for the first object file created from the source file YURFILE.ASM.

> **Important:** Because ProDOS 16 does not allow filenames longer than 15 characters, you must be careful not to specify a filename in the KEEP parameter that will result in an output filename longer than 15 characters. For example, if you specify KEEP=%.OUT and the source filename is LONGNAME, the compile will fail when the shell tries to open the file LONGNAME.OUT.ROOT, which has 17 characters.

If you specify both a KEEP directive in the source file and a KEEP parameter on the command line, the command-line parameter takes precedence.

Specifying a Default Object Filename With the KeepName Variable

If you do not use the KEEP parameter, the compiler looks for a KeepName shell variable to determine the default output filename. To specify a default filename, use the following commands (replace *value* with the output filename you want to use):

```
SET KEEPNAME value
EXPORT KEEPNAME
```

The SET command specifies the value for the KeepName variable. The EXPORT command makes that value available in Exec files.

The KeepName variable can include the wildcard characters % and $. As for the KEEP parameter, the percent sign (%) is replaced with the source filename and the dollar sign ($) is replaced with the source filename with the last extension removed. You must be careful

not to specify a combination of a KeepName variable and a source filename that will result in an output filename longer than 15 characters, or the compile will fail.

You can include a definition for the KeepName shell variable in your LOGIN file. For example, to set KeepName to %.O, so that the default output filename is the source filename with the extension .O, put the following lines in your LOGIN file:

```
SET KEEPNAME %.O
EXPORT KEEPNAME
```

Note that in this case the EXPORT command is required to make the value of KeepName available on the command line as well as in Exec files.

Specifying the Object Filename in the Source File

If your assembler or compiler provides a way to do so, you can specify the root filename of the object files in the source file. If you are using the APW Assembler, for example, put a KEEP directive at the beginning of the source file.

If you have linked several source files together (such as with APPEND directives in assembly language or #append directives in C), the output filename you specify at the beginning of the first source file is used as the root filename for all the object files, even if the files are not all in the same language.

Depending on the compiler you are using, the root filename specified in the source file may be overridden by the filename in the KEEP parameter and by the default filename set by the KeepName variable.

Specifying the Load Filename on a Shell Command Line

To specify names for load files, you use methods similar to those used to specify root names for object files.

As for object-file root names, you can use the APW Shell's command-line KEEP parameter to specify the name of the load file. For example, to compile MYFILE, link the object files, and write a load file with the filename MYFILE.O, you can use the following command:

```
ASML MYFILE KEEP=MYFILE.O
```

If you specify multiple source filenames on the command line, you must use a wildcard character in the filename as described in the section "Specifying the Object Filename on a Shell Command Line" earlier in this chapter. For example, assume you execute the folllowing command:

```
CMPL MYFILE YURFILE KEEP =%.O
```

The shell uses the name MYFILE.O.ROOT for the first object file created from the source file MYFILE and the name YURFILE.O.ROOT for the first object file created from the source file YURFILE. It uses the name MYFILE.O for the load file.

If you perform separate compiles and links, you can specify the load filename independently of the object filename. For example, you could perform a compile and link with the following commands:

```
COMPILE MYFILE YURFILE KEEP =%.O
LINK MYFILE.O YURFILE.O KEEP=MYLOAD
```

The result is identical to that for the CMPL command, except that the load file is named MYLOAD instead of MYFILE.O. Note that the LINK command requires only the root filenames of the object files; the linker automatically links all files that share the same root filename.

You can also specify a KEEP parameter on an ALINK command line. ALINK executes a LinkEd command file. If you specify a KEEP parameter that includes a wildcard character on an ALINK command line, the name of the LinkEd file (rather than the root name of the object file) is used to create the load filename.

The LINK and ALINK commands are discussed further in the section "Linking Your Program: The LINK and ALINK Commands" later in this chapter.

Specifying a Default Load Filename With a Shell Variable

If you don't use the KEEP parameter with the LINK command, the shell checks to see if you have specified a default output filename with the LinkName shell variable. To specify a default filename, use the following commands (replace *value* with the output filename you want to use):

```
SET LINKNAME value
EXPORT LINKNAME
```

The SET command specifies the value for the LinkName variable. The EXPORT command makes that value available in Exec files.

The LinkName variable can include the wildcard characters % and $. When you use the percent sign (%) wildcard, the shell replaces it with the object file's root filename. When you use the dollar sign ($) wildcard, the shell removes the last extension from the root filename and replaces the dollar sign with the resulting filename. For example, if LinkName is set to %.O and you execute the command LINK MYFILE, the shell uses the name MYFILE.O for the load file. Similarly, if LinkName is set to $ and you execute the command LINK MYFILE.CC, the shell uses the name MYFILE for the load file.

> **Important:** Because ProDOS 16 does not allow filenames longer than 15 characters, you must be careful not to specify a root filename-LinkName combination that will result in a load filename longer than 15 characters. For example, if LinkName is set to %.LOADFILE and the root name is LONGNAME, the link will fail when the shell tries to open the file LONGNAME.LOADFILE, which has 17 characters.

You can include a definition for the LinkName shell variable in your LOGIN file. For example, to set LinkName to %.O, so that the default output filename is the object root filename with the extension .O, put the following lines in your LOGIN file:

```
SET LINKNAME %.O
EXPORT LINKNAME
```

Note that in this case the EXPORT command is required to make the value of LinkName available on the command line as well as in Exec files.

> **Note:** The LinkName variable can be used only with the LINK command. If you are using a LinkEd file, you can use the KeepName variable to set a default load filename. The ASML and ASMLG commands (and their aliases) use the root filename of the first object file as the load filename, as discussed in the section "Using the Object-File Root Filename for the Load Filename," later in this chapter.

Specifying the Load Filename in a LinkEd File

If you perform a separate link by using a LinkEd file instead of the LINK command, you can use the LinkEd KEEP command to specify the load filename. There are three ways to execute a LinkEd file:

- You can execute the LinkEd file separately from the compile with the shell's ALINK command.
- You can append the LinkEd file to your source code (using an APPEND directive) and compile and link the program with one COMPILE or ASSEMBLE command.
- You can name the LinkEd file as the last source file on the COMPILE or ASSEMBLE command line.

If you specify both a KEEP command in the LinkEd file and a KEEP parameter on the command line, the command-line parameter takes precedence.

LinkEd and the LinkEd KEEP command are described in the section "Using the Advanced Linker" in Chapter 5. The ALINK command is discussed further in the section "Linking Your Program: The LINK and ALINK Commands" later in this chapter.

> **Note:** You cannot execute a LinkEd file with a LINK command. Use the ALINK, ASSEMBLE, or COMPILE commands to execute a LinkEd file.

Using the Object-File Root Filename for the Load Filename

The ASML, CMPL, ASMLG, CMPLG, and RUN commands automatically call the standard linker after a successful compile. If you use one of these commands, the root filename used for the first object file created is also used as the load filename. For example, the following command assembles the file MYFILE, links the resulting object files, and then runs the program:

```
RUN MYFILE KEEP=MYFILE.O
```

The first object file has the filename MYFILE.O.ROOT. The linker links all the object files with the root filename MYFILE.O and creates the load file MYFILE.O. After successful execution of this command, then, the following files will be on the disk:

`MYFILE`	source code
`MYFILE.O.ROOT`	first object file
`MYFILE.O.A`	second object file
`MYFILE.O`	load file

This relationship between the object file root filename and the load filename holds whether you used a directive in the source file, the `KEEP` parameter on the command line, or the `KeepName` shell variable to specify the output filename.

> **Important:** Because the shell will not let you overwrite a source file with a load file, you cannot set `KeepName` to `%` and use it with a link. For example, if `KeepName` is set to `%` and you try to execute the command `CMPL MYFILE`, the link will fail when the linker tries to write a load file named `MYFILE`. Similarly, the command `CMPL MYFILE KEEP=MYFILE` will fail when the linker tries to overwrite the source file `MYFILE` with a load file of the same name.

Assume, for example, that you execute the following command to compile and link three files, two in C and one in assembly language:

```
CMPL MYFILE1.CC MYFILE2.CC MYFILE3.ASM
```

Assume further that you set the `KeepName` variable to `$`. After this `CMPL` command has been successfully executed, the following files are on the disk:

`MYFILE1.CC`	C source code
`MYFILE2.CC`	C source code
`MYFILE3.ASM`	ASM65816 source code
`MYFILE1.ROOT`	object file from the first C source file
`MYFILE2.ROOT`	object file from the second C source file
`MYFILE3.ROOT`	first object file from the assembler source file
`MYFILE3.A`	second object file from the assembler source file
`MYFILE1`	load file

Notice that the root filename of the first object file created is used as the load filename.

Specifying the File Type of Your Load File

By default, load files created by the APW Linker have ProDOS 16 file type $B5 (shell load file). You can change the file type of any file with the shell's `FILETYPE` command, which is described in the section "Command Descriptions" in Chapter 3. You can also change the default file type created by the linker. To do so, use the following commands:

```
SET KEEPTYPE type
EXPORT KEEPTYPE
```

For *type*, substitute the hexadecimal file type that you want assigned to load files by the linker. For example, to cause the linker to create files of type $B3 (ProDOS 16 system load files), use the following command:

```
SET KEEPTYPE $B3
```

Valid file types for load files are $B3 through $BE as shown in Table 2.5

Table 2.5. Load File Types

Value	Abbreviation	File Type
$B3	S16	ProDOS 16 system load
$B4	RTL	Run-time library
$B5	EXE	Shell load
$B6	STR	Startup load
$B8	NDA	New desk accessory
$B9	CDA	Classic desk accessory
$BA	TOL	Tool set file

You can include a definition for the `KeepType` shell variable in your `LOGIN` file. To set `KeepType` to $B3, for example, so that the default file type created by the linker is a system load file, put the following lines in your `LOGIN` file:

```
SET KEEPTYPE $B3
EXPORT KEEPTYPE
```

Shell Commands for Assembling, Compiling, and Linking

The APW Shell provides several commands for assembling, compiling, and linking programs. These commands are powerful and versatile, and therefore somewhat complex. There are many ways of assembling, linking, and running a program in APW; the method you use will depend on your needs and personal preferences. In this section, we explore some of the functions performed by the shell commands for compiling and linking files.

All of the shell commands are described in detail in the section "Command Descriptions" in Chapter 3. Examples of the use of these commands are given in the section "A Sample Assembly and Link" earlier in this chapter.

The following APW Shell commands assemble or compile a program and then return control to the shell:

- `ASSEMBLE`
- `COMPILE`

The following commands first assemble or compile a program, then call the linker, and then return control to the shell:

- `ASML`
- `CMPL`

The following commands assemble or compile a program, call the linker, run the program, and then return control to the shell:

- ASMLG
- CMPLG
- RUN

The following commands call the linker and then return control to the shell:

- ALINK
- LINK

The ASSEMBLE and COMPILE Commands

The ASSEMBLE and COMPILE commands are identical; that is, they are **aliases** for the same command. In the simplest case, you need name only the source file on the command line. The shell determines the language type of the source file and calls the appropriate assembler or compiler. For example, to assemble or compile the file MYFILE, you can use either of the following commands:

```
ASSEMBLE MYFILE
COMPILE MYFILE
```

As before, for simplicity's sake we use the terms *compiler* and *compile* to include *assembler* and *assemble*.

The complete syntax of these commands is as follows:

```
COMPILE  [+E|-E]  [+L|-L]  [+S|-S]  [+T|-T]  [+W|-W]
         file1  [file2 ]  [...]  [KEEP=outfile]
         [NAMES=(seg1  [seg2]  [...])] [language1=(option ...)
         [language2=(option ...) ]  [...] ]
```

Square brackets indicate optional parameters. Italics indicate variables that must be replaced with specific values. The vertical bar (|) indicates a choice. See the section "Command Descriptions" in Chapter 3 for a complete discussion of command syntax.

In the following sections, we explain each of the components of these commands. Since the ASSEMBLE and COMPILE commands perform a single main function (that is, an assembly or compile), they are a bit simpler to explain than most of the other commands listed above. Therefore, we explore these commands in some detail before going on to discuss some of the extra nuances introduced by the other commands.

Diagnostic Output: the L and S Options

By default, the only output of a compile written to the screen is the name of the compiler or assembler called, plus possibly the name of the segment being processed. If any errors occur, they are also written to the screen. The ASSEMBLE and COMPILE commands provide parameters to control the output of source listings (the L parameter) and symbol tables (S) for those APW compilers that support such output.

The source listing shows the contents of the source file, with each line preceded by the line number used by the compiler while it processes the file. The line number is used in error messages and symbol-table listings, for example. The contents of a symbol-table listing, if any, depend on the compiler you are using; see the manual that came with your compiler for a description. Typically, the symbol table provides a cross-reference of symbolic references in the file and the numbers of the lines on which they are defined.

Since the default is to *not* print the source listing or symbol table, you will usually use the +L and +S forms of these parameters; these forms enable the listings. However, since your compiler might also let you control these listings from the source file, the shell also provides the -L and -S forms of these parameters; these forms override the source-file directives and turn the listings off.

The L and S parameters must be placed *before* the name of the first source-code file on the command line. For example, to assemble the file MYFILE and write the source-code listing on the screen as the assembly proceeds, you could use the following command:

```
ASSEMBLE +L MYFILE
```

Error Handling: the E, T, and W Options

Under APW, there are two fundamental types of errors: fatal and nonfatal. A fatal error is one that precludes any further processing by the compiler or linker that is currently executing. In the case of a fatal error, processing stops immediately. When a nonfatal error occurs, processing can continue, but the file that is created almost certainly contains bugs. In the case of a combined compile and link (such as for the ASML command), the compiler can pass control to the linker or can tell the shell to terminate execution of the command when the compile is finished, depending on the severity of the error. The error messages generated by the compiler or linker can help you diagnose the cause of the error.

Linker errors and error levels are described in Appendix C. If you are writing a compiler or linker, see the section "Entry and Exit" in Chapter 6.

The APW Shell provides several options that affect the way errors are handled. For compilers and linkers that support these options, you can control some aspects of APW's error handling with the E, T, and W command-line options, as follows:

> **Note:** Although APW defines conventions for handling errors and provides shell command-line options that affect error handling, any individual compiler or linker may or may not follow these conventions. See the manual that came with the compiler or linker that you're using to see which of these options are valid for that program.

- If you specify +E, when the compiler terminates execution due to a fatal error, it calls the APW Editor. The editor displays the source file with the offending line on the fourth line on the screen (or as far down on the screen as possible, if the error is in one of the first three lines of the file). If you specify -E and a fatal error occurs, you are returned to the shell's command line or the Exec file that executed the command. The default for this option is +E when the command is executed from the command line and -E when the command is executed from an Exec file.

- If you select +T, any error causes the compile to terminate. If you select both +T and +E, an error causes the shell to call the APW Editor and display the offending line as the fourth line on the screen. If you omit this option or select -T, only fatal errors cause immediate termination of the compile.
- If you select +W, the compiler stops and waits for a key press when any error occurs, to give you the opportunity to read the error message and to decide whether to continue (that is, to continue the compile in case of a nonfatal error or to call the editor in case of a fatal error). Press Apple-Period (⌘-.) to halt execution, or press any character key or the spacebar to continue. If you omit this option or select -W, execution continues without pausing when an error occurs.

Specifying Source Files

In APW, you are not required to have all of your source code in a single file. You can use a separate command to compile each source file, or you can compile several source files with a single command. There are two ways to use a single command to compile multiple source files:

- You can append one source file to the end of another, using directives in the source files (if the language you are using provides such a directive). In APW Assembler, for example, you can use APPEND directives; in APW C you can use #append directives.

  If the appended file is in the same language as the file being processed, the compiler treats the appended code as if it were in the file it is already processing. In this case, the compiler continues processing the program without a break.

  If, on the other hand, the appended file is in a different language than the file being processed, the compiler returns control to the shell, which calls the appropriate compiler for the new file.
- You can name more than one source file in the command line. For example, to compile files MYFILE1 and MYFILE2, you can use the following command:

  ```
  COMPILE MYFILE1 MYFILE2
  ```

  In this case, regardless of the language type of MYFILE2, when the compiler finishes processing MYFILE1, it returns control to the shell. The shell then calls the appropriate compiler for MYFILE2, which opens that file and compiles it.

The KEEP Parameter

If you have named an output file in a directive in the source file or if you have set the KeepName shell variable (see the section, "Specifying Names for Output Files" earlier in this chapter), MYFILE is compiled and, if successful, one or more object files are written to disk. If you have not previously specified an output filename, you can do so on the command line with the KEEP parameter. For example, to compile MYFILE and write the object files to files with the root filename MYFILE.O, you can use the following command:

```
ASSEMBLE MYFILE KEEP=MYFILE.O
```

The first object file is named MYFILE.O.ROOT. A second file named MYFILE.O.A may also be written, depending on the assembler or compiler you are using and the number of segments in the source file.

If you have listed more than one source file on the command line, you must use one or more wildcard characters in the KEEP parameter, as described in the section "Specifying the Object Filename on a Shell Command Line" earlier in this chapter.

The NAMES Parameter

The NAMES parameter is provided for those APW compilers that can do partial compiles. A partial compile involves recompiling one or more, but not all, of the segments in a program after at least one full compile has already been done. When you link the program, the linker automatically selects only the latest version of each segment.

The purpose of partial compiles is to speed up the development process. Suppose you have written an assembly-language program that includes 25 segments, have assembled the program, and have discovered bugs in two of the segments. With the APW Assembler, you can correct the two problem segments and then reassemble those segments only; there is no need to reassemble the segments that were not changed.

To perform a partial compile, you use the NAMES parameter on the command line to specify the segments to be compiled. For example, to do a partial assembly of the file MYFILE, reassembling the segments DICK and JANE, you could use the following command:

```
ASSEMBLE MYFILE NAMES=(DICK JANE)
```

Important: There are three circumstances in which you cannot perform a partial compile, but must do a full compile instead, as follows:

- When you delete or rename a segment. If you do a partial compile in this case, the linker will not know that the old version of the segment is no longer valid and will link it into your program.
- When the order in which the segments are linked is significant.
- When you change the definition of a global symbol. References to global symbols within each file are resolved by the compiler, so if you change the definition of a global symbol you must be sure that you recompile every segment in which that symbol is used. The best way to make sure you have caught every occurance of the symbol is to do a full compile.

Partial assemblies and compiles are discussed in detail in the section "Partial Assemblies or Compiles" in Chapter 3.

Language-Specific Parameters

Since each compiler or assembler operating under APW has its own requirements and abilities, the APW Shell allows you to pass parameters directly to compilers. To do this, you include the language name of the compiler on the command line, followed by an equal sign and a list of options in parentheses. The APW Shell does not do any error checking on the options string; it merely sends it on to the compiler.

For example, to pass the -D parameter (which assigns a definition to a symbol) to a C program named MYFILE, you might use the following command:

```
COMPILE MYFILE CC=(-Dlucky=13)
```

Note that the language name of C is CC. The language names of APW languages are listed in Appendix B.

Linking Your Program: The LINK and ALINK Commands

After you have created one or more object files by compiling your program, you must call the APW Linker to create an executable load file. While there are several APW commands that automatically call the standard linker, we first consider a separate link in order to make the relationships between object files, library files, and load files clearer.

How much attention you must focus on the link process depends on the sort of programming you are doing. If you are writing simple programs or utilities using one or at most two or three source files, the standard linker with few or no parameters is adequate for your use. On the other hand, if you want to select specific segments out of object files, assign object segments to load segments during the link process, or control the diagnostic output of the linker during the link process, then you must use the advanced linker. In between these two extremes are command-line options that let you search nonstandard library files, link object files with different root filenames, and turn on or off the linker's diagnostic output.

> **Note:** You may have to use the advanced linker to make a segment dynamic. Refer to the manual that came with the APW language you are using to see if there is a way to assign dynamic segments in the source code.

For example, suppose you have compiled a program named MYFILE and created the object files MYFILE.O.ROOT and MYFILE.O.A. To link this program, you could use the following command:

```
LINK MYFILE.O KEEP=MYFILE.O
```

Assuming that there are no source files in the directory named MYFILE.O, the linker links the object files MYFILE.O.ROOT and MYFILE.O.A, creating the load file MYFILE.O. If, after processing MYFILE.O.ROOT and MYFILE.O.A, there were any unresolved references, the linker would automatically search any library files in the library prefix, prefix 2. When it finds the segment it needs in a library file, the linker extracts only that segment and links it into the program.

This program, in fact, would be an ideal candidate for use with one of the commands that automatically links the program after the compile. These commands are discussed in the next section.

Now assume that your program has been compiled into two sets of object files, with root filenames MYFILE.O and MYTFINE.O. Assume further that you have created your own library file called MYLIB that you want to search as part of the link. To link this program, you could use the following command:

```
LINK MYFILE.O MYTFINE.O MYLIB KEEP=MYPROG
```

The linker first links MYFILE.O, then links MYTFINE.O, and then (if there are any unresolved references) searches MYLIB. Finally, if there are still unresolved references, the linker searches the library files in the library prefix. It creates a load file named MYPROG.

Note that you have considerable control over the link with this command. The linker links object files and searches libraries in the sequence in which you list them. For example, suppose you had used the following command to link this program:

```
LINK MYTFINE.O MYLIB MYFILE.O KEEP=MYPROG
```

In this case, the linker would first link MYTFINE.O, then search MYLIB, and then link MYFILE.O. Finally, if there were still unresolved references, the linker would search the library files in the library prefix. As before, it would create a load file named MYPROG.

The LINK command takes the L and S parameters as well. The +L parameter generates a **link map**, which is a listing of the segments in the object file with the starting address, length in bytes, and segment type of each segment. The +S parameter generates a symbol-table listing of all the global references in the object file. The default for these parameters is to generate no listings; since there is no way to turn these listings on in the object files, the -L and -S parameters have no real function in the LINK command. You may wish to use -L and -S in Exec files, however, just to remind yourself that the command will generate no diagnostic output other than error messages.

If you need complete control over the link, you must use the advanced linker. The advanced linker is controlled by executing a LinkEd file. To execute a LinkEd file in a separate link, use the ALINK command. Actually, the ALINK command is an alias for the ASSEMBLE and COMPILE commands. A LinkEd file is treated by APW like any other source file, except that when the shell processes a LinkEd file, it calls the advanced linker instead of calling a compiler or assembler.

Although ALINK is an alias for ASSEMBLE, note that several of the parameters and options provided by APW for the ASSEMBLE command make no sense for a LinkEd file. Since only one link is done for each program, for example, listing two LinkEd files on the ALINK command line would cause two separate links to take place, not one link of two files. Because there is no such thing as a partial link, the NAMES parameter is not useful for LinkEd files, and because you cannot append a source file in any other language to a LinkEd file, the language-specific parameters have no function on an ALINK command line.

The S and L parameters operate for ALINK exactly as they do for LINK, except that, because LinkEd files can include commands to turn the diagnostic output on, the -S and -L parameters are useful in the ALINK command to override those commands and turn the output off.

Because LinkEd files are treated by the APW Shell like language files, they can be executed in all the ways that languages can. That means they can be appended to the last source file processed or listed as the last language source file on an ASSEMBLE or COMPILE command line. Note that the LinkEd file must be the *last* file processed, because no more source files will be compiled after the link.

The advanced linker is described in detail in the section "Using the Advanced Linker" in Chapter 5.

Compiling and Linking : ASML, ASMLG, CMPL, CMPLG, and RUN

There are several APW commands that automatically call the standard linker after a successful compile or assembly: ASML and CMPL compile and link the program; ASMLG, CMPLG, and RUN compile, link, and run the program. Actually, there are only two distinct commands: ASML and ASMLG. CMPL is an alias for ASML, while CMPLG and RUN are aliases for ASMLG. Furthermore, ASMLG is identical to ASML except that after a successful compile and link, ASML returns control to the shell whereas ASMLG runs the program. Therefore, the command ASML is used in this section as representative of this entire set of commands.

All of these commands take the same parameters as the ASSEMBLE and COMPILE commands. Because the linker is called automatically, however, output files include the load file in addition to object files, and input files can include object files and library files in addition to source files. The section "Specifying Names for Output Files," earlier in this chapter, presents the various ways you can determine the names of output files. In this section we explain how you can control the compile and link process by naming input files.

When the shell executes the ASML command, it uses the following procedure:

1. The shell looks for a source file with the filename of the first input file listed on the command line. If there is no source file by that name, the shell assumes it is an object file or a library file and goes on to the next filename.
2. The shell calls the compiler that corresponds to the APW language type of the first source file it finds on the command line.
3. The shell passes to the compiler the option flags (if any), the KEEP filename (if any), the NAMES segment list, and any language-specific options that match the language type of the file.
4. The compiler processes the source file and any source files of the same language type appended to that source file. If the compiler comes to an appended file of a different language type, it goes to step 5. If it has come to the end of the last appended file, it goes to step 6.
5. The compiler returns control to the shell, which calls the compiler that corresponds to the language type of that file. Return to to step 3.
6. The compiler returns control to the shell, which looks for the next source filename on the command line. If it finds another source filename, it returns to step 3. If there are no more source filenames, the shell goes on to step 7.
7. The shell calls the linker and passes to it the option flags (if any), any KEEP filename, and the list of input files from the command line.
8. The linker looks for an object file that corresponds to the first filename on the command line. If the first filename was a source file, the linker looks for the object file created from that source file. If the first filename was an object-file root filename, that object file is processed first.
9. The linker processes all of the remaining files in the sequence in which the filenames are listed. If a source filename is listed, the linker looks for the object files created

from that source file. If an object-file root filename is listed, all the files with that root filename are linked. If a library filename is listed, that file is searched for any unresolved references.

10. If there are still any unresolved references, the linker searches the library files in the library prefix.
11. The linker writes the load file to disk and returns control to the shell. (In the case of the ASMLG, CMPLG, and RUN commands, the shell executes the load file.) The name of the load file is the name used in the KEEP parameter on the command line, if any. If there was no KEEP parameter, the root filename of the first object file created is used as the load filename.

For example, suppose you have a program consisting of source files MYFILE and MYTFINE, plus object files START.ROOT, GDAY.ROOT, and GDAY.A. In addition to the system libraries, you want to search the library file MYLIB, but you want to search that file *before* GDAY is linked. Assume that the user has set the KeepName shell variable to %.O. The first segment of the program is in START.ROOT. To compile, link, and run this program, you can use the following command:

```
CMPLG START MYFILE MYTFINE MYLIB GDAY
```

The program is processed in the following sequence:

1. The shell looks for a source file with the filename START. Since there is no source file by that name, the shell assumes it is an object file or a library file and goes on to the next filename. It is important to note that, if there *were* a source file by that name, it would be compiled and the file START.ROOT would be overwritten.
2. The shell calls the compiler that corresponds to the APW language type of the first source file it finds on the command line—in this case, MYFILE.
3. The compiler processes MYFILE, including any source files of the same language type appended to MYFILE. The resulting object files are written to the disk with the filenames MYFILE.O.ROOT and MYFILE.O.A. Assume that the compiler finds no appended source files of a different language type; it comes to the end of the last appended file and returns control to the shell.
4. The shell looks for the next source filename on the command line—in this case, MYTFINE.
5. The compiler processes MYTFINE, including any source files of the same language type appended to MYTFINE. The resulting object files are written to the disk with the filenames MYTFINE.O.ROOT and MYTFINE.O.A. Assume that this time the compiler finds a source file of a different language type appended to MYTFINE, called MYOTHER; in this case, the compiler returns control to the shell, which calls the compiler that corresponds to the language type of the appended file.
6. The new compiler processes the appended file (MYOTHER). The resulting object file is written to the disk with the filename MYTFINE.O.B. When the compiler comes to the end of the last appended file, it returns control to the shell.
7. The shell can find no more source files, so it calls the linker and passes to it the list of input files from the command line, with the appropriate object-file root filenames substituted for the source filenames. For example, the name MYFILE.O is passed

to the linker instead of MYFILE. Since the first object file created had the root filename MYFILE.O, that name is passed to the linker as the KEEP filename.

8. The linker looks for an object file that corresponds to the first filename on the command line, START. It finds the file START.ROOT and links it.
9. The linker processes MYFILE.O.ROOT, MYFILE.O.A, MYTFINE.O.ROOT, MYTFINE.O.A, and MYTFINE.O.B.
10. Assuming there are some unresolved references, the linker searches MYLIB.
11. The linker links GDAY.ROOT and GDAY.A
12. If there are still any unresolved references, the linker searches the library files in the library prefix.
13. The linker writes the load file to disk with the load filename MYFILE.O and returns control to the shell.
14. The shell executes MYFILE.O.

Before executing the CMPLG command, the following files were in your directory:

START.ROOT	object file containing first segment of program
MYFILE	source file
MYTFINE	source file
MYOTHER	source file
MYLIB	library file
GDAY.ROOT	object file
GDAY.A	object file

After executing the CMPLG command, you have the following files in the directory:

START.ROOT	object file containing first segment of program
MYFILE	source file
MYTFINE	source file
MYOTHER	source file
MYLIB	library file
GDAY.ROOT	object file
GDAY.A	object file
MYFILE.O.ROOT	object file from MYFILE
MYFILE.O.A	object file from MYFILE
MYTFINE.O.ROOT	object file from MYTFINE
MYTFINE.O.A	object file from MYTFINE
MYTFINE.O.B	object file from file appended to MYTFINE
MYFILE.O	load file

In summary, each source file listed on the command line is first compiled independently, then the linker is called just as if the LINK command had been used. The compilers ignore object files and library files. The shell replaces each source filename with the root filename of the object files created from that source file, and then the entire list of filenames is sent to the linker. All command-line parameters that would be passed by the COMPILE command are passed to each compiler called. All parameters that would be passed by the LINK command are passed to the linker.

The ASML and related commands let you combine the functions of the COMPILE and LINK commands in one line. In the case of a fairly simple program, with one source file and no custom library files, the ASML command makes the link process nearly invisible. You can compile and link the program with one command, without worrying about which object files to link or which library files to search. In the case of a complex program for which you want to compile some source files and link them to some files already compiled (that is, to some object files), and perhaps search some custom library files while you're at it, the ASML command gives you the power to do so. For even more control over the link process, you can use a LinkEd command file. The cost of increased versatility and power, however, is increased complexity.

Compacting Your Load File

As a final step in program development, you can run the Compact utility program. Compact converts a load file to the most compact form provided by the object module format. To compact the load file MYPROG and create the compacted load file MYPROG.CMPCT, for example, use the following command:

```
COMPACT MYPROG MYPROG.CMPCT
```

Compacted load files take up less space on disk and load faster than noncompacted load files. In addition, Compact can be used to help make programs restartable. See the section "Command Types and the Command Table" in Chapter 3 for a discussion of restartability.

Not all load files are significantly improved by compacting, however, so you may want to test both a compacted and noncompacted version of your program before releasing it.

> **Important:** In order to load a compacted load file, you must have version 1.2 or later of the system loader on your boot disk.

Launching Programs

Under ProDOS 16 on the Apple IIGS computer there are two principal types of executable load files: system load files (file type $B3) and shell load files (file type $B5). After you have written a program, you can use the FILETYPE command (described in Chapter 3) to assign a file type to it.

System load files take over complete control of the computer; they do not operate under a shell program. APW itself is an example of such a program. To execute a system load file, the calling program (such as a finder or chooser) executes a ProDOS 16 QUIT call, shutting itself down. When the called program finishes and executes a QUIT call, ProDOS 16 normally relaunches the calling program. (For a more complete description of the QUIT

call, see the *Apple IIGS ProDOS 16 Reference*.) A system load file must make Apple IIGS tool calls to set up the environment it needs, including the graphics or text screen it needs and the input it accepts.

Shell load files run under a shell program (such as the APW Shell); they do not remove the shell from memory. The shell uses System Loader calls to load the program, and then transfers control to it. When the program terminates, it returns control to the shell. A shell load file uses the environment set up for it by the shell under which it runs. For more information on writing a program to run under a shell, see the sections "APW Utilities" in Chapter 6 and "Shell Load Files" in Chapter 7.

To launch a program of either file type from the APW shell, enter the pathname of the file as a command. For example, if you want to run a program called `STAR.WARP` that is in the subdirectory `/MYPROGS/GAMES/`, type the following line and press Return:

```
/MYPROGS/GAMES/STAR.WARP
```

Note that ProDOS 8 SYS files can be launched by APW only if the ProDOS 8 operating system (the file `P8`) is present in the system prefix (prefix 4) of your disk and that BIN files cannot be executed by the APW Shell.

Using the Apple IIGS Debugger

Once you have created an executable load file, you can use the Apple IIGS Debugger as an aid in debugging it. The Apple IIGS Debugger is both powerful and versatile. The debugger can execute a load file in memory one instruction at a time, showing you the contents of Apple IIGS registers, stack, direct page, and memory at any step. You can execute each instruction individually, or have the debugger automatically execute each in turn until it reaches a breakpoint that you have set (or until the program hits a `BRK` instruction or crashes). If you have timing-critical code, you can execute specified subroutines or the entire program at the full speed of the Apple IIGS CPU. You can change the contents of registers or memory locations at any time and resume execution of the program. You can display any of the debugger's diagnostic displays or the normal display of your program, and you can switch back and forth between displays at any time.

The debugger shows an assembly-language disassembly of the machine code in memory as it steps through your program. It shows absolute addresses in the disassembly. The debugger cannot translate machine code into any higher-level language or keep track of symbols. You will probably find the debugger of most use, therefore, in debugging assembly-language programs, because it is relatively easy to relate your assembly-language code to the disassembly. For higher-level languages, the debugger might give you some insight into what is going wrong with the execution of the program, but it is up to you to figure out the source-code command or statement responsible.

The APW Debugger is described in detail in the *Apple IIGS Debugger Reference*.

Using the Utilities

The Apple IIGS Programmer's Workshop Shell includes most of the functions that you need to write, compile, link, and run programs. A few functions, however, are

implemented as separate routines designed to be run under the shell; these are referred to as APW *utility programs*, or *utilities*.

Most APW utilities, such as Init (which formats disks), require no more input than any other shell command; in this manual such utilities are referred to as **external commands**. You use them just like other APW commands, but they must be present in the utilities prefix (prefix 6). You may have a few utilities on your system, however, that perform more complex functions and that require some interactive input. If you add such a utility program to your system, refer to the documentation and help file that came with the program for instructions in its use.

Summing It All Up: Developing and Running a Program

This section illustrates the interactions among the various programs in the Apple IIGS Programmer's Workshop by presenting a typical sequence of procedures and events involved in developing and running a multilanguage program. For this purpose, we assume that you are developing an application written mostly in C, with some routines written in 65816 assembly language. See the manuals that came with your APW languages for actual multilanguage programming examples that you can run on your Apple IIGS. The process described here is illustrated in Figure 2.2.

See the *Apple IIGS ProDOS 16 Reference* manual for a complete description of the program load process as implemented by the System Loader and Memory Manager.

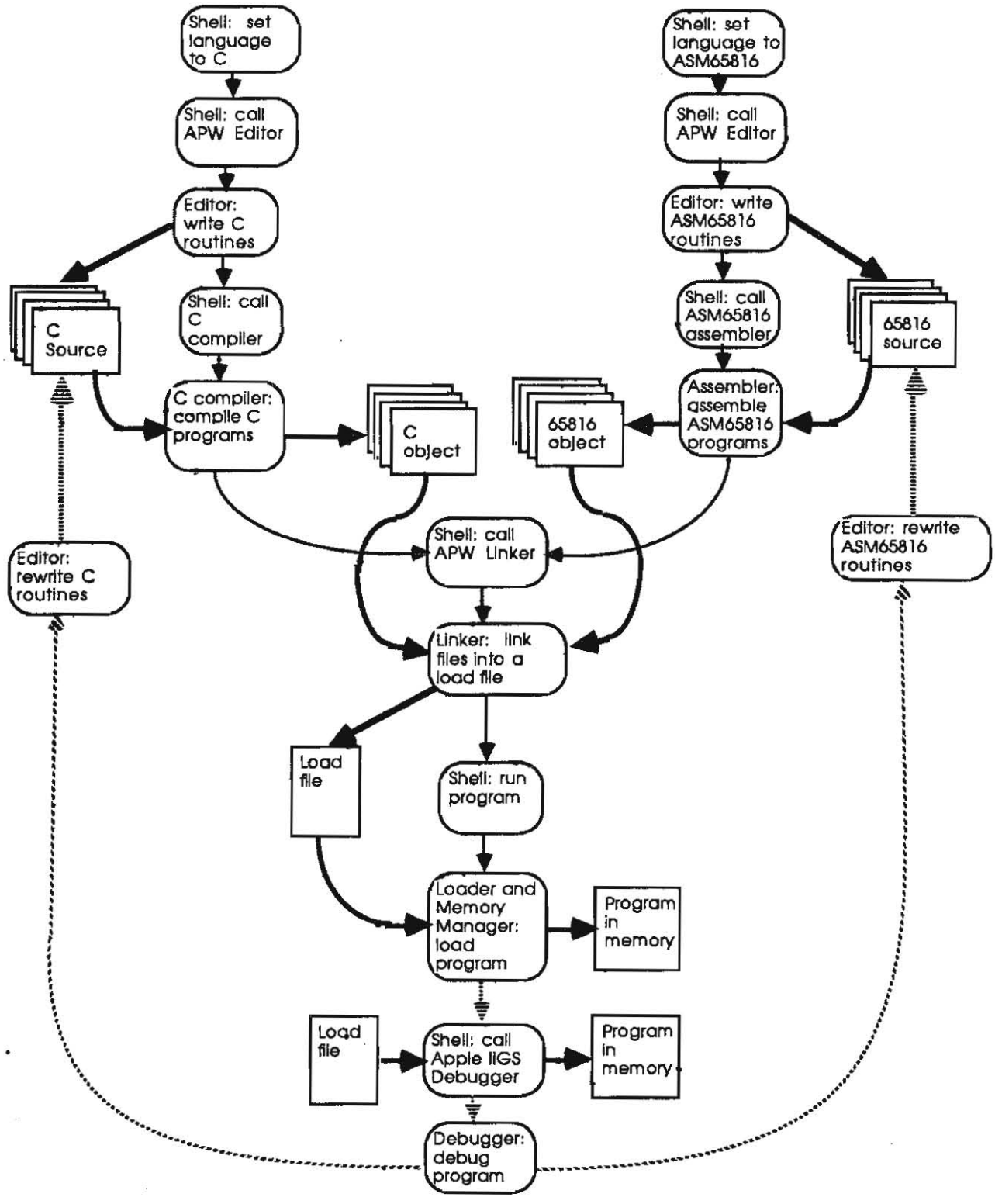


Figure 2.2. Program Interactions

1. Using an APW Shell command, set the **current language** for APW to CC. (Every APW file has an APW language type; if you open a new file, it is given the current APW language type.)
2. Call the APW Editor and open a new file.
3. Use the editor to write the C-language routines. You can divide the program among as many files as you wish. In APW C, you can specify which subroutines go in which load segments. You do not have to return to the shell between files; instead, you can save one file and open another within the editor. Until you open a non-C file with the editor or use a shell command to change the current language, the current language remains CC.
4. Quit the editor, change the current language to `ASM65816`, call the editor, and open a new file. You can divide the 65816 assembly-language routines among as many files and as many segments per file as you wish. The APW Assembler allows you to specify which object segments go in which load segments. Make the assembly-language routines relocatable: that is, use no absolute addresses—use labels and relative addressing only.

   If you have used macros in your assembly-language program, you can run the MacGen utility to generate a custom macro file for the program.

   Until you use a shell command to change the current language or open a non-assembly-language file with the editor, the current language remains ASM65816.
5. Quit the editor, call the APW Assembler to assemble the 65816 assembly-language routines, and call the APW C Compiler to compile the C routines. You can compile both languages with a single command by listing all the files on the same command line or by appending one file to the other with `APPEND` directives and then executing the `COMPILE` command.
6. Use the APW Linker to link the object files into a load file. You can link the object files with the standard linker or with the advanced linker. The standard linker places all object segments with the same load-segment name into a single load segment, while the advanced linker ignores the load-segment names in the object file and follows the directions in a LinkEd file to determine which object segments go in each load segment.

   You can compile and link both languages with a single command by listing all the files on the same command line or by appending one file to the other with `APPEND` directives and then executing the `CMPL` command.

   The shell checks the language type of the first file and calls the C compiler. When the compiler gets to a 65816 file, it returns control to the shell, which calls the APW Assembler. When the assembler is finished, it returns control to the shell again, which calls the standard linker. The object files output from the C compiler and those that are output from the APW Assembler are all in the same format and so are indistinguishable to the linker. The linker combines the object files, resolves references, writes the load file, and returns control to the shell.

   If you want to change load-segment assignments or control the sequence in which load segments are created, you must use the advanced linker. Write a LinkEd file like a language source file: that is, first set the system language to `LINKED` and then use the editor to write the file.

   To compile and link the entire program in one operation using the advanced linker, do the following:

a. Using the editor, tie all of your source files together by placing an APPEND directive (in assembly language) or a #append function (in C) at the end of each file.

b. Put an APPEND directive that references the LinkEd file at the end of the last file in the program.

c. In the shell, execute the COMPILE command.

Alternatively, you can list all the source files, including the LinkEd file, on the COMPILE command line. Make sure that the LinkEd file is the last one listed.

The shell checks the language type of the first file and calls the C compiler. When the compiler gets to a 65816 file, it returns control to the shell, which calls the APW Assembler. When the assembler gets to the LinkEd file, it returns control to the shell again, which calls the advanced linker. The advanced linker, controlled by the commands in the LinkEd file, can do the following:

- combine the object files
- resolve references
- assign object segments to load segments
- label certain load segments as dynamic
- search libraries
- write the load file

When it is finished, the linker returns control to the shell.

7. Run the program by typing in the name of the load file and pressing the Return key. (You can also automatically execute a program after linking by using the CMPLG command.) When a program is run on the Apple IIGS, the following events occur:

   a. The System Loader loads the first segment into memory (calling the Memory Manager to request the block of memory it needs). This segment is static: that is, it remains in memory during the execution of the program. The loader uses the relocation dictionary of the segment to relocate the code to its present location in memory.

   b. The loader loads all other static segments into memory, relocating them as necessary.

   c. The loader passes control of the system to the program and the program begins to execute.

   d. When a reference to a subroutine in a dynamic segment is encountered, control is returned to the System Loader. If the segment is already in memory, the loader transfers control to the segment. If not, the loader locates the segment, loads it into memory, and transfers control to the segment. The System Loader keeps track of all the segments in memory.

      When there is insufficient room in memory to load a segment, the Memory Manager calls the System Loader to unload a dynamic segment from memory.

8. If the program does not run correctly, you can use the Apple IIGS Debugger to step through or trace the code, to insert breakpoints, to disassemble the machine code, and to examine the contents of registers and memory locations. You can modify the code in memory and rerun the program until the bug is fixed.

9. Correct the source code and recompile the program. If the language you are using supports partial compiles, you can do a partial compile to recompile only the routine containing the bug.

10. Relink the program and rerun it. If you have used partial compiles, the linker selects only the most recent version of each segment to put in the load file.
11. You can use the Crunch utility to combine the files created by partial compiles into a single object file. Then link the program once again. Using Crunch is optional; if you have performed several partial assemblies, using this utility speeds up the link process.
12. After you have finished debugging the program, you can use the Compact utility to decrease the size of the load file and to make it load faster.

Advanced Features

This chapter has covered the simpler and more basic procedures you need in order to write, compile or assemble, link, debug, and run a program using the Apple IIGS Programmer's Workshop. APW has many additional capabilities not covered in this chapter. The following list gives some indication of other functions and where to find information about them in this manual. See the Preface for a chapter-by-chapter description of this book. Use the table of contents and the index to find the specific topics in which you are interested.

- You can pipeline commands: that is, you can automatically use the output of one command as the input of another. See "Pipelining Programs" in Chapter 3.
- You can redirect to a disk file the output that would normally go to the screen. You can redirect input that would normally come from the keyboard to be from a disk file. See "Redirecting Input and Output" in Chapter 3.
- You can link two or more object files that have different root filenames into the same load file. See the discussions of the `ASML` and `LINK` commands in Chapter 3.
- You can list the segments, segment-header contents, and segment contents of any file on disk in object module format. See the discussion of the `DUMBOBJ` command in Chapter 3.
- You can control the APW Linker from a file of linker commands, called a *LinkEd file*. LinkEd files provide much more versatile control of the linker than do the shell `LINK`, `CMPL`, and `CMPLG` commands. The LinkEd command language makes it possible for you to
  - choose specific segments to link
  - place specific object segments in a load segment
  - create multiple load segments
  - start segments at specified locations
  - link any number of program files
  - search a library
  - set the program counter
  - open a file for output
  - control the printed output

  For more information about using LinkEd files, see the section "Using the Advanced Linker" in Chapter 5.

- You can specify which subdirectories are searched for specific routines. For example, you can change the subdirectory searched for utility programs from `/APW/UTILITIES/` to `/PRODOS/WORKSHOP/EXT.COM/`. See the section "Standard Prefixes" and the discussion of the `PREFIX` command in Chapter 3.
- You can use shell commands to initialize disks, to move, copy, and rename files, and to create subdirectories. See the section "Command Descriptions" in Chapter 3.
- You can define temporary aliases for commands with the `ALIAS` command. See the discussion of the `ALIAS` command in Chapter 3.
- You can read in a new command table at any time to define new command names or aliases or to add new external commands to the system. See the section "Command Types and the Command Table" and the discussion of the `COMMANDS` command in Chapter 3.
- You can create your own library files. See the discussion of the `MAKELIB` command in Chapter 3.
- You can create a ProDOS 8 executable load file (a BIN file) from a ProDOS 16 load file. See the discussion of the `MAKEBIN` command in Chapter 3.

Part II
Reference

Chapter 3

Shell

The Apple IIGS Programmer's Workshop Shell includes the command interpreter that you use to control the Apple IIGS Programmer's Workshop, and it provides the interface between APW and the Apple IIGS operating system. The shell also provides the following features of APW:

- facilities for copying, renaming, deleting, and moving files
- executable command files (Exec files) for automatic execution of shell commands
- redirection of input and output
- pipelining of programs
- the addition, deletion, and renaming of commands
- the creation of aliases for commands
- a command to assign subdirectories to the ProDOS 16 prefix designators
- commands for assembling, compiling, linking, and running programs
- commands to execute other APW programs such as the editor and the MacGen utility
- the ability to execute other Apple IIGS programs

This chapter provides a reference guide to the APW Shell commands and Exec files. The first part of the chapter explains how to redirect input and output, set standard prefixes, and pipeline commands. It includes an explanation of partial compiles and provides an introduction to shell command types and the **command table**. The central and largest part of this chapter provides complete descriptions of all of the shell commands. You should turn to this section any time you need a full explanation of a command or command parameters. The last part of the chapter explains how to write and use Exec files and describes in detail the commands that are used primarily in Exec files.

The following shell-related topics are covered in Chapter 2, "How to Use the Shell and Editor":

- entering commands
- scrolling through previously entered commands
- entering multiple commands on a single command line
- responding to parameter prompts
- using prefix numbers for pathname prefixes
- using device names for volume names
- using wildcard characters in filenames

See Part III, "Inside the Apple IIGS Programmer's Workshop," for the information you need in order to add a program to APW.

Standard Prefixes

When you specify a file on the Apple IIGS, as when indicating which file to edit or utility to execute, you must specify the file's pathname as discussed in the section "Pathnames" in Chapter 2. ProDOS 16 provides eight prefix numbers that can be used in the place of prefixes in pathnames. This section describes the APW default prefix assignments for these ProDOS 16 prefixes.

APW uses six of the ProDOS 16 prefixes (0 and 2 through 6) to determine where to search for certain files. When you start APW, these prefixes are set to the default values shown in Table 3.1 . You can change any of the eight ProDOS 16 prefixes with the APW `PREFIX` command, as described in this chapter, and you can include `PREFIX` commands in the `LOGIN` file, as shown in the section "Using Prefix Numbers" in Chapter 2.

You can use the ProDOS 16 prefix numbers instead of prefixes in pathnames. For example, if you set prefix 7 to `/APW/MYPROGS/`, you can specify the pathname of `/APW/MYPROGS/C.SOURCE/GOODSTUFF` as `7/C.SOURCE/GOODSTUFF`.

Table 3.1. Standard Prefixes

Prefix Number	Use	Default
*	Boot prefix	Boot prefix
0	Current prefix	Undefined
1	Application	Prefix of `APW.SYS16`
2	APW library	`1/LIBRARIES/`
3	APW work	`1/`
4	APW system	`1/SYSTEM/`
5	APW language	`1/LANGUAGES/`
6	APW utility	`1/UTILITIES/`
7	undefined	

Each time you restart your Apple IIGS, ProDOS 16 retains the volume name of the boot disk; this volume name is the **boot prefix**. You can use an asterisk (*) in a pathname in some commands to specify the boot prefix. You cannot change the volume name assigned to the boot prefix except by rebooting the system.

> **Note:** When you use an asterisk followed by a space on a command line, it is interpreted as a comment; see the description of the `COMMENT` command in the section "Exec Files" in this chapter for details.

The **application prefix** is the prefix of the last application run, whether it's the APW Shell, the APW Editor, a utility program, or any other program.

The **current prefix** (also called the **default prefix**) is the one that is assumed when you use a partial pathname. After APW is started, the current prefix (prefix 0) is undefined unless you use the `PREFIX` command to set it.

When you start APW, ProDOS 16 and the APW Shell become resident in memory, so changing the current prefix does not affect the ability of the shell to function. The following paragraphs describe APW's use of the standard prefixes.

APW looks in the current prefix (prefix 0) when you use a partial pathname for a file.

When you first boot APW, the application prefix (prefix 1) is set to the prefix of the `APW.SYS16` file and is used to set the other prefixes used by APW. As soon as you run another program, such as the editor or a compiler, prefix 1 is reset. The other APW prefixes do not change when prefix 1 changes, however.

The standard linker searches the files in the APW library prefix (prefix 2) to resolve any references not found in the object files being linked. For information on creating and using APW Assembler library files, see the discussion of the `MAKELIB` command in this chapter, and the *Apple IIGS Programmer's Workshop Assembler Reference* manual.

The work prefix (prefix 3) is used by some APW programs for temporary files. For example, when you pipeline two or more programs so that the output of one program becomes the input to the next, APW creates temporary files in the work prefix for the intermediate results (pipelines are described in the section "Pipelining Programs" later in this chapter). Commands that use the work prefix operate faster if you set the work prefix to a RAM disk, since I/O is faster to and from memory than to and from a disk. If you have enough memory in your system to do so (1280K should be sufficient), use the Apple IIGS control panel to set up a 256K RAM disk, and then use the `PREFIX` command to change the work prefix. To change prefix 3 to a RAM disk named `/RAM5`, for example, use the following command:

```
PREFIX 3 /RAM5
```

APW looks in the APW system prefix (prefix 4) for the following files:

- `EDITOR` The APW Editor (see Chapter 4).
- `LOGIN` An optional Exec file that is executed automatically at load time if it is present (see the section "Exec Files" later in this chapter).
- `SYSCMND` The shell's command table (see the section "Command Types and the Command Table" later in this chapter).
- `SYSEMAC` Editor macros (see the section "Macros" in Chapter 4).
- `SYSHELP` The help file for the editor (see the discussion of the `HELP` command in Chapter 4).
- `SYSTABS` The default tab and editing-mode settings for the editor (see the section "Setting Editor Defaults" in Chapter 4).

APW looks in the language prefix (prefix 5) for the APW Linker, the APW Assembler, and any other assemblers, compilers, and text formatters that you have installed in your copy of APW.

APW looks in the utility prefix (prefix 6) for all of the APW utility programs except for the editor, assembler, and compilers. Prefix 6 includes the programs that execute external commands, such as `CRUNCH`, `INIT`, and `MAKELIB`. The utility prefix also contains the `HELP/` subdirectory, which contains the text files used by the `HELP` command. Command

types are described in the the section "Command Types and the Command Table" later in this chapter.

> **Note:** The `UTILTIES/` subdirectory on the `/APW` disk that comes with APW Version 1.0 contains no help files and only a subset of the utility programs. The `/APWU` disk contains a complete `UTILTIES/` subdirectory. If you are running APW from floppy disks, you can use the `MU` command to change the utility prefix to `/APWU/UTILTIES/` and the `UMU` command to change the utility prefix to `/APW/APW/UTILTIES/`

Redirecting Input and Output

Most Apple IIGS programs use tool calls to accomplish input and output functions. The Text Tool Set accepts input from whatever device driver routine is pointed to by **standard input** and sends output to the device driver routine pointed to by **standard output**. (The Text Tool Set is described in Volume II of the *Apple IIGS Toolbox Reference*.) Input received through standard input is usually from the keyboard, while output through standard output is usually sent to the screen. APW allows you to redirect the input and output of any program—including the APW Shell and utilities—that uses standard I/O. Input to a command or program can come from a text file or from the output of a program instead of from the keyboard. Output from a command or program can be sent to a printer or a disk file instead of to the screen.

Error messages can be redirected independently of other output. Error output is used instead of standard output so that error messages can be displayed on the screen (for example) even when standard output is going to a file. By redirecting error output, you can place error messages in a separate disk file from that used for program listings and other output.

Note that the input and output of programs that do not use standard I/O cannot be redirected by APW.

To redirect standard input or output, use the following conventions on the command line:

<inputdevice	Redirect standard input to be from *inputdevice*.
>outputdevice	Redirect standard output to go to *outputdevice*.
>>outputdevice	Redirect standard output to be appended to the current contents of *outputdevice*.
>&outputdevice	Redirect error output to go to *outputdevice*.
>>&outputdevice	Redirect error output to be appended to the current contents of *outputdevice*.

You can include spaces before or after the redirection operators (<, >, >>, >&, >>&) to improve readability, but no spaces are necessary.

The input device can be the keyboard or any text or source file. The keyboard is the default input device. To reassign the keyboard as the input device after input has been redirected, use the device name .CONSOLE.

The output device can be the screen, the printer, or any file. If the file named does not exist, APW opens a file with that name. To redirect output to the printer, use .PRINTER. The screen is the default output device. To reassign the screen as the output device after output has been redirected, use the device name .CONSOLE.

> **Warning:** Be sure the printer is on-line before directing output to it. With some hardware (such as parallel printer cards), the system hangs if the printer is not on-line.

If you use output redirection to open a new file on disk, the file has the file type TXT (ProDOS 16 file type $04). If you use output redirection to send output to an existing file, the file's language type is not changed.

> **Warning:** If you redirect output to an existing file, the original contents of that file are destroyed without warning, regardless of the file type of the file. If you do not want to overwrite an existing file, check to make sure there is no file with the pathname you intend to specify *before* executing the redirection.

> **Important:** If a disk file is used for input or output, the disk must remain on-line until the command finishes executing.

Both input and output redirection can be used on the same command line. The input and output redirection instructions can appear in any position on the command line.

For example, to redirect output from an assembly of the program MYPROG to the printer, you could use either of the following commands:

```
ASSEMBLE MYPROG >.PRINTER

ASSEMBLE >.PRINTER MYPROG
```

To redirect output from the CATALOG command to be appended to the data already in a disk file named CATSN.DOGS, use the following command:

```
CATALOG >>CATSN.DOGS
```

To redirect input in response to the AINPUT directives in an assembly-language source file to be from the file ANSWERS rather than the keyboard, you could use one of the following commands:

```
ASSEMBLE <ANSWERS MYPROG

ASSEMBLE MYPROG <ANSWERS
```

Input and output redirection can be used in Exec files. See the section "Exec Files" later in this chapter for a description of Exec files.

Pipelining Programs

APW lets you automatically execute two or more programs in sequence, directing the output of one program to the input of the next. As illustrated in Figure 3.1, the output of each program but the last is written to a temporary file in the work subdirectory named `SYSPIPE`*n*, where *n* is a number assigned by APW. The first temporary file opened is assigned an *n* of 0; if a second `SYSPIPE`*n* file is opened for a given pipeline, it is named `SYSPIPE1`, and so forth.

> **Important:** Pipelining works only with programs that take their input from standard input and send their output to standard output.

To **pipeline**, or sequentially execute programs `PROG0`, `PROG1`, and `PROG2`, use the following command:

```
PROG0 | PROG1 | PROG2
```

The output of `PROG0` is written to `SYSPIPE0`. The input for `PROG1` is taken from `SYSPIPE0` and the output is written to `SYSPIPE1`. The input for `PROG2` is taken from `SYSPIPE1`, and the output is written to standard output.

`SYSPIPE`*n* files are text files (ProDOS 16 file type $04) and can be opened by the editor.

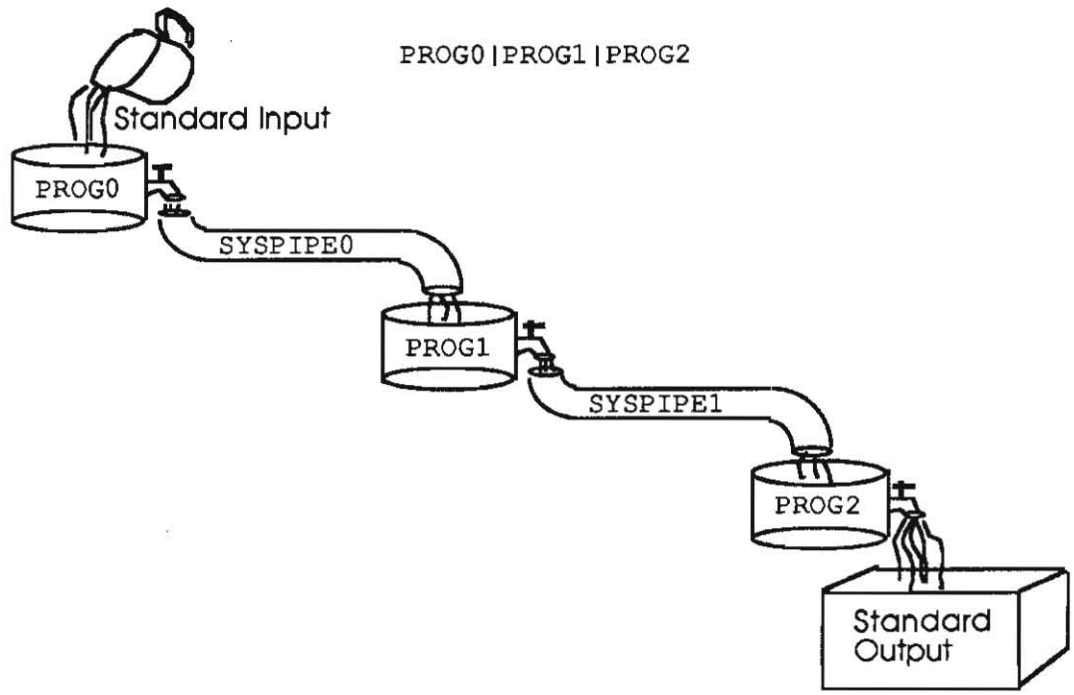


Figure 3.1. Pipelining Programs

For example, suppose you have a utility program called `ALPHA` that takes a list of items from standard input, alphabetizes them, and writes them to standard output. You could use the following command line to alphabetize the contents of the current directory and write them to the screen:

```
FILES|ALPHA
```

To send the output to the file `LISTING` rather than to the screen, use the following command line:

```
FILES|ALPHA >LISTING
```

The `SYSPIPE`*n* files are not deleted by APW after the pipeline operation is complete. For this reason, you can use the editor to examine the intermediate steps of a pipeline as an aid to finding errors. The next time a pipeline is executed, however, any existing `SYSPIPE`*n* files are overwritten.

Partial Assemblies or Compiles

If you are writing a large program, you may find that the debugging process is being slowed considerably by the amount of time it takes to compile the program. You can often speed up this process by using partial compiles or assemblies.

> **Note:** Currently, the APW Assembler is the only program that provides this capability, but other APW compilers may support partial compiles in the future. Check the manual that came with your compiler to see if it allows you to perform partial compiles.

In a partial compile or assembly, you specify which object segments are to be compiled. The new versions of the segments are placed in a file with the same **root filename** as the rest of the program, but with the next higher alphabetic extension. The root filename of a file is the filename minus any filename extensions added by the assembler or compiler. For example, the files `MYFILE.ROOT`, `MYFILE.A`, and `MYFILE.B` all have the same root filename: `MYFILE`. (The root filename can include a period (.). For example, `MYFILE.O` is a valid root filename from which the APW Assembler would create files `MYFILE.O.ROOT` and `MYFILE.O.A`.)

> **Important:** The root filename cannot be longer than 10 characters for files to which the `.ROOT` extension will be appended because ProDOS 16 limits the entire filename to 15 characters. Using more than 10 characters in such a filename will result in a fatal assembler or compiler error (`Unable to open output file`).

To do a partial compile or assembly, you must use one of the following shell commands:

- `ASSEMBLE`
- `ASML`
- `ASMLG`
- `COMPILE`
- `CMPL`
- `CMPLG`
- `RUN`

These commands are all very similar. The ASML and CMPL commands automatically link the program after compiling or assembling it; ASMLG, CMPLG, and RUN also link the program, and then automatically run the program after linking it. The COMPILE command is actually an alias for the ASSEMBLE command, as is CMPL for ASML and CMPLG (and RUN) for ASMLG. All of these commands are described in this chapter.

Each of these commands has an optional parameter called NAMES, which you follow with a list of the names of segments you want to compile or assemble. When the shell finds a NAMES parameter, it performs a partial compile or assembly. Keep the following points in mind when using the NAMES parameter:

- The name to list is the *object*-segment name, not the *load*-segment name. In an assembly-language source file, the label of a START, PRIVATE, DATA, or PRIVDATA directive is the object-segment name; the operand of the directive is the load-segment name. Any number of object segments can have the same load-segment name. Load-segment names are used by the linker and have no effect on an assembly.
- Object-segment names are case-sensitive. If the language you are using is not case-sensitive, it converts all names to uppercase in the object file, regardless of how they appear in the source file. When using the NAMES parameter for case-insensitive languages, you must enter the segment names in all uppercase. For case-sensitive languages, on the other hand, you must list all object-segment names exactly as they appear in the source code. Assembly language is case-insensitive in APW unless you have used the CASE ON directive in the source file.
- You can include in one NAMES parameter list the names of all the segments you want to use. This is true whether you have only one source file or you are compiling several files at once.
- Be sure to include a KEEP directive in the file or a KEEP parameter in the command line, or to define a shell KeepName variable. If you don't, the program is compiled, but the output is not saved.
- Use the same root filename for the partial compile as you used for the original compile. The linker can automatically scan all files with the same root filename for the most recent version of each segment.

An example of a sequence of partial assemblies is given at the end of this section.

There are two circumstances under which a file with a higher alphabetic suffix (.B, .C, and so on) is created.

- If you include a NAMES parameter on the command line to request a partial assembly or compile, only the segments named are compiled, and they are placed in a file with the next available alphabetic extension. For example, if the files MYPROG.ROOT and MYPROG.A are already on the disk, a partial assembly creates the file MYPROG.B.
- If the compile involves more than one language, then the first compiler or assembler creates the .ROOT file and may create the .A file, the second compiler might create the .B file, and so on.

> **Note:** You cannot have more than 26 alphabetic-extension files (`.A` through `.Z`) for each root filename. You can use the CRUNCH command described in this chapter to combine all the alphabetic-extension files for one root filename into one `.A` file at any time.

When the linker links the program, it uses the following procedure:

1. It starts with the `.ROOT` file and links the segment contained in that file.
2. It looks for a `.A` file. If it finds one, the linker looks for a `.B` file, and so on.
3. It links the segments in the file that has the highest alphabetic suffix it has found.
4. It works its way back through the alphabet to the `.A` file, ignoring any segments with object-segment names identical to those it has already found, and linking the rest.

You can also control which segments are linked and in what sequence by using a LinkEd command file; see the section "Linking With a LinkEd Command File" in Chapter 5 for details.

> **Important:** During a partial compile, the compiler first looks for a `.ROOT` file, then a `.A` file, then a `.B` file, and so on. The search is terminated as soon as one file in the sequence is not found, and the next file created is given the next higher alphabetic suffix. Therefore, if the files `MYFILE.A`, `MYFILE.B`, and `MYFILE.D` are in the subdirectory, but `MYFILE.C` is not, the assembler or compiler never finds `MYFILE.D`. The next file created by a partial assembly or compile, then, would be `MYFILE.C`. You must be careful not to let such a situation occur, because (in this example) the linker would start the next link with the file `MYFILE.D`.

An example of a partial compile and assembly is shown in Figure 3.2. Assume you have written a program consisting of two source files. The first file, named MYPROG, is in 65816 assembly language. It includes the main part of the program, and has four object segments named MAIN, SEG1, SEG2, and DATA. The second file, named MYPROGC, is in C. It includes a couple of mathematical subroutines that you didn't want to write in assembly language. The subroutines are named `Lagrange` and `Fourier`. At the end of the assembly-language routine is an APPEND directive that appends MYPROGC. MYPROG begins with a KEEP directive that names the output file as TRANSFORM. To assemble MYPROG and compile MYPROGC, enter the following command:

```
ASSEMBLE MYPROG
```

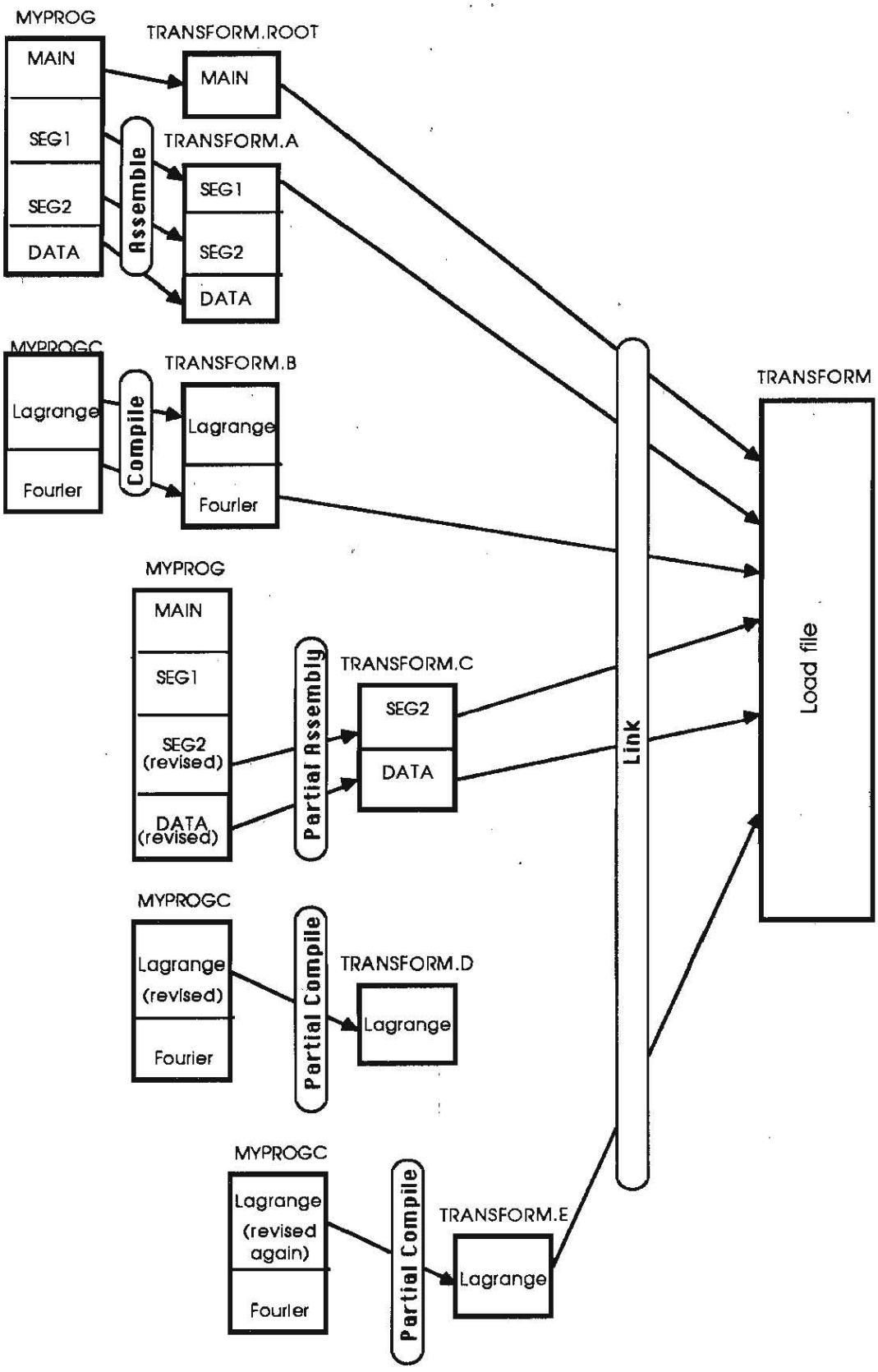


Figure 3.2. An Example of the Use of Partial Compiles

The APW Shell processes the program as follows:

1. The shell checks the language type of MYPROG and calls the APW Assembler.
2. The assembler starts to assemble MYPROG; it opens TRANSFORM.ROOT and puts the first segment (MAIN) in that file.
3. The assembler closes TRANSFORM.ROOT, opens TRANSFORM.A, and puts the rest of the segments in there.
4. When it gets to the APPEND directive, it opens MYPROGC and checks its APW language type. Finding that it's not an assembly-language file, the assembler closes MYPROGC and TRANSFORM.A and returns control to the shell.
5. The shell calls the C compiler, which compiles MYPROGC, placing both subroutines in TRANSFORM.B.

The following files are now present on disk:

- MYPROG assembly-language source file
- MYPROGC C source file
- TRANSFORM.ROOT object file containing the segment MAIN
- TRANSFORM.A object file containing SEG1, SEG2, and DATA
- TRANSFORM.B object file containing segments Lagrange and Fourier

After working on the program for a while, you have changed segments SEG2, DATA, and subroutine Lagrange. Rather than reprocess the entire program, you perform a partial assembly by using the following command:

```
ASSEMBLE MYPROG NAMES=(SEG2 DATA Lagrange)
```

Note that the segment names for the C routine must be entered exactly as they appear in the source code, since C is a case-sensitive language. The assembler finds the segments SEG2 and DATA in MYPROG, assembles them, and places them in the file TRANSFORM.C. Then the shell calls the C compiler, which extracts subroutine Lagrange and places it in the file TRANSFORM.D.

Finally, you make one more change to Lagrange. To recompile that routine only, you need not process MYPROG at all. Instead, use the following command:

```
COMPILE MYPROGC KEEP=TRANSFORM NAMES=(Lagrange)
```

This time you used the COMPILE command rather than the ASSEMBLE command because it satisfied your sense of aesthetics to use a compile command with a compiler. Actually, the COMPILE and ASSEMBLE commands are aliases—they call the same APW Shell routine. Note that you have to use the KEEP parameter in the command line, since the file MYPROGC contains no KEEP command.

This last partial compile creates the file TRANSFORM.E.

Finally, to link the program and create the load file TRANSFORM, you use the following command:

```
LINK TRANSFORM KEEP=TRANSFORM
```

The linker does the following:

1. It finds the file TRANSFORM.ROOT and links the segment MAIN.
2. It finds the file TRANSFORM.A, then searches for TRANSFORM.B, and so on until it finds TRANSFORM.E. It links Lagrange from TRANSFORM.E.
3. It finds Lagrange in TRANSFORM.D, realizes it has already linked it, and ignores it.
4. It links SEG2 and DATA in TRANSFORM.C.
5. It links Fourier in TRANSFORM.B, ignoring the older version of Lagrange it finds there.
6. It links SEG1 in TRANSFORM.A, ignoring SEG2 and DATA.

Command Types and the Command Table

The Apple IIGS Programmer's Workshop includes a large number of commands that perform a variety of functions, from listing a disk directory to compiling a program. There are three types of commands in APW: **internal, external**, and **language**.

- An *internal* command is one included in the APW Shell. Internal commands are resident in memory whenever you are in the Apple IIGS Programmer's Workshop.
- An *external* command is a separate APW utility program. These programs are in the utility prefix (prefix 6) and are loaded from disk when you execute the commands.
- A *language* command changes the default APW language type. Any new file opened for editing with the EDIT command is given the default language type.

  Note: The existence of a language command on your system does not necessarily indicate that you have the compiler for that language on your disk. Check the contents of the language prefix (prefix 5) to see which compilers are installed in your copy of APW.

The APW language type of a file is stored in the ProDOS 16 directory entry for the file, but is separate from the ProDOS 16 file type. The APW language types include all assemblers and compilers recognized by APW, plus ASCII text files (which have the language type PRODOS), LinkEd command files (LINKED), and shell command files (EXEC).

Your APW disk or directory includes a language subdirectory (prefix 5, normally LANGUAGES/). Note that, while compilers, assemblers, and the APW Linker are included in the language prefix, some of the language types included in APW do not have corresponding files in the language prefix. The EXEC, PRODOS, and TEXT language types, for example, do not have compilers and so do not appear in the language prefix.

If you open an existing file for editing, the default language type changes automatically to match that file. The language type of any source or text file can be changed with the CHANGE command. Use the SHOW LANGUAGES command (note the plural) for a list of the language commands available, and the SHOW LANGUAGE (singular) command for the current default.

All APW compiler and assembler source files, LINKED files, and EXEC files are ProDOS 16 file type $B0. PRODOS text files are ProDOS 16 file type $04.

Table 3.2 shows some of the language types that APW recognizes. For a more complete list of language numbers that have been assigned for APW, see Appendix B. The assignment of a language number does not necessarily imply that that language is currently available. To see a complete list of all APW language numbers, obtain the latest version of Apple IIGS Technical Note #20.

Table 3.2. APW Language Types

Language	Number	Use
ASM65816	3	65816 assembler source
CC	10	APW C source
EXEC	6	command file
LINKED	9	APW Linker command language
PRODOS	0	ProDOS 16 text file (ProDOS 16 file type $04)
TEXT	1	APW text file

All APW source files have a ProDOS 16 file type of $B0; the APW language type allows APW to distinguish between source files in different languages. APW TEXT files are standard ASCII files with ProDOS 16 file type $B0 and an APW language type of TEXT. The TEXT language type is provided to support any text formatters that may be added to APW in the future. The PRODOS language type creates standard ASCII files with ProDOS 16 file type $04; these files are shown in a directory listing as TXT files. See the *Apple IIGS ProDOS 16 Reference* for a discussion of ProDOS 16 file types.

You can add, delete, and rename commands by editing the default command table, which is in a file called SYSCMND in the system prefix (prefix 4). This command table is read by the shell at load time. You can also cause the shell to read a custom command table at any time by using the COMMANDS command.

You can alter the contents of the command table to add command names to the shell, to create permanent aliases for commands, or to delete commands. To change the contents of a command table, open the command-table file with the EDIT command. Each command in the command table is on a separate line; each line contains three fields, separated by spaces, as illustrated in Figure 3.3. The fields specify the commands as follows:

1. The first field is the command name, which must follow the rules for a legal ProDOS 16 filename. Command names are not case-sensitive.
2. The second field indicates the command type. Enter a C for an internal command, a U (for *utility*) for an external command, or an L for a language. If you precede the command type with an asterisk (*), the shell assumes that the program can be restarted and does not remove it from memory as long as that memory is not needed for other purposes. In this case, if that command is executed again, the program need not be reloaded from disk. (This feature is useful only for utilities (U) and compilers (L); if you precede a C with an asterisk, the shell ignores the asterisk.)

Important: If you put an asterisk in the command table in front of the command type of a utility or language that cannot be restarted, an error may occur the first time the shell tries to restart that program. See the discussion of restartability following Figure 3.3.

3. The third field specifies the command number or language number. For internal commands, you must use an existing command number. For languages, you must use one of the recognized language numbers listed in Appendix B. For external commands, the third field is blank.

Note: If you add an internal shell command to the command table with a new command number, the command will not work and an error message will be generated when you try to execute the command. To add a new command to APW, create a utility program as described in Chapter 6, "Adding a Program to APW."

The third field can be followed by a space or tab and a comment. Blank lines are ignored. You can create a comment line by starting it with a semicolon (`;`).

```
; Sample command table
     ALINK           C     3
     ASM65816        *L    3
     ASML            C     1
     ASMLG           C     2
     ASSEMBLE        C     3
     CC              *L    10
     COMMANDS        C     35
     COMPILE         C     3
     CONTINUE        C     26
     COPY            C     5
     DUMPOBJ         *U
     ECHO            C     29
```

Figure 3.3. Sample of a Command Table

In Figure 3.3, you can see that `ALINK`, `ASSEMBLE`, and `COMPILE` all have the same command number: `3`. These internal commands (type `C`) are all aliases. `ASM65816`, on the other hand, is a language command (type `L`). Language number 3 is not related to command number 3. `DUMPOBJ` is a restartable utility program.

Restartability: For a program to be restartable, it must reinitialize all variables and arrays each time it starts. To make it easier for you to write restartable programs, OMF version 2.0 provides *initialization segments*, which are reloaded from disk and executed each time a program is restarted from memory, and *reload segments*, which are reloaded from disk each time a program is restarted.

Because OMF Version 1.0 does not support reload segments, and because the current version of the linker generates OMF Version 1.0 files, the linker cannot create reload segments. The Compact utility, however, converts OMF Version 1.0 load files to OMF Version 2.0. If it finds a load segment named `~globals` or `~arrays`, Compact makes it a reload segment. In addition, if you set the `KIND` field in the segment header to $1F, Compact makes that segment a reload segment.

See the description of the LinkEd SEGMENT command in Chapter 5 for a way to set the KIND field of a load segment. The Compact utility is described later in this chapter.

The commands delivered with APW are shown in Table 3.3.

Table 3.3. APW Commands

Command	Use	Type
*	Comment character	Internal
ALIAS	Assign a temporary alias to a command	Internal
ALINK	Compile a linker command file	Internal
ASM65816	Change default language to 65816 assembly language	Language
ASML	Assemble and link the program	Internal
ASMLG	Assemble, link, and go (run the program)	Internal
ASSEMBLE	Assemble the program	Internal
BREAK	Exec-file command	Internal
CANON	Replaces words with the canonical spelling specified in a dictionary file	External
CAT	List the disk directory	Internal
CATALOG	List the disk directory	Internal
CC	Change default language to APW C	Language
CHANGE	Change the language type of an existing source file	Internal
CMPL	Compile and link the program	Internal
CMPLG	Compile, link, and go (run the program)	Internal
COMMANDS	Read the command table	Internal
COMMENT	Comment	Internal
COMPACT	Converts load file to compact form	External
COMPILE	Compile the program	Internal
CONTINUE	Exec-file command	Internal
COPY	Copy a file, directory, or volume	Internal
CREATE	Create a new subdirectory	Internal
CRUNCH	Combine object files formed by partial compiles or assemblies into a single file	External
DEBUG	Execute the Apple IIGS Debugger program(if installed)	External
DELETE	Delete a file	Internal
DISABLE	Disable file attributes	Internal
DUMPOBJ	List the contents of an OMF file to standard output	External
ECHO	Exec-file command	Internal
EDIT	Edit an existing file, or open a new file	Internal
ELSE	Exec-file command	Internal
ENABLE	Enable file attributes	Internal
END	Exec-file command	Internal
EQUAL	Compares two files or directores for equality	External
EXEC	Change default language to EXEC command language	Language
EXECUTE	Execute an Exec file at present command level	Internal
EXIT	Exec-file command	Internal
EXPORT	Export a shell variable	Internal
FILES	List the contents of a directory, including subdirectories	External
FILETYPE	Change file type to type specified	Internal
FOR	Exec-file command	Internal
HELP	Provide on-screen help for commands, or list all available commands	Internal

Table 3.3. APW Commands (continued)

HISTORY	List last 20 commands entered	Internal
IF	Exec-file command	Internal
INIT	Initialize a disk	External
LINK	Link an object file	Internal
LINKED	Change default language to the LinkEd command language	Language
LOOP	Exec-file command	Internal
MACGEN	Generate a macro library for a specific program	External
MAKEBIN	Create a ProDOS 8 binary file from a ProDOS 16 load file	External
MAKELIB	Generate a library file from an object file	External
MOVE	Move a file to another directory or volume	Internal
PREFIX	Change the default prefixes	Internal
PRODOS	Change default language to ProDOS 16 text	Language
QUIT	Quit APW	Internal
RENAME	Change a filename	Internal
RUN	Same as ASMLG or CMPLG	Internal
SEARCH	Search a file for a string	External
SET	Set shell variables	Internal
SHOW	Show languages, system default language, prefixes, time, volumes on line	Internal
TEXT	Change default language to TEXT	Language
TYPE	Type a file to standard output	Internal
UNALIAS	Delete an alias created with the ALIAS command	Internal
UNSET	Delete a shell variable	Internal
VERSION	Displays version of APW you are using	External

See Chapter 6 for instructions on adding APW utilities to the Programmer's Workshop.

Command Descriptions

The following notation is used to describe commands:

UPPERCASE Uppercase letters indicate a command name or an option that must be spelled exactly as shown. The command interpreter is not case sensitive; that is, you can enter commands in any combination of uppercase and lowercase letters. Segment names *are* case-sensitive. In case-sensitive languages, segment names must be entered exactly as they appear in the source code. Segment names in case-insensitive languages must be entered in uppercase.

italics Italics indicate a variable that you must replace with specific information, such as a pathname or file type.

directory This parameter indicates any valid directory pathname or partial pathname. It does *not* include a filename. If the volume name is included, *directory* must start with a slash (/); if *directory* does not start with a slash, the current prefix is assumed. For example, if you are copying a file to the subdirectory `SUBDIRECTORY` on the volume `VOLUME`, the *directory* parameter would be `/VOLUME/SUBDIRECTORY/`. If the current prefix were `/VOLUME/`, you could use `SUBDIRECTORY` for *pathname* .

The device names `.D1`, `.D2`, . . . `.D`*n* can be used for volume names. If you use a device name, do not precede it with a slash. For example, if the volume `VOLUME` in the above example were in disk drive `.D1`, you could enter the *directory* parameter as `.D1/SUBDIRECTORY/`.

ProDOS 16 prefix numbers can be used for directory prefixes. Two periods (..) can be used to indicate one subdirectory above the current subdirectory. If you use one of these substitutes for a prefix, do not precede it with a slash. For example, the `HELP/` subdirectory on the APW disk can be entered as `6/HELP/`.

filename This parameter indicates a filename, *not* including the prefix. The unit names `.CONSOLE` and `.PRINTER` can be used as filenames.

pathname This parameter indicates a full pathname, including the prefix and filename, or a partial pathname, in which the current prefix is assumed. For example, if a file is named `FILE` in the subdirectory `DIRECTORY` on the volume `VOLUME`, the *pathname* parameter would be `/VOLUME/DIRECTORY/FILE`. If the current prefix were `/VOLUME/`, you could use `DIRECTORY/FILE` for *pathname*. A full pathname (including the volume name) must begin with a slash (/); do *not*, however, precede *pathname* with a slash if you are using a partial pathname.

The device names `.CONSOLE` and `.PRINTER` can be used as filenames; the device names `.D1`, `.D2`, . . . `.D`*n* can be used for volume names; and ProDOS 16 prefix numbers or double periods (..) can be used instead of a prefix.

| A vertical bar indicates a choice. For example, `+L|-L` indicates that the command can be entered as either `+L` or as `-L`.

A | **B**	An underlined choice is the default value.
[ ]	Parameters enclosed in square brackets are optional.
. . .	Ellipses indicate that a parameter or sequence of parameters can be repeated as many times as you wish.

The following pointers will help you use the APW Shell command interpreter:

- The command-line prompt is a number sign (#); whenever a number sign appears at the left edge of the screen followed by a cursor, you can enter a command.
- You must separate the command from its parameters by one or more spaces.
- You can use the Right Arrow key to expand command names as described in the "Entering Commands" section of Chapter 2; you can use the Up and Down Arrow keys to scroll through previously entered commands.
- There are no abbreviations for command names, except for those aliases that you add to the system as described in the section "Command Types and the Command Table" in this chapter and in the description of the `ALIAS` command in this chapter.
- All commands and parameters, except for segment names, can be entered in any combination of uppercase and lowercase characters. Segment names *are* case-sensitive. In case-sensitive languages, segment names must be entered exactly as they appear in the source code. Segment names in case-insensitive languages must be entered as all uppercase characters.
- When you are calling an assembler, compiler, or linker, if there is a conflict between a parameter in a command line and a source-code command, the command-line parameter takes precedence. When neither a source-code command nor a command-line parameter has been used, the default parameter is used.

  For example, if you specify `+L` as a parameter for the `ASSEMBLE` command, a source-code listing is included in the output even if you include a `LIST OFF` directive in the source code. If you include neither the `LIST` directive nor the `L` parameter, the default (no listing) is used.
- If you fail to enter a required parameter, you are prompted for it, as described in Chapter 2.
- Any of the APW Shell commands can be placed in an Exec command file for automatic execution. Exec files are described in the section "Exec Files" later in this chapter.

ALIAS

`ALIAS [`*alias* `[`*command*`]]`

This internal command assigns an alias to an APW Shell command.

alias	The alias that you want to assign to this command. If you do not include an alias, all aliases for all commands that are in effect are written to the screen.
command	The shell command for which you want to define an alias. You can include one or more parameters as part of the command. If you do not include a command name, the command for which *alias* is an alias is written to the screen.

You can use this command to assign temporary aliases for APW commands. An alias assigned by this command is local to the Exec file in which it is defined and is passed on to any Exec file called by that file. An alias defined in the LOGIN file is valid on the command line and in all Exec files. An alias defined in an Exec file is valid in all Exec files called by that file, but not at higher levels unless the Exec file is called with an EXECUTE command.

For example, suppose you want to create the alias DIR for the CATALOG command. To do so, execute the following command:

```
ALIAS DIR CATALOG
```

Now, each time you type DIR, you get a catalog listing.

As another example, suppose you want to create an alias called VOLUMES that lists the units that are on line plus the current date and time. To do so, execute the following command:

```
ALIAS VOLUMES SHOW UNITS TIME
```

Now, each time you type VOLUMES, the system responds as if you had typed SHOW UNITS TIME.

The ALIAS command makes a single substitution for whatever alias name you specify. It also allows you to use an existing command for an alias. Thus, the following command is valid:

```
ALIAS SHOW SHOW UNITS
```

This command makes SHOW an alias for the command SHOW UNITS. If you execute this ALIAS command, then each time you enter SHOW, the system responds as if you typed SHOW UNITS. For example, if you type SHOW TIME, the system responds as if you typed SHOW UNITS TIME.

> **Note:** The ALIAS command does not check to see if the command for which you are creating an alias is valid. Therefore, you can create a nonfunctional alias or even accidently make an existing command nonfunctional. For example, if you enter the command ALIAS SHOW TIME, the alias SHOW is substituted for the command TIME, making the original command SHOW inoperative. Because there is no TIME command, when you enter SHOW, you get the error message ProDOS: File not found. To correct this condition, use the UNALIAS command (in this example, UNALIAS SHOW).

The ALIAS command does not modify the command table; aliases are lost each time you boot the system (unless they are included in the LOGIN file). See the section "Command Types and the Command Table" earlier in this chapter for instructions on creating permanent aliases for commands.

Use the UNALIAS command to delete aliases set with the ALIAS command.

ALINK

ALINK [*option ...*] *file1* [*file2 ...*] [KEEP=*outfile*]

This internal command calls the APW Linker to process one or more files of LinkEd commands.

Note: ALINK is a synonym for ASSEMBLE; you can use the ASSEMBLE or COMPILE commands instead of ALINK if you prefer. The ALINK command accepts all of the parameters of the ASSEMBLE command; however, some of these parameters are ignored by the linker. Only those parameters that are used by the linker are described here. See the ASML command for a complete list of parameters.

option... You can specify as many of the following options as you wish by separating the options with spaces.

+E | -E If you specify +E, when the linker terminates execution due to a fatal error or because you also specified +T, it calls the APW Editor. The editor displays the source file with the offending line on the fifth line on the screen (or as far down on the screen as possible, if the error is in one of the first four lines of the file). If you specify -E and a fatal error occurs, you are returned to the shell's command line or the Exec file that executed the command. The default for this option is +E when the command is executed from the command line and -E when the command is executed from an Exec file.

+L | -L If you specify +L, the linker generates both a listing of the LinkEd source code and a listing (called a **link map**) of the segments in the object file, including the starting address, the length in bytes (hexadecimal) of each segment, and the segment type. If you specify -L, the source-code listing and link map are not produced. The L parameter in this command overrides any LIST and SOURCE commands in the LinkEd source file.

+S | -S If you specify +S, the linker produces an alphabetical listing of all global references in the object file (called a **symbol table**). If you specify -S, the symbol table is not produced. The S parameter in this command overrides the SYMBOL command in the LinkEd source file.

+T | -T If you select +T, any error causes the link to terminate. If you omit this option or select -T, only fatal errors cause immediate termination of the link.

+W | -W If you select +W, the linker stops and waits for a key press when any error occurs in order to give you the opportunity to read the error message and to decide whether to continue (that is, to continue the link in case of a nonfatal error or to return to the shell in case of a fatal error). Press Apple-Period (⌘-.) to halt execution, or press any character key or the Space bar to continue. If you omit this option or select -W, execution continues without pausing when an error occurs.

file1 file2 ... The full pathnames or partial pathnames (including the filenames) of one or more LinkEd source files. The files are processed in the sequence in which they are listed. Each source file performs a separate link and creates a separate load file. To use multiple LinkEd files to create a single load file, use the LinkEd APPEND and COPY commands.

KEEP=*outfile* You can use this parameter to specify the pathname or partial pathname of the load file. There must not be any spaces between KEEP and the equal sign (=).

In order to use the KEEP parameter when you specify multiple LinkEd source filenames on the command line, you must use a wildcard character in the filename. Two wildcard characters are available for this purpose: % and $. The percent sign (%) is replaced with the pathname of the LinkEd file. The dollar sign ($) is replaced with the pathname of the LinkEd file but with the last extension removed. For example, assume you execute the following command:

```
ALINK LINK1 LINK2 KEEP =%.O
```

The shell uses the name LINK1.O for the load file created by executing the LinkEd file LINK1 and the name LINK2.O for the load file created by the LinkEd file LINK2. Similarly, if you execute the command ALINK MYFILE.LNK KEEP=$, the shell uses the name MYFILE for the load file.

Important: Remember the following points regarding the KEEP parameter:

- If you have a KEEP command in the LinkEd file and you also use the KEEP parameter, the KEEP parameter on the command line takes precedence.
- You can specify a default load filename by using the KeepName shell variable. Shell variables are described in the section "Variables" later in this chapter.
- If you use neither the KEEP parameter, the KeepName shell variable, nor the KEEP command, no load file is produced.
- To use the KEEP parameter with multiple LinkEd source files, you must use a wildcard character in the KEEP parameter.
- Because ProDOS 16 does not allow filenames longer than 15 characters, you must be careful not to specify a filename in the KEEP parameter that will result in a load filename longer than 15 characters. For example, if you specify KEEP=%.LOADFILE and the LinkEd name is LONGNAME, the link will fail when the shell tries to open the file LONGNAME.LOADFILE, which has 17 characters.
- If a load file named *outfile* already exists, it is overwritten without a warning when this command is executed.
- If a source file named *outfile* already exists, APW will not let you overwrite it and the link will fail.

The output listing of the link is sent to the screen unless you redirect output to the printer or use the PRINTER ON LinkEd command. Output redirection is described in the section "Redirecting Input and Output" earlier in this chapter.

Important: If you do not need to take advantage of the advanced link capabilities provided by LinkEd, do *not* create a LinkEd file, and do not use the ALINK command. Instead, use one of the following commands to link your program: LINK, ASML, ASMLG, CMPL, CMPLG, or RUN. The linker is described in detail in Chapter 5.

ASM65816

ASM65816

This language command sets the shell default language to APW 65816 assembly language.

ASML

ASML [*option ...*] *file1* [*file2*] [...] [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2*] [...])] [*language1*=(*option ...*)
[*language2*=(*option ...*)] [...]]

This internal command assembles (or compiles) one or more source files and links one or more object and library files. The APW Shell checks the language of each source file and calls the appropriate assembler or compiler. If the maximum error level returned by each assembler or compiler is less than or equal to the maximum allowed (0 unless you specify otherwise with the MERR directive or its equivalent in the source file), the standard linker is called to link the resulting object files plus any other object and library files named on the ASML command line.

The CMPL command is an alias for ASML.

Note: Not all compilers or assemblers make use of all the parameters provided by this command (or by the ASSEMBLE, ASMLG, COMPILE, CMPL, CMPLG, and RUN commands, which use the same parameters). The APW Assembler, for example, includes no language-specific options, and so makes no use of the *language*=(*option ...*) parameter. If you include a parameter that a compiler or assembler cannot use, it ignores it; no error is generated.

If you include more than one source file, or use APPEND directives to tie together source files in more than one language, then all parameters are passed to every compiler. Each compiler uses those parameters that it recognizes. See the reference manual for the compiler you are using for a list of the options that it accepts.

Note: Command-line parameters (those described here) override source-code options when there is a conflict.

option... You can specify as many of the following options as you wish by separating the options with spaces.

`+E | -E` If you specify `+E`, when the compiler terminates execution due to a fatal error, it calls the APW Editor. The editor displays the source file with the offending line on the fifth line on the screen (or as far down on the screen as possible, if the error is in one of the first four lines of the file). If you specify `-E` and a fatal error occurs, you are returned to the shell's command line or the Exec file that executed the command. The default for this option is `+E` when the command is executed from the command line and `-E` when the command is executed from an Exec file.

`+L | -L` If you specify `+L`, the assembler or compiler generates a source listing; if you specify `-L`, the listing is not produced. The `L` parameter in this command overrides the `LIST` directive in the source file.

`+S | -S` If you specify `+S`, the linker produces an alphabetical listing of all global references in the object file; the assembler or compiler may also produce a symbol table. The APW Assembler, for example, produces an alphabetical listing of all local symbols following each `END` directive. If you specify `-S`, these symbol tables are not produced. The `S` parameter in this command overrides the `SYMBOL` directive in the source file.

`+T | -T` If you select `+T`, any error causes the compile to terminate. If you omit this option or select `-T`, only fatal errors cause immediate termination of the compile. Note that if you select both `+T` and `+E`, any error causes the shell to call the APW Editor and display the offending line as the fifth line on the screen.

`+W | -W` If you select `+W`, the compiler stops and waits for a key press when any error occurs, to give you the opportunity to read the error message and to decide whether to continue (that is, to continue the compile in case of a nonfatal error or to call the editor in case of a fatal error). Press Apple-Period (⌘-.) to halt execution, or press any character key or the Space bar to continue. If you omit this option or select `-W`, execution continues without pausing when an error occurs.

file1 file2 ... The full pathnames or partial pathnames (including the filenames) of the source files to be assembled (or compiled). You can also include the full pathnames or partial pathnames, minus filename extensions, of additional object and library files to be passed on to the linker. You may include as many source, object, and library files as you choose, but at least one of the files must be a source file. Separate the filenames with spaces.

The source files do not all have to have the same APW language type. Note, however, that if you include a LinkEd file, it must be the last file listed. This is because once the advanced linker has been called by a LinkEd file, the linker is not called again regardless of how many source or object files follow the LinkEd file.

The first object file you list must have a .ROOT file; for the other object files, either a .ROOT file or a .A file must be present. On the command line, use the filename without any .ROOT or alphabetical extension. For example, the program TEST might consist of object files named TEST.ROOT, TEST.A, and TEST.B, all in directory /APW/MYPROG/. In this case, you would use /APW/MYPROG/TEST for the object-file filename.

Any library files specified are searched in the order listed. If a library file is listed before one or more object or source files, the library file is searched before those files are linked. Only the segments needed to resolve references that haven't already been resolved are extracted from library files. See the discussion of the MAKELIB command in this chapter for more information on library files.

The ASML command is equivalent to an ASSEMBLE command followed by a LINK command. During the assembly stage, the source files are compiled or assembled as if they had been listed in an ASSEMBLE command; the object and library files listed on the command line are ignored. The source filenames are replaced with the root filenames of the object files created from those source files; then the entire list of filenames is sent to the standard linker as if they were listed in a LINK command.

Important: If there is a source file on the disk with the same name as a root filename you list in this command, the source file will be compiled and the object file you had intended to use will be overwritten or ignored. For example, if the object file MYFILE.ROOT and the source file MYFILE are both on the disk and you include the filename MYFILE on the command line with the intention of linking MYFILE.ROOT, the shell will find MYFILE during the assembly stage of this command and assemble it instead.

KEEP=*outfile* You can use this parameter to specify the pathname or partial pathname (including the filename) of the output file. There must not be any spaces between KEEP and the equal sign (=).

For a one-segment program, the assembler or compiler names the object file *outfile*.ROOT. If the program contains more than one segment, the assembler places the first segment in *outfile*.ROOT and the other segments in *outfile*.A. If this is a partial assembly (or several source files with different programming languages are being compiled), other filename extensions may be used; see the section "Partial Assemblies or Compiles" in this chapter.

If the assembly is followed by a successful link, the load file is named *outfile*.

In order to use the KEEP parameter when you specify multiple source filenames on the command line, you must use a wildcard character in the filename. Two wildcard characters are available for this purpose: % and $. The percent sign (%) is replaced with the pathname of the source file. The dollar sign ($) is replaced with the pathname of the source file with the last extension removed. For example, assume you execute the following command:

```
ASML MYFILE YURFILE KEEP =%.O
```

The shell uses the name MYFILE.O.ROOT for the first object file created from the source file MYFILE and the name YURFILE.O.ROOT for the first object file created from the source file YURFILE. Similarly, if you execute the command
ASML MYFILE.C KEEP=$, the shell uses the name MYFILE.ROOT for the first object file.

In the case of multiple source files, if the assembly is followed by a successful link, the load file is given the root name of the first object file linked.

Important: Keep the following points in mind regarding the KEEP parameter:

- If you have a KEEP directive in the source file and you also use the KEEP parameter, the parameter on the command line takes precedence.
- You can specify a default filename for object files by using the KeepName shell variable. Shell variables are described in the section "Variables" later in this chapter.
- The rules by which the shell names load files, in order of precedence, are as follows:
  - If the KEEP parameter is specified on the command line, that name is used. If the KEEP parameter contains a wildcard character, the root name is based on the name of the first object file created. (Note that, if the first filename on the command line is an object file, that is the first file *linked*, but it was not the first object file *created* by the ASML command, so its root name is not used for the load file.)
  - If there is no KEEP parameter, the root name specified by KeepName is used for the load filename. If the KeepName variable includes a wildcard character, the load filename is based on the name of the first object file created.
  - If KeepName has not been set, the root name of the first object file as determined by a directive in the source file is used.
  - If you use neither the KEEP parameter, the KEEP directive, nor the KeepName shell variable, then the object files are not saved at all. In this case, the link cannot be performed, because there is no object file to link.
- To use the KEEP parameter with multiple source files, you must use one or more wildcard characters in the KEEP parameter.
- Because ProDOS 16 does not allow filenames longer than 15 characters, you must be careful not to specify a filename in the KEEP parameter that will result in an output filename longer than 15 characters. For example, if you specify KEEP=%.OUT and the source filename is LONGNAME, the compile will fail when the shell tries to open the file LONGNAME.OUT.ROOT, which has 17 characters.
- If *object* files with the root filename *outfile* already exist, they are overwritten without a warning when this command is executed. Similarly, if a load file named *outfile* already exists, it is overwritten without a warning when the program is linked.
- If a source file named *outfile* already exists, APW will not let you overwrite it and the link will fail.
- The linker may attempt to link any file in the same prefix as *outfile* that has the root filename *outfile* and ends in an alphabetic suffix. For example, suppose *outfile* is named OUTFILE and there is already a file named OUTFILE.B in the same prefix.

When you execute the ASML command, the assembler creates the files OUTFILE.ROOT and OUTFILE.A; then the Linker attempts to link OUTFILE.B along with the other files. Make sure no such files are present in the prefix of *outfile* before executing the ASML command.

NAMES=(*seg1 seg2* ...) This parameter causes the assembler or compiler to perform a partial assembly or compile; the operands *seg1, seg2, ...* specify the names of the segments to be assembled or compiled. There must not be any spaces between NAMES and the equal sign (=). Separate the segment names with one or more spaces. The APW Linker automatically selects the latest version of each segment when the program is linked.

In case-sensitive languages, segment names must be entered exactly as they appear in the source code. Segment names in case-insensitive languages must be entered as all uppercase characters.

The object file created when you use the NAMES parameter contains only the specified segments. In assembly language, you assign names to segments with START, DATA, PRIVATE, or PRIVDATA directives. In most high-level languages, each subroutine becomes an object segment and the segment name is the same as the subroutine name.

You must use the same output filename for every partial compile or assembly of a program. For example, if you specify the output filename as OUTFILE for the original assembly of a program, the assembler creates object files named OUTFILE.ROOT and OUTFILE.A. In this case you must also specify the output filename as OUTFILE for the partial assembly. The new output file is named OUTFILE.B and contains only the segments listed with the NAMES parameter. When you link a program, the linker scans all the files whose filenames are identical except for their extensions and takes the latest version of each segment.

If you include more than one source-file filename on the command line, the complete list of segment names in the NAMES parameter is used for each source file. Thus, for example, if you list two source-file filenames and include the parameter NAMES=(TOM DICK HARRY) on the command line, then a partial assembly or compile is done for each of the two source files and each of the source files is searched for segments TOM, DICK, and HARRY.

Note: No spaces are permitted immediately before or after the equal sign in the NAMES parameter.

See the section "Partial Assemblies or Compiles" earlier in this chapter for more information on partial assemblies.

language1=(*option ...*) ... This parameter allows you to pass parameters directly to specific APW compilers or assemblers. For each compiler or assembler for which you want to specify options, type the name of the language (exactly as shown by the SHOW LANGUAGES command), an equal sign (=), and the string of options enclosed in parentheses. The contents and syntax of the options string is specified in the compiler or assembler reference manual. Note that the APW Shell does no error checking on this string before passing it through to the compiler or assembler. You can include option strings in the command line for as many languages as you wish; if a language compiler is not called, the string for that language is ignored.

Note: No spaces are permitted immediately before or after the equal sign in this parameter.

When you execute the ASML command, the following sequence of events occurs:

1. The shell calls the assembler or compiler that corresponds to the APW language type of the first source file listed on the command line. For example, if the APW language type of the first source file is ASM65816, the shell calls the APW Assembler.
2. The assembler or compiler processes the source file. One or more object files are created, assuming that you provided a filename for it and that no errors are found with error level greater than that set by MERR.
3. The assembler or compiler passes control back to the shell, which calls the appropriate assembler or compiler to process the next source file. This process is repeated until all source files have been compiled.
4. The shell calls the standard linker and passes to it the root filenames of all the object files to be linked, together with any library files listed on the command line. The object files include those created in steps 2 and 3 plus any object files you listed on the command line.
5. The linker links the object files and searches the library files in the sequence in which the files were named on the command line. For example, suppose the command line included the source file MYPROG, the object-file root filename MYOBJ, and the source file MYCPROG, as follows:

   ```
   ASML MYPROG MYOBJ MYCPROG
   ```

   Assume further that MYPROG generates the object files MYPROG.ROOT and MYPROG.A, that MYCPROG generates the object file MYCPROG.ROOT, and that the object files MYOBJ.ROOT, MYOBJ.A, and MYOBJ.B are present on the disk. In that case, the linker would process these files in the following sequence:

   ```
   MYPROG.ROOT
   MYPROG.A
   MYOBJ.ROOT
   MYOBJ.A
   MYOBJ.B
   MYCPROG.ROOT
   ```

6. Any library files listed on the command line are searched for unresolved references (if any) in the sequence in which those files are listed. For instance, assume the example in step 5 had included the library file MYLIB, as follows:

   ```
   ASML MYPROG MYOBJ MYLIB MYCPROG
   ```

In that case, the linker would process the files in the following sequence:

```
MYPROG.ROOT
MYPROG.A
MYOBJ.ROOT
MYOBJ.A
MYOBJ.B
MYLIB
MYCPROG.ROOT
```

7. If there are still any unresolved references, the library files in the library prefix (prefix 2) are searched.
8. The linker creates a load file. The filename is the same as the root filename of the first object file created. It is determined by KEEP parameter in the command line; if there is is no KEEP parameter, the KeepName shell variable is used; if no KeepName variable has been set, the KEEP directive in the first source file is used.

Press Apple-Period (⌘-.) to stop the assembly or compile after it has begun. The assembler or compiler may respond by halting execution and calling the editor with the first line of your source file at the top of the screen, or it may return you to the shell.

Listings and error messages are sent to the screen unless you either include a PRINTER ON directive (or equivalent) in the source file or redirect output to a disk file or the printer. Output redirection is described in the section "Redirecting Input and Output" earlier in this chapter.

The following command assembles and links a source file named MYFILE and writes the load file to disk as the file MYPROG. No source listing or symbol table is produced unless called for by directives in MYFILE:

```
ASML MYFILE KEEP=MYPROG
```

The following command also assembles and links a source file named MYFILE and writes the load file to disk as the file MYPROG. A symbol table is produced, but no source listing is generated regardless of whether one is called for by directives in MYFILE:

```
ASML -L +S MYFILE KEEP=MYPROG
```

The following command assembles the segments TOOLCALL and TEXT_OUT in the source file named MYFILE, links the program, and writes the load file to disk as the file MYPROG:

```
ASML MYFILE KEEP=MYPROG NAMES=(TOOLCALL TEXT_OUT)
```

The following command assembles the source file named MYFILE.SRC. If MYFILE.SRC or a file appended to MYFILE.SRC is a C program, the C-compiler option that adds a prefix to the include-file path list is passed to the C compiler. Object files are named MYFILE.ROOT, MYFILE.A, and so on. After the program is assembled or compiled, it is linked and the load file is written to disk as the file MYFILE:

```
ASML MYFILE.SRC KEEP=$.EXE CC=(-I/APW/LIBRARIES/CINCLUDE)
```

The following command assembles the ASM65816 source file named MYFILE and compiles the C source file named MYCFILE. The object files created are saved with names specified by the KeepName variable or by KEEP directives in the source files. After the programs are assembled and compiled, the object files created from MYFILE, the object files with the root name MYOBJ, and the object files created from MYCFILE are linked. The name of the load file is the root filename of the first object file created:

```
ASML MYFILE MYOBJ MYCFILE
```

The following command assembles the source file MYFILE. The object files created are saved with the root filename MYPROG. After the program is assembled, the object files created from MYFILE and the object files with the root name MYOBJ are linked. If the linker cannot resolve all references, it searches the library file MYLIB. The load file is written to disk with the filename MYPROG:

```
ASML MYFILE MYOBJ MYLIB KEEP=MYPROG
```

Note: If you have appended a LinkEd file to the end of your program, the link is controlled by the commands in the LinkEd file. In this case, the standard linker is not called, and the operation of the ASML command is identical to that of the ASSEMBLE command.

For more examples and discussion of the use of the ASML command, see the section "Compiling (or Assembling) and Linking a Program" in Chapter 2.

ASMLG

ASMLG [*option* ...] *file1* [*file2*] [...] [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2*] [...])] [*language1*=(*option* ...)
[*language2*=(*option* ...)] [...]]

This internal command assembles (or compiles) one or more source files, links one or more object and library files, and runs the resulting load file. Its function is identical to that of the ASML command, except that once the program has been successfully linked, it is executed automatically. See the ASML command for a list of options and a description of the parameters.

The CMPLG and RUN commands are aliases for ASMLG.

ASSEMBLE

ASSEMBLE [*option* ...] *file1* [*file2*] [...] [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2*] [...])] [*language1*=(*option* ...)
[*language2*=(*option* ...)] [...]]

This internal command assembles (or compiles) one or more source files. Its function is identical to that of the ASML command, except that the ASSEMBLE command does not call the linker to link the object files it creates; therefore, no load file is generated. You can use the LINK command or a LinkEd file to link the object files created by the ASSEMBLE

command. See the ASML command for a list of options and a description of the parameters.

The COMPILE command is an alias for ASSEMBLE.

BREAK

BREAK

This internal command is used to terminate a FOR or LOOP statement. See the section "Exec Files" later in this chapter for a more complete discussion of this command.

CANON

CANON [+A|-A] [+C *n*] [+S|-S] *dictionary* [*inputfile*]

This utility compares the spelling of words in the input file with words in the dictionary file. Any words in the input file that are included in the dictionary file are replaced with the canonical spelling specified in the dictionary file. The result is written to standard output (by default, the screen).

+A|-A If you specify +A, the following characters are treated like letters by Canon:
$ % @
If you specify -A or omit this parameter, these characters cannot be included in the character strings in the dictionary, except as leading characters for search strings (as explained below).

+C *n* If you specify +C followed by a number, only *n* characters are considered significant when matching patterns. If you do not specify this parameter, all characters are considered significant. Note that there must be space between +C and *n*.

+S|-S If you specify +S, pattern matching is case sensitive. If you specify -S or omit this parameter, pattern matching is not case sensitive.

dictionary The full pathname or partial pathname (including the filename) of the dictionary file.

inputfile The full pathname or partial pathname (including the filename) of the input file. You can use wildcard characters in the filename.

Canon works in a manner similar to a global search and replace function in a text editor, except that any number of different character strings can be searched for simultaneously. The dictionary file is a text file that specifies the character strings to be replaced and the replacement strings (the canonical spellings).

The maximum length of a line in the dictionary file is 256 characters. Each string must begin with a letter and can contain any number of letters and numerals. The underscore character (_) is considered a letter by Canon. If you specify the +A option, the dollar sign ($), percent sign, (%), and at sign (@) are also considered letters.

Each line of the dictionary file can contain either one or two strings. If a line contains one string, Canon uses that string as both the search and replace string. For example, suppose the dictionary contains the following line:

```
main
```

If your search is not case sensitive (that is, the command line does not include the +S parameter), the following strings are all converted to `main`:

```
Main
MAIN
mAIN
```

If your search is restricted to four characters (that is, the command line includes the +C 4 parameter), the following strings are all converted to `main`:

```
mainstream
mainly
maintenance
```

If a line of the dictionary file contains two strings, Canon searches for the first string and replaces it with the second. For example, suppose the dictionary contains the following line:

```
Main main
```

If your search is not case sensitive (that is, the command line does not include the +S parameter), this line functions exactly like a line containing the single string `main`. If the search is case sensitive, however, only the string `Main` is converted to `main`. In this case, the following strings would *not* be converted to `main`:

```
MAIN
mAIN
MaiN
```

The search string can include a prefix consisting of any number of characters that are not recognized as letters or numerals by Canon. Canon replaces only those strings that match the entire search string, including the prefix, but does not replace the prefix. For example, suppose the dictionary contains the following line:

```
.seconds tenths
```

In this case, Canon would convert the string `hours.minutes.seconds` to `hours.minutes.tenths`. The string `hours/minutes/seconds` would not be changed, however.

Canon writes the converted file to standard output (by default, the screen). To save the result in a file, you must redirect output to a pathname. For example, to process the file `MYPROG.SRC` with the dictionary `C.CONVERT`, creating the new file `MYPROG.CC`, you could use the following command line:

```
CANON C.CONVERT MYPROG.SRC > MYPROG.CC
```

CAT

CAT [*pathname* ...]

This internal command is an alias for CATALOG.

CATALOG

CATALOG [*pathname* ...]

This internal command lists to the screen the directory of the volume or subdirectory you specify.

pathname The pathname or partial pathname of the volume, directory, subdirectory or file for which you want directory information. You can include any number of pathnames; the directory for each pathname is listed in turn. If you include a filename, you can use wildcard characters in the filename.

For example, to list the entire contents of the current directory, use the following command:

```
CATALOG
```

To list the entire contents of the subdirectory /APW/UTILITIES/, use the following command:

```
CATALOG /APW/UTILITIES
```

To get directory information about the MAKELIB file in the UTILITIES/ subdirectory when the current prefix is /APW/, use the following command:

```
CATALOG UTILITIES/MAKELIB
```

To list every file beginning with M or N in the UTILITIES/ subdirectory, use the following command:

```
CATALOG /APW/UTILITIES/M= /APW/UTILITIES/N=
```

Or, for example, if /APW/UTILITIES/ were the current directory, you could use the following command to achieve the same result:

```
CATALOG M= N=
```

See the section "Listing the Directory" in Chapter 2 for a description of the fields in the directory listing. A list of ProDOS 16 file types is given in Table 3.4 in the discussion of the FILETYPE command.

CC

CC

This language command sets the shell default language to APW C.

CHANGE

CHANGE *pathname language*

This internal command changes the language type of an existing file.

pathname The full pathname or partial pathname (including the filename) of the source file whose language type you wish to change. You can use wildcard characters in the filename.

language The language type to which you wish to change this file.

In APW, each source or text file is assigned the current default language type when it is created. When you assemble or compile the file, APW checks the language type to determine which assembler, compiler, linker, or text formatter to call. Use the CATALOG command to see the language type currently assigned to a file. Use the CHANGE command to change the language type to any of the languages listed by the SHOW LANGUAGES command. The section "Command Types and the Command Table" earlier in this chapter includes a discussion of language types and language commands.

You can use the CHANGE command to correct the APW language type of a file if the editor was set to the wrong language type when you created the file, for example. Another use of the CHANGE command is to assign the correct APW language type to an ASCII text file (ProDOS 16 file type $04) created with another editor.

CMPL

CMPL [*option ...*] *file1* [*file2*] [...] [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2*] [...])] [*language1*=(*option ...*)
[*language2*=(*option ...*)] [...]]

This internal command compiles (or assembles) one or more source files and links one or more object and library files. Its function, options, and parameters are identical to those of the ASML command. See your compiler manual for the language-specific options available.

The CMPL command is an alias for ASML.

CMPLG

CMPLG [*option ...*] *file1* [*file2*] [...] [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2*] [...])] [*language1*=(*option ...*)
[*language2*=(*option ...*)] [...]]

This internal command compiles (or assembles) one or more source files, links one or more object and library files, and runs the resulting load file. See the ASML command for a list of options and a description of the parameters. See your compiler manual for the language-specific options available.

The CMPLG and RUN commands are aliases for ASMLG.

COMMANDS

COMMANDS *pathname*

This internal command causes APW to read a command-table file, resetting all the commands to those in the new command table.

pathname The full pathname or partial pathname (including the filename) of the file containing the command table.

When you load APW, it reads the command-table file named SYSCMND in prefix 4. You can use the COMMANDS command to read in a custom command table at any time. Command tables are described in the section "Command Types and the Command Table" earlier in this chapter.

Note: The shell does no error checking when it executes the COMMANDS command. Any error in your custom command table will not show up until you try to execute the command that is defined in the line that contains the error.

COMMENT

COMMENT [*text*]

This internal command, or an asterisk (*) is used to enter comments into Exec files. There must be a space between the COMMENT command and the comment. See the section "Exec Files" later in this chapter for a more complete discussion of this command.

COMPACT

COMPACT *infile* [-O *outfile*] [-P] [-R] [-S]

This utility converts a load file to the most compact form provided for by the object module format.

infile The full pathname or partial pathname (including the filename) of the load file that you wish to compact. You can use wildcard characters in the filename.

outfile The full pathname or partial pathname (including the filename) of the output file. If you do not specify `-O` *outfile*, then *infile* is overwritten.

`-P` If you specify the `-P` option, a progress report is written to standard output. The progress report first shows the version number of Compact that you are using, and then shows the number of the segment being processed and the operation being performed on that segment.

`-R` If you specify the `-R` option, any load segment named `~globals` or `~arrays` is made a reload segment.

`-S` If you specify the `-S` option, a summary report is written to standard output when Compact is finished. The summary report shows the total number of segments in the file and the number of each OMF record type compacted, copied, or created.

Press Apple-Period (⌘-.) to cancel the command.

The Compact utility can decrease the size of load files by 20 percent to 70 percent and make them load up to 25 percent faster. The amount of reduction in size and loading time achieved for a particular file depends on the number and nature of symbolic references in the file.

In addition to compacting a file, the Compact utility converts OMF version 1.0 files to version 2.0. If you specify the `-R` option and Compact finds a load segment named `~globals` or `~arrays`, Compact makes it a reload segment. See Chapter 7 for a description of OMF 2.0 and reload segments, and the section "Command Types and the Command Table" earlier in this chapter for a discussion of restartability.

Important: In order to load a compacted load file, you must have Version 1.2 or later of ProDOS 16 and the System Loader.

Use Compact as the last step in program development, after the program has been completely debugged, to maximize the performance and minimize the size of the load file.

COMPILE

`COMPILE` [*option ...*] *file1* [*file2 ...*] [`KEEP=`*outfile*]
[`NAMES=(`*seg1* [*seg2* [...]])] [*language1*`=(`*option ...*`)`
[*language2*`=(`*option ...*`)` [...]]]

This internal command compiles (or assembles) one or more source files. Its function is identical to that of the `ASML` command, except that it does not call the linker to link the object files it creates; therefore, no load file is generated. You can use the `LINK` command or a LinkEd file to link the object files created by the `COMPILE` command. See the `ASML` command for a list of options and a description of the parameters. See your compiler manual for the language-specific options available.

The `COMPILE` command is an alias for `ASSEMBLE`.

CONTINUE

```
CONTINUE
```

This internal command causes control to skip over following statements to the next `END` statement that isn't the `END` for an `IF` statement. See the section "Exec Files" in this chapter for a more complete discussion of this command.

COPY

`COPY [-C]` *pathname1* `[`*pathname2*`]`
`COPY [-D]` *volume1 volume2*

This internal command copies a file to a new subdirectory or to a duplicate file with a different filename. This command can also be used to copy an entire directory or to perform a block-by-block disk copy.

–C If you specify `-C` before the first pathname, `COPY` does not prompt you if the target filename (*pathname2*) already exists.

pathname1 The full or partial pathname (including the filename) of the file or directory to be copied. Wildcard characters can be used in the filename. If you do include wildcard characters, both files and subdirectories that match *pathname1* are copied.

pathname2 The full or partial pathname (including the filename) to be given to the copy of the file or to the directory to which the file is to be copied. Wildcard characters cannot be used in this filename. If *pathname1* does not include wildcard characters, all of the directories and subdirectories in *pathname2* must already exist. If *pathname1* does include wildcard characters and the last filename in *pathname2* does not exist, a subdirectory with that name is created. If *pathname1* and *pathname2* are both directories, a subdirectory named *pathname1* is created in *pathname2* and the entire directory (including all the files, subdirectories, and files in the subdirectories) is copied into it. If you omit *pathname2*, the current directory is used and the new file has the same name as the file being copied.

`-D` If you specify `-D` before the first pathname, both pathnames are volume names, and both volumes are the same size, then a block-by-block disk copy is performed.

> **Note:** A block-by-block disk copy is much faster than a file-by-file copy. Use the `-D` option whenever you can.

volume1 The name of a volume that you want to copy onto another volume. The entire volume (including all the files, subdirectories, and files in the subdirectories) is copied. If both pathnames are volume names, both volumes are the same size, *and* you specify the `-D` parameter, then a block-by-block disk copy is performed. You can use a device name (such as `.D1`) instead of a volume name.

volume2 The name of the volume that you want to copy onto. You can use a device name instead of a volume name.

> **Warning:** Under certain conditions, COPY can perform a recursive copy. For example, if CDIR is a subdirectory of BDIR, which in turn is a subdirectory of /ADIR, and the current directory is BDIR, then the following command copies /ADIR and all of its subdirectories into CDIR: COPY -C /ADIR CDIR. Because CDIR is a subdirectory of /ADIR, it is copied into itself, then all of the subdirectories of /ADIR, including the new copy of CDIR, are copied into the new CDIR, and so forth.

Unless you specify the -D parameter, the COPY command copies individual files. If a file with the same filename as one you are trying to copy exists in the target subdirectory, you are asked if you want to replace the target file. Type Y and press Return to replace the file. Type N and press Return to copy the file to the target prefix with a new filename. In the latter case, you are prompted for the new filename. Enter the filename, or press Return without entering a filename to cancel the copy operation. If you specify the -C parameter, the target file is replaced without prompting.

> **Note:** If you do not include any parameters after the COPY command, you are prompted for a pathname, since APW prompts you for any required parameters. Since the target prefix and filename are not required parameters, however, you are *not* prompted for them. Consequently, the current prefix is always used as the target directory in such a case. To copy a file to any subdirectory *other* than the current one, you *must* include the target pathname as a parameter either in the command line or following the pathname entered in response to the `Source file name` prompt.

If you use volume names for both the source and target and specify the -D parameter, the COPY command copies one volume onto another. In this case, the contents of the target disk are destroyed by the copy operation. The target disk must be initialized as a ProDOS 16 volume (use the INIT command) *before* this command is used. Because this command performs a block-by-block copy, it makes an exact duplicate of the disk. Therefore, both disks must be the same size for this command to work. You can use device names rather than volume names to perform a disk copy; device names are described in the section "Using Device Names" in Chapter 2.

The following command makes a copy of the file FILEA on the current prefix, gives the copy the filename FILEB, and places it in the same prefix:

```
COPY FILEA FILEB
```

Notice that trailing slashes (/) are not significant to ProDOS 16; they are stripped by the shell. If a target directory already exists, ProDOS recognizes it as a directory. If a target filename does not already exist and you are copying a single file to it, the target is treated as a filename; if you are copying more than one file to it, the target is treated as a directory name and a directory by that name is created.

Assume, for example, that you have a directory named PROGRAMS/ on your disk. In this case, the following command copies the file MYPROG from the directory APW/ into the subdirectory /APW/PROGRAMS/ without changing the name of MYPROG:

```
COPY /APW/MYPROG /APW/PROGRAMS/
```

If there is no file or directory named /APW/PROGRAMS/ on the disk, however, this same command copies the file /APW/MYPROG to another file named /APW/PROGRAMS. On the other hand, the following command copies all the files and subdirectories that begin with the string MYPROG from the directory /APW/ into the directory /APW/PROGRAMS/:

```
COPY /APW/MYPROG= /APW/PROGRAMS/
```

If PROGRAMS/ does not already exist, it is created by this command.

The following command copies the subdirectory /APW/UTILITIES/HELP/ into the subdirectory /HARDISK/DOCUMENTS/HELP/:

```
COPY /APW/UTILITIES/HELP/ /HARDISK/DOCUMENTS
```

An error results if /DOCUMENTS/ does not already exist.

The following command performs a block-by-block disk copy of the volume /APW onto the volume in disk drive .D2:

```
COPY -D /APW/ .D2
```

CREATE

CREATE *directory1* [*directory2* ...]

This internal command creates a new subdirectory.

directory1 directory2 ... The pathnames or partial pathnames of the subdirectories you wish to create.

CRUNCH

CRUNCH *rootname*

This external command combines the object files created by partial assemblies or compiles into a single object file. For example, if an assembly and subsequent partial assemblies have produced the object files FILE.ROOT, FILE.A, FILE.B, and FILE.C, then the CRUNCH command combines FILE.A, FILE.B, and FILE.C into a new file called FILE.A, deleting the old object files in the process. The new FILE.A contains only the latest version of each segment in the program. New segments added during partial assemblies are placed at the end of the new FILE.A.

rootname The full pathname or partial pathname, including the filename but minus any filename extensions, of the object files you wish to combine. For example, if your object files are named FILE.ROOT, FILE.A, and FILE.B in subdirectory /HARDISK/MYFILES/, use /HARDISK/MYFILES/FILE for *rootname*.

All files with the root filename you specify and an alphabetic suffix must be object files. For example, if the object files FILE.ROOT, FILE.A, and FILE.B are present together with the source file FILE.C, an error occurs when the Crunch utility tries to process

FILE.C. In addition, there must be no gaps in the sequence. For example, if you have the object files FILE.ROOT, FILE.A, FILE.B, and FILE.D, the Crunch utility cannot find FILE.D.

Use the DUMPOBJ command to obtain a listing of the segments in any object or load file. See the section "Partial Assemblies or Compiles" earlier in this chapter for more information on partial assemblies.

DEBUG

DEBUG

This external command calls the Apple IIGS Debugger if it is present in the utility subdirectory (prefix 6). If you do not have the debugger, you get a message informing you that the debugger is not available.

The debugger is described in detail in the *Apple IIGS Debugger Reference*.

DELETE

DELETE [-C] *pathname1* [*pathname2* ...]

This internal command deletes the file or files you specify.

-C If you specify -C before the pathname, DELETE does not prompt you before deleting the contents of a directory.

pathname1 pathname2 ... The full pathnames or partial pathnames (including the filenames) of the files to be deleted. Wildcard characters may be used in the filenames.

To delete all the contents of a directory, use the pathname of the directory followed by an equal-sign (=) wildcard character. For example, to delete the contents of the directory /MYFILES/BACKUPS/, use the following command:

```
DELETE /MYFILES/BACKUPS/=
```

When you do so, the prompt Are you sure? appears on the screen. Type Y and press Return to execute the command. Type N and press Return to abort the command. To suppress the prompt, use the -C parameter with the DELETE command.

You cannot delete a directory that is not empty; you must delete the contents of the directory first, and then delete the directory. To delete the directory /MYFILES/BACKUPS/ together with all its contents, suppressing the Are you sure? prompt, for example, use the following commands:

```
DELETE -C /MYFILES/BACKUPS/=
DELETE /MYFILES/BACKUPS
```

DISABLE

`DISABLE D|N|W|R` *pathname1* `[`*pathname2 ...*`]`

This internal command disables one or more of the access attributes of a ProDOS 16 file.

`D` "Delete" privileges. If you disable this attribute, the file cannot be deleted.

`N` "Rename" privileges. If you disable this attribute, the file cannot be renamed.

`W` "Write" privileges. If you disable this attribute, the file cannot be written to.

`R` "Read" privileges. If you disable this attribute, the file cannot be read.

pathname1 pathname2 ... The full pathnames or partial pathnames (including the filenames) of the files whose attributes you wish to disable. You can use wildcard characters in the filenames.

Note: The "backup required" flag cannot be disabled by the `DISABLE` command. This flag can be disabled only by backup programs: that is, programs that create backup copies of files on a disk. When set, the backup required flag indicates that the file has not been backed up since the last time the file was modified.

You can disable more than one attribute at one time by typing the operands with no intervening spaces. For example, to "lock" the file `TEST` so that it cannot be written to, deleted, or renamed, use the command

```
DISABLE DNW TEST
```

Note: In order to protect a file so that it cannot be altered by the editor, you must disable the Delete access attribute. This is because, when the editor saves a file, it first deletes any existing file with that name and then creates a new file with the same name.

Use the `ENABLE` command to reenable attributes you disabled with the `DISABLE` command.

When you use the `CATALOG` command to list a directory, the attributes that are currently enabled are listed in the `Access` field for each file. ProDOS 16 access attributes are described in the *Apple IIGS ProDOS 16 Reference*. Directory listings are described in the section "Listing a Directory" in Chapter 2.

DUMPOBJ

`DUMPOBJ [`*option ...*`]` *pathame* `[NAMES=(`*seg1* `[`*seg2*`] [...])]`

The DumpOBJ utility writes the contents of an object file to standard output (normally the screen). The default format for the listing is object-module-format (OMF) operation codes and records. You can also list the file as a 65816 machine-language disassembly or as hexadecimal codes.

option... You can specify as many of the following options as you wish by separating the options with spaces. If you select two mutually exclusive options (such as +X and +D), the last one listed is used. If an option can't function due to the other options set, it is ignored. For example, if you select -H to suppress segment headers, and also specify -S to select short headers, then the -S is ignored.

- -A Suppress all information but the operation codes and operands for each line of an OMF-format or 65816-format disassembly. The default is to include the displacement into the file and the program counter for each line at the beginning of the line.
- +D Write the file dump as a 65816 disassembly rather than as OMF records.
- -F Suppress the checking of the file type. You can use this option to dump the contents of any file, whether it is in OMF or not. See the following discussion for more information on examining non-OMF files.
- -H If the output format is hexadecimal codes (+X option), this option causes the headers to also be listed as hexadecimal codes. For all other output formats, the headers are not printed at all.
- -I For 65816 disassembly listings, assume that the CPU is set to short index (X and Y) registers at the start of the disassembly, rather than starting in full native mode. This option has no effect on OMF-format and hexadecimal listings.
- -L Don't show the contents of CONST and LCONST records for an OMF-format disassembly. This option lets you see the structure of an OMF file without listing all of the data in the file.
- -M For 65816 disassembly listings, assume that the CPU is set to short memory (accumulator) registers at the start of the disassembly, rather than starting in full native mode. This option has no effect on OMF-format and hexadecimal listings.
- -O Don't show the contents of the segments; that is, list the headers only.
- -S Write only the name of the segment and the segment type for the segment headers. The default is to include all of the information in the segment header.
- +X Write the file dump in hexadecimal codes rather than as OMF records. Segment headers are always printed in ASCII text unless you also select the -H option.

pathname The full pathname or partial pathname (including the filename) of the file you wish to dump. The file may be a library file, the output of an assembler or compiler, a load file, or any other file that conforms to APW object module format. If you use the -F option, you can specify a file of any file type.

seg1 seg2 ... The names of specific segments you wish to dump. If you specify the NAMES parameter, only the segments you specify are processed. To get a list of segments in the file, use the -O and -S options with the DUMPOBJ command. In case-sensitive languages, segment names must be entered exactly as they appear in the source code. Segment names in case-insensitive languages must be entered as all uppercase characters.

Press Apple-Period (⌘-.) to cancel the DumpOBJ listing and return to the shell.

If the file consists of more than one segment, each segment is listed separately. Each segment listing begins with the segment header, followed by the segment body. A typical segment header is shown in Figure 3.4. The fields in the segment header are described in the section "Object Module Format" in Chapter 7.

```
Byte count     : $00000078           120
Reserved space: $00000000             0
Length         : $00000021            33
Label length   : $0A                  10
Number length  : $04                   4
Version        : $02                   2
Bank size      : $00010000         65536
Kind           : $0000       static code segment
Org            : $00000000             0
Alignment      : $00000000             0
Number sex     : $00                   0
Segment number: $0001                  1
Segment entry  : $00000000             0
Disp to names  : $002C                 44
Disp to body   : $0040                 64
Load name      :
Segment name   : SECOND
```

Figure 3.4. Sample DumpOBJ Segment Header

The format in which the body of the segment is shown depends on the option used. The default is to show the contents of each record in the segment in object module format. A typical OMF segment dump is shown in Figure 3.5. The first column shows the actual displacement into the segment, in bytes, of that record. Because the segment header takes up 61 bytes (that is, it ends at byte $3C), the first record in the segment starts at $3D. The second column shows the setting of the program counter for that segment: that is, the cumulative number of bytes that the linker will create in the load file. The third and fourth columns show the record type and operation code of the OMF record shown on that line. The last column shows the contents of the record. Expressions are shown in postfix form: that is, the values being acted on are written first, followed by the operator. OMF records and expressions are described in the section "Object Module Format" in Chapter 7.

Note: The OMF dump is provided to aid in the debugging of compilers. If you are not highly familiar with the OMF, the default DumpOBJ listing will not be of much use to you. You can, however, use the options provided to examine the contents of an object file in machine-language disassembly format or as hexadecimal codes.

```
00003D 000000 | USING    ($E4) | DATA
000043 000000 | CONST    ($03) | 4BABAE
000047 000003 | EXPR     ($EB) | 02 : L:MSG2
000050 000005 | CONST    ($04) | A00000B9
000055 000009 | BEXPR    ($ED) | 02 : MSG2
00005E 00000B | CONST    ($04) | DA5A4820
000063 00000F | BEXPR    ($ED) | 02 : ~COUT
00006D 000011 | CONST    ($0A) | 7AFAC8CAD0F1A9000060
000078 00001B | END      ($00)
```

Figure 3.5. DumpOBJ OMF-Format Segment Body

If you select the +D option, the segment body is displayed in 65816 disassembly format. A typical disassembly segment dump is shown in Figure 3.6. The first column shows the actual displacement into the segment, in bytes, of the first byte in the line. The second column shows the setting of the program counter for that segment: that is, the cumulative number of bytes that the linker will create in the load file. The third column shows the disassembly. The disassembly starts with LONGA and LONGI directives, indicating whether the disassembler is assuming long or short operands for the accumulator and index registers. The APW Assembler is described in the *Apple IIGS Programmer's Workshop Assembler Reference* manual.

> **Note:** The disassembler tries to keep track of REP and SEP instructions, which are used to set bits in the status register. The status register settings determine whether 16-bit (native mode) or 8-bit (emulation mode) index-register (X and Y) and accumulator-register transfers are used by the CPU. Any time the disassembler finds an REP or SEP instruction, it inserts the appropriate LONGA and LONGI directives in the disassembly to indicate the state of the registers. (The LONGA and LONGI directives tell the APW Assembler whether to use long or short operands for transfer instructions.) LONGA and LONGI directives are also placed at the beginning of every segment in the disassembly to indicate the state of the registers on entering the segment. If an expression involving a label was used as the operand of the REP or SEP instruction, the disassembly might lose track of the setting of the status register.

```
00003D 000000 |            LONGA ON
00003D 000000 |            LONGI ON
00003D 000000 | SECOND     START
00003D 000000 |            USING DATA
000043 000000 |            PHK
000045 000001 |            PLB
000046 000002 |            LDX   L:MSG2
00004B 000005 |            LDY   #$0000
00004F 000008 |            LDA   MSG2,Y
000054 00000B |            PHX
000056 00000C |            PHY
000057 00000D |            PHA
000058 00000E |            JSR   ~COUT
00005D 000011 |            PLY
00005F 000012 |            PLX
000060 000013 |            INY
000061 000014 |            DEX
000062 000015 |            BNE   *+$F1
000064 000017 |            LDA   #$0000
000067 00001A |            RTS
000068 00001B |            END
```

Figure 3.6. DumpOBJ Disassembly-Format Segment Body

If you select the +X option, the segment body is displayed in hexadecimal format. A typical hexadecimal segment dump is shown in Figure 3.7. The first column shows the actual displacement into the segment, in bytes, of the first byte in the line. The next four columns show the next 16 bytes in the file. The last column shows the ASCII equivalents of those bytes. The hexadecimal dump starts with the first byte after the segment header (unless you specify the -H option, in which case the segment header is included in the hexadecimal dump), and ends at the last byte before the next segment header. Because all segments in object files start on block (that is, 512-byte) boundaries, the bytes from the END record to the end of the block are meaningless (in Figure 3.7 they contain repetitions of the data in the segment).

```
00003D  |  E4044441 5441034B ABAEEB02 84044D53  |  d DATA K+.k    MS
00004D  |  47320004 A00000B9 ED028304 4D534732  |  G2      9m    MSG2
00005D  |  0004DA5A 4820ED02 83057E43 4F555400  |    ZZH m    ~COUT
00006D  |  0A7AFAC8 CAD0F1A9 00006000 434F4E44  |   zzHJPq)   ` COND
00007D  |  E4044441 5441034B ABAEEB02 84044D53  |  d DATA K+.k    MS
00008D  |  47320004 A00000B9 ED028304 4D534732  |  G2      9m    MSG2
00009D  |  0004DA5A 4820ED02 83057E43 4F555400  |    ZZH m    ~COUT
0000AD  |  0A7AFAC8 CAD0F1A9 00006000 434F4E44  |   zzHJPq)   ` COND
0000BD  |  E4044441 5441034B ABAEEB02 84044D53  |  d DATA K+.k    MS
0000CD  |  47320004 A00000B9 ED028304 4D534732  |  G2      9m    MSG2
0000DD  |  0004DA5A 4820ED02 83057E43 4F555400  |    ZZH m    ~COUT
0000ED  |  0A7AFAC8 CAD0F1A9 00006000 434F4E44  |   zzHJPq)   ` COND
0000FD  |  E4044441 5441034B ABAEEB02 84044D53  |  d DATA K+.k    MS
00010D  |  47320004 A00000B9 ED028304 4D534732  |  G2      9m    MSG2
00011D  |  0004DA5A 4820ED02 83057E43 4F555400  |    ZZH m    ~COUT
00012D  |  0A7AFAC8 CAD0F1A9 00006000 434F4E44  |   zzHJPq)   ` COND
00013D  |  E4044441 5441034B ABAEEB02 84044D53  |  d DATA K+.k    MS
00014D  |  47320004 A00000B9 ED028304 4D534732  |  G2      9m    MSG2
00015D  |  0004DA5A 4820ED02 83057E43 4F555400  |    ZZH m    ~COUT
00016D  |  0A7AFAC8 CAD0F1A9 00006000 434F4E44  |   zzHJPq)   ` COND
00017D  |  E4044441 5441034B ABAEEB02 84044D53  |  d DATA K+.k    MS
00018D  |  47320004 A00000B9 ED028304 4D534732  |  G2      9m    MSG2
00019D  |  0004DA5A 4820ED02 83057E43 4F555400  |    ZZH m    ~COUT
0001AD  |  0A7AFAC8 CAD0F1A9 00006000 434F4E44  |   zzHJPq)   ` COND
0001BD  |  E4044441 5441034B ABAEEB02 84044D53  |  d DATA K+.k    MS
0001CD  |  47320004 A00000B9 ED028304 4D534732  |  G2      9m    MSG2
0001DD  |  0004DA5A 4820ED02 83057E43 4F555400  |    ZZH m    ~COUT
0001ED  |  0A7AFAC8 CAD0F1A9 00006000 434F4E44  |   zzHJPq)   ` COND
0001FD  |  E40444                               |  d D
```

Figure 3.7. DumpOBJ Hexadecimal-Format Segment Body

DumpOBJ can be used to dump the contents of any file, even if it is not in OMF. To dump the contents of a non-OMF file, use the `-H` and `-F` options, together with either the `+X` or `+D` options.

> **Important:** Any other combination of options, or no options, will probably produce unusable results, since in that case DumpOBJ attempts to scan the file for segments as if it were in OMF.

DumpOBJ is extremely useful for debugging compilers and assemblers, but it is also useful whenever you want to see the contents of an OMF file. For example, before using the `SELECT` command in a LinkEd file to extract specific segments from the object file `GOOD.STUFF`, you could use the following command to list the names and segment types of all the segments in the file:

```
DUMPOBJ -S -O GOOD.STUFF
```

DumpOBJ specifies the type of each segment (such as static data, static code, dynamic data, and so forth). Code segments are created by a `START`–`END` pair of directives in an assembly-language source file; data segments are created by a `DATA`–`END` pair. In most high-level languages, each subroutine corresponds to an object segment. Static and dynamic segments are assigned by the linker; you can use LinkEd commands to control these assignments. See Chapter 5 for a discussion of LinkEd commands.

ECHO

ECHO *string*

This internal command lets you write messages to the screen from an Exec file. See the section "Exec Files" earlier in this chapter for a more complete discussion of this command.

EDIT

EDIT *pathname*

This internal command calls the APW Editor and opens a file to edit.

pathname The full pathname or partial pathname (including the filename) of the file you wish to edit. If the file named does not exist, a new file with that name is opened. If you use a wildcard character in the filename, the first file matched is opened.

The APW default file type changes to match the file type of the open file. If you open a new file, that file is assigned the current default file type. Use the CHANGE command to change the file type of an existing file. To change the APW default file type before opening a new file, type the name of the language you wish to use, and press Return.

The editor is described in Chapter 4.

ELSE

ELSE
ELSE IF

This internal command is used as part of an IF command. See the section "Exec Files" later in this chapter for a more complete discussion of this command.

ENABLE

ENABLE D|N|B|W|R *pathname1* [*pathname2* ...]

This internal command enables one or more of the access attributes of a ProDOS 16 file, as follows:

D "Delete" privileges. If you enable this attribute, the file can be deleted.

N "Rename" privileges. If you enable this attribute, the file can be renamed.

B "Backup required" flag. If you enable this attribute, a backup utility program will assume that this file has not been backed up since the last time it was modified.

W "Write" privileges. If you enable this attribute, the file can be written to.

R "Read" privileges. If you enable this attribute, the file can be read.

pathname1 pathname2 ... The full pathnames or partial pathnames (including the filenames) of the files whose attributes you wish to enable. You can use wildcard characters in the filename.

You can enable more than one attribute at one time by typing the operands with no intervening spaces. For example, to "unlock" the file TEST so that it can be written to, deleted, or renamed, use the command

```
ENABLE DNW TEST
```

When a new file is created, all the access attributes are enabled. Use the ENABLE command to reverse the effects of the DISABLE command.

END

END

This internal command terminates a FOR, IF, or LOOP command. See the section "Exec Files" later in this chapter for a more complete discussion of this command.

EQUAL

EQUAL [*option ...*] *pathame1 pathname2*

The Equal utility compares two files or directories for data equality and can show differences in file dates or types.

option You can specify as many of the following options as you wish by separating the options with spaces.

+D | -D If you specify -D, Equal does not compare the creation and modification dates and times of files. If you do not include this option or specify +D, creation and modification dates and times are compared.

+M | -M If you specify -M, Equal does not list the names of missing files: that is, files that are present in one of the directories you listed but not in the other. If you do not include this option or specify +M, missing files are listed.

+N *n* Display the first *n* mismatched bytes. If you specify a value for this option, the output for each file stops after *n* bytes of the files that do not match have been listed. If you set *n* to 0, no mismatches are displayed. If you do not select this option, the display stops after 10 mismatches for each file. Note that there must be space between N and *n*.

+P | -P If you specify +P, Equal shows progress information. Progress information consists of brief messages that tell you what the utility is currently working on: for example, which subdirectory is currently being processed. If you do not include this option or specify -P, progress information is not produced.

+T | -T If you specify -T, Equal does not compare the file types of files. If you do not include this option or specify +T, file types are compared.

pathname1 pathname2 The full pathnames or partial pathnames of the two directories or files that you want to compare. If you name two directories, all files and subdirectories in the two directories are compared.

You can use Equal to determine whether two files are identical or whether the contents of two directories are the same. If you list the pathnames of two directories, a file-by-file comparison is made of the directories. Equal reads the filename, file type, and creation and modification dates of a file in the first directory and then checks to see if a file of the same name exists in the second directory. If it does, Equal compares the files byte-by-byte, leaving the files and going on to the next pair after listing 10 bytes if they are not identical. You can set options to suppress the comparison of file types, to suppress the comparison of file dates and file times, and to specify a different number of bytes to compare before going on to the next file. By default, Equal lists any filenames of files that exist in one file but not in the other. You can suppress that output as well.

By specifying filenames instead of directory names, you can compare two files with different filenames.

EXEC

EXEC

This language command sets the shell default language to EXEC. When you type the name of a file that has the EXEC language type and press Return, the shell executes each line of the file as a shell command. Exec command files are described in the section "Exec Files" later in this chapter.

EXECUTE

EXECUTE *pathname* [*paramlist*]

This internal command executes an Exec file. If this command is executed from the APW Shell command line, the variables defined in the Exec file are treated as if they were defined on the command line.

pathname The full or partial pathname of an Exec file. This filename cannot include wildcard characters.

paramlistt The list of parameters being sent to the Exec file.

You can execute an Exec file by using the EXECUTE command instead of just typing the name of the Exec file and typing Return. The difference between these two methods of executing Exec files is as follows: if you use the EXECUTE command from the shell command line, any variables defined in the Exec file remain valid after control returns to the shell; if, on the other hand, you do not use the EXECUTE command, variables are valid only within the Exec file in which they are defined. See the section "Exec Files" in this chapter for a more complete discussion of this command.

EXIT

`EXIT [`*number*`]`

This internal command terminates execution of an Exec file. See the section "Exec Files" later in this chapter for a more complete discussion of this command.

EXPORT

`EXPORT [`*variable ...*`]`

This internal command makes the specified variables available to Exec files called by the current Exec file. See the section "Exec Files" later in this chapter for a more complete discussion of this command.

FILES

`FILES [`*option ...*`] [`*directory*`]`

The Files utility lists the contents of a directory. You can use this utility to list the full contents of a directory, including the contents of all included subdirectories. You can also search for filenames that include a specified string.

option You can specify as many of the following options as you wish by separating the options with spaces.

`+C` *n* When you specify `+C` followed by a number, Files displays the filenames in *n* columns. If you omit this parameter, one column is used. You cannot specify the `+C` parameter together with the `+L`, `+F`, or `+R` options. Note that there must be a space between the `+C` and the *n*.

`+F` *string* When you specify `+F` followed by a character string, Files lists all files in the specified directory whose filenames include the string. This option searches all included subdirectories regardless of whether the `+R` option is also used. When you specify both the `+F` and `+L` options, the `+L` option is ignored. Note that there must be a space between `+F` and *string*.

`+L` | `-L` When you specify `+L`, Files lists detailed information about each file, similar to the information listed by the `CATLOG` command (see below). If you omit this parameter or specify `-L`, Files lists only filenames.

`+P` | `-P` When you specify `+P`, Files shows the version number and the current date and time. If you omit this parameter or specify `-P`, Files lists the directory contents with no header information.

`+R` | `-R` When you specify `+R`, Files lists all files in the directory, including all files in included subdirectories. If you omit this parameter or specify `-R`, Files lists only the files in the directory specified by *directory*.

directory The full pathname or partial pathname of the directory for which you want a catalog listing.

When you specify the +L option, you get detailed information about the directory, as follows:

Name Type Size (in blocks) Modified (date and time) Created (date and time) Subtype

The subtype corresponds to the Subtype column in the CATALOG command, but does not display mnemonics. The subtype of source files (file type SRC) indicates the APW language type; use the SHOW LANGUAGES command to get a listing of the language numbers of the APW languages installed in your system.

The Files utility generates output with no column headings to facilitate its use as input to future utilities.

FILETYPE

FILETYPE *pathname filetype*

This internal command changes the ProDOS 16 file type of a file.

pathname The full pathname or partial pathname (including the filename) of the file whose file type you wish to change.

filetype The ProDOS 16 file type to which you want to change the file. Use one of the following three formats for *filetype*:

- A decimal number 0–255.
- A hexadecimal number $00–$FF.
- The three-letter abbreviation for the file type used in disk directories: for example, S16, OBJ, EXE. A partial list of ProDOS 16 file types is shown in Table 3.4. See the *Apple IIGS ProDOS 16 Reference* for a complete list of file types.

You can change the file type of any file with the FILETYPE command; APW does not check to make sure that the format of the file is appropriate.

Table 3.4. ProDOS File Types

Decimal	Hex	Abbreviation	File Type
004	$04	TXT	Text
006	$06	BIN	ProDOS 8 binary load
015	$0F	DIR	Directory
176	$B0	SRC	Source
177	$B1	OBJ	Object
178	$B2	LIB	Library
179	$B3	S16	ProDOS 16 load
180	$B4	RTL	Run-time library
181	$B5	EXE	Shell load
182	$B6	STR	Startup load

184	$B8	NDA	New desk accessory
185	$B9	CDA	Classic desk accessory
186	$BA	TOL	Tool set file
249	$F9	P16	ProDOS 16 system file
255	$FF	SYS	ProDOS 8 load

FOR

FOR *variable* [IN *value1 value2* ...]

This internal command, together with the END statement, creates a loop that is executed once for each parameter-value listed. See the section "Exec Files" in this chapter for a more complete discussion of this command.

HELP

HELP [*commandname* ...]

This internal command provides on-line help for all the commands in the command table provided with APW. If you omit *commandname*, the HELP command causes a list of all the commands in the command table to be printed on the screen.

commandname ... The names of the APW Shell commands about which you want information.

When you specify *commandname*, the shell looks for a text file with the specified name in the HELP/ subdirectory in the utility prefix (prefix 6). If it finds such a file, the shell prints the contents of the file on the screen. Help files contain information about the purpose and use of commands. They show the command syntax in the same format as used in this manual.

If you add commands to the command table or change the name of a command, you can add, copy, or rename a file in the HELP/ subdirectory to provide information about the new command.

HISTORY

HISTORY

This internal command lists to the screen the last 20 commands that you have entered on the APW command line. Use the Up Arrow and the Down Arrow keys to scroll through these commands as described in the section "Scrolling Through Commands" in Chapter 2.

IF

IF *expression*

This internal command, together with the ELSE IF, ELSE, and END statements, provides conditional branching in Exec files. See the section "Exec Files" later in this chapter for a more complete discussion of this command.

INIT

INIT [-C] *device* [*name*]

This external command formats a disk as a ProDOS 16 volume.

-C If you specify -C before the pathname, INIT does not prompt you before destroying the current contents of the disk.

Warning: INIT destroys any files on the disk being formatted. Be very careful when using the -C parameter: it is possible to delete the entire contents of a hard disk without warning by using this command.

device The device name (such as .D1) of the disk drive containing the disk to be formatted. If the disk being formatted already has a volume name, you can specify the volume name instead of a device name.

name The new volume name for the disk. The volume name must begin with a slash (/). If you do not specify *name*, the name /BLANK is used.

APW recognizes the device type of the disk drive specified by *device* and uses the appropriate format. INIT works for all disk formats supported by ProDOS 16.

If you do not include the -C parameter and INIT finds a readable directory on the disk you want to format, the following prompt appears on the screen:

```
Destroy /diskname (Y or N)?
```

Here *diskname* is the name of the volume you specified in the INIT command. Type Y and press Return to initialize the disk. Type N and press Return to cancel the command.

INSTALL

INSTALL *volume*
INSTALL /APW *directory*

This external command installs an APW distribution disk.

volume The name of the APW volume that you want to install.

directory The name of the directory into which you want to install APW. This parameter is used only when you are installing the /APW disk.

This command can be used to install any APW disk distributed by Apple. To install the /APW and /APWU disks that came with APW, see the section "Installing APW on a Hard Disk" in Chapter 2. (If you are using APW on floppy disks, these disks require no installation.) To install any other APW disk, such as APWC, see the installation instructions in the manual that came with that disk.

LINK

`LINK [+L|-L] [+S|-S] [+W|-W] file1 [file2] [...] [KEEP=outfile]`

This internal command calls the APW Linker, which links object files to create a load file. You can use this command to link object files created by APW assemblers or compilers and to cause the linker to search library files. If any unresolved references remain after all the specified object files and library files have been specified, the library files in prefix 2 are searched in the order in which they appear in the directory.

The linker is described in detail in Chapter 5.

`+L|-L` If you specify `+L`, the linker generates a listing (called a *link map*) of the segments in the object file, including the starting address, the length in bytes (hexadecimal) of each segment, and the segment type. If you specify `-L`, the link map is not produced.

`+S|-S` If you specify `+S`, the linker produces an alphabetical listing of all global references in the object file (called a *symbol table*). If you specify `-S`, the symbol table is not produced.

`+W|-W` If you select `+W`, the linker stops and waits for a key press when a nonfatal error occurs, to give you the opportunity to read the error message and to decide whether to continue the link. Press Apple-Period (⌘-.) to halt execution, or press any character key or the Space bar to continue. If you omit this option or select `-W`, execution continues without pausing when a nonfatal error occurs. Execution terminates immediately when a fatal error occurs, regardless of the setting of this option.

file1 file2 ... The full pathnames or partial pathnames, minus filename extensions, of all the object files to be included, plus the full or partial pathnames of any library files you want to search. Separate the filenames with spaces. The first file you list, *file1*, must have a `.ROOT` file; for the other object files, either a `.ROOT` file or a `.A` file must be present. For example, the program `TEST` might consist of object files named `TEST1.ROOT`, `TEST1.A`, `TEST1.B`, `TEST2.A`, and `TEST2.B`, all in directory `/APW/MYPROG/`. In this case, you would use `/APW/MYPROG/TEST1 /APW/MYPROG/TEST2` for *objectfile*.

You can also specify one or more library files (ProDOS 16 file type $B2) to be searched. Any library files specified are searched in the order listed. If a library file is listed before an object file, the library file is searched before that object file is linked. Only the segments needed to resolve references that haven't already been resolved are extracted from library files. See the discussion of the `MAKELIB` command in this chapter for more information on library files.

`KEEP=`*outfile* Use this parameter to specify the pathname or partial pathname of the executable load file.

You can specify a default load filename by using the `LinkName` shell variable. Shell variables are described in the section "Variables" later in this chapter. If you do not specify either the `KEEP` parameter or a `LinkName` variable, the link is performed but the load file is not saved.

Important: If you do not include any parameters after the LINK command, you are prompted for an input filename, as APW prompts you for any required parameters. Since the output pathname is not a required parameter, however, you are *not* prompted for it. Consequently, the link is performed, but the load file is not saved unless you have specified a LinkName variable. Note that you can include the KEEP parameter following the pathname you enter in response to the File name prompt.

As an example of the use of the LINK command, suppose you want to link /APW/TEST1, consisting of object files TEST1.ROOT, TEST1.A, and TEST1.B. The following command creates the load file /APW/MYTEST; no link map or symbol table is produced:

```
LINK /APW/TEST1 KEEP=/APW/MYTEST
```

Suppose you want to link TEST1 consisting of object files TEST.1.ROOT, TEST.1.A, and TEST.1.B, search the library file MYLIB, and link TEST.2 consisting of object files TEST.2.A and TEST.2.B. The following command creates the load file MYTEST, printing the link map but suppressing the symbol table. Note that the library file MYLIB is searched before TEST.2 is linked:

```
LINK +L -S TEST.1 MYLIB TEST.2 KEEP=MYTEST
```

To automatically link a program after assembling or compiling it, use one of the following commands instead of the LINK command: ASML, ASMLG, CMPL, CMPLG, RUN.

If you need to take advantage of the advanced link capabilites provided by the APW Linker, create a file of LinkEd commands and process it using the ALINK command (or by appending it to the last source file when you compile or assemble your program). The linker is described in detail in Chapter 5.

Important: The LINK command can be used only to process object files and library files; do *not* try to process a LinkEd file with the LINK command.

LINKED

LINKED

This language command sets the default language type to the APW Linker command language, LINKED. To process a file of LinkEd commands, use one of the following shell commands: ALINK, ASSEMBLE, or COMPILE.

If you do not need to take advantage of the advanced link capabilities provided by LinkEd, do *not* create a LinkEd file, and do not use the ALINK command. Instead, use one of the following commands to link your program: LINK, ASML, ASMLG, CMPL, or CMPLG. The linker is described in detail in Chapter 5.

LOOP

LOOP

Together with the END statement, this internal command defines a loop that repeats continuously until a BREAK or EXIT command is encountered. The loop is also terminated if any command in the loop returns a nonzero error status while the value of the variable Exit is not null (see the section "Variables" later in this chapter for a discussion of Exit). This statement is used primarily in Exec files. See the section "Exec Files" later in this chapter for more information on loops in Exec files.

MACGEN

MACGEN [+C|-C] *infile outfile macrofile1* [*macrofile2* ...]

The MacGen utility creates a custom macro file for an APW Assembler program by searching one or more macro libraries for the macros referenced in the program and placing the referenced macros in a single file.

+C|-C — If you omit this parameter or specify +C, all excess spaces and all comments are removed from the macro file to save space. If you use the GEN ON directive (to include expanded macros in your source-file listing) or the TRACE ON directive (to include conditional execution directives in your source-file listing), then use the -C parameter with the MACGEN command to improve the readability of the listing.

infile — The full pathname or partial pathname (including the filename) of the APW Assembler source file. MacGen scans *infile* for references to macros.

outfile — The full pathname (including the filename) of the macro file to be created by MacGen.

macrofile1 macrofile2 ... — The full pathnames or partial pathnames (including the filenames) of the macro libraries to be searched for the macros referenced in *infile*. At least one macro library must be specified. Wildcard characters can be used in the filenames. If you specify more than one filename, separate the names with one or more spaces.

Since macro-library searches are time-consuming and any given program may use macros from several macro libraries, it is often more efficient to create a custom macro library containing only those macros needed by your program. The MacGen utility generates such a library.

MacGen scans *infile*, including all files referenced with COPY and APPEND directives, and builds a list of the macros referenced by the program. Next, MacGen scans *macrofile1* for macros referenced in *infile*. If there are still unresolved references to macros, MacGen then scans *macrofile2*, and so on. MacGen can handle macros that call other macros. If there are still unresolved references to macros after all the macro files you specified in the command line have been scanned, MacGen lists the missing macros and prompts you for the name of another macro library. Press Return without a filename to terminate the process before all macros have been found. After all macros have been found (or you press Return to end the process), *outfile* is created.

The following example scans the file /APW/MYPROG for macro names, searches the macro libraries /LIB/MACROS and /LIB/MATHMACS for the referenced macros, and creates the macro file /APW/MYMACROS:

```
MACGEN /APW/MYPROG /APW/MYMACROS /LIB/MACROS /LIB/MATHMACS
```

You can specify a previous version of *outfile* as one of the macro libraries to be searched. For example, suppose the program MYPROG already has a custom macro file called MYMACROS, but you want to add one or more macros from the file LIB.MACROS. In this case, you could use the following command:

```
MACGEN MYPROG MYMACROS MYMACROS LIB.MACROS
```

Important: Before you assemble your program, make sure that the source code contains the directive MCOPY *outfile* to cause the assembler to search *outfile* for the macros.

MAKEBIN

MAKEBIN *loadfile* [*binfile*] [ORG=*val*]

The MakeBin utility converts a ProDOS 16 load file (file type $B5 only) to a ProDOS 8 binary load file (file type $06).

loadfile The full or partial pathname of a load file that contains a single static load segment.

binfile The full or partial pathname of the binary file you want to create. If you do not specify *binfile*, *loadfile* is overwritten with the binary file.

ORG=*val* The binary file is given a fixed start location at *val* and all code is relocated for execution starting at the address *val*. You can use a decimal number for *val* or you can specify a hexadecimal number by preceding *val* with a dollar sign ($). If you omit this parameter, *loadfile* is relocated to start at $2000.

The MakeBin utility does no checking to make sure that your program will run under ProDOS 8. The load file must consist of a single static load segment. It can be absolute or relocatable. If you include an ORG directive in the source file, that ORG is respected; if there is a source-file ORG and you specify a conflicting ORG in the MAKEBIN command, however, an error occurs and the the binary file is not created. See the *ProDOS 8 Reference* manual for the requirements for a binary load file.

APW does not launch or run binary load files (file type $06). You can use the BLOAD and BRUN commands in Applesoft BASIC to run these programs. (Applesoft BASIC is the program BASIC.SYSTEM on your Apple IIGS system disk.) See the *BASIC Programming with ProDOS* manual for a description of the BLOAD and BRUN commands.

MAKELIB

MAKELIB [-F] [-D] *libfile* [+*objectfile* ...] [-*objectfile* ...] [^*objectfile* ...]

The MakeLib utility creates or modifies a library file.

-F If you specify -F, a list of the filenames included in *libfile* is produced. If you leave this option out, no filename list is produced.

-D If you specify -D, the dictionary of symbols in the library is listed. Each symbol listed is a global symbol occurring in the library file. If you leave this option out, no dictionary is listed.

libfile The full pathname or partial pathname (including the filename) of the library file to be created, read, or modified.

+objectfilen The full pathname or partial pathname (including the filename) of an object file to be added to the library. You can specify up to eight object files to add. Separate object filenames with spaces.

-objectfilen The filename of a component file to be removed from the library. This parameter is a filename only, not a pathname. You can specify up to eight component files to remove. Separate filenames with spaces.

^objectfilen The full pathname or partial pathname (including the filename) of a component file to be removed from the library and written out as an object file. If you include a prefix in this pathname, the object file is written to that prefix. You can specify up to eight files to be written out as object files. Separate filenames with spaces.

Note: You must specify at least one object file and no more than eight object files. If you do not specify at least one object file, the message `No action requested` appears on the screen.

An APW library file (ProDOS 16 file type $B2) consists of one or more component files, each containing one or more segments. Each library file contains a library dictionary segment that the linker uses to find the segments it needs.

As illustrated in Figure 3.8, MakeLib creates a library file from any number of object files. In addition to indicating where in the library file each segment is located, the library dictionary segment indicates which object file each segment came from. The MakeLib utility can use that information to remove any component files you specify from a library file. MakeLib can even recreate the original object file by extracting the segments that made up that file and writing them out as an object file. Use the `-F` and `-D` parameters to list the contents of an existing library file.

Note: The `MAKELIB` command is for use only with APW object-module-format (OMF) library files used by the linker. For information on the creation and use of libraries used by language compilers, consult the manuals that came with those compilers.

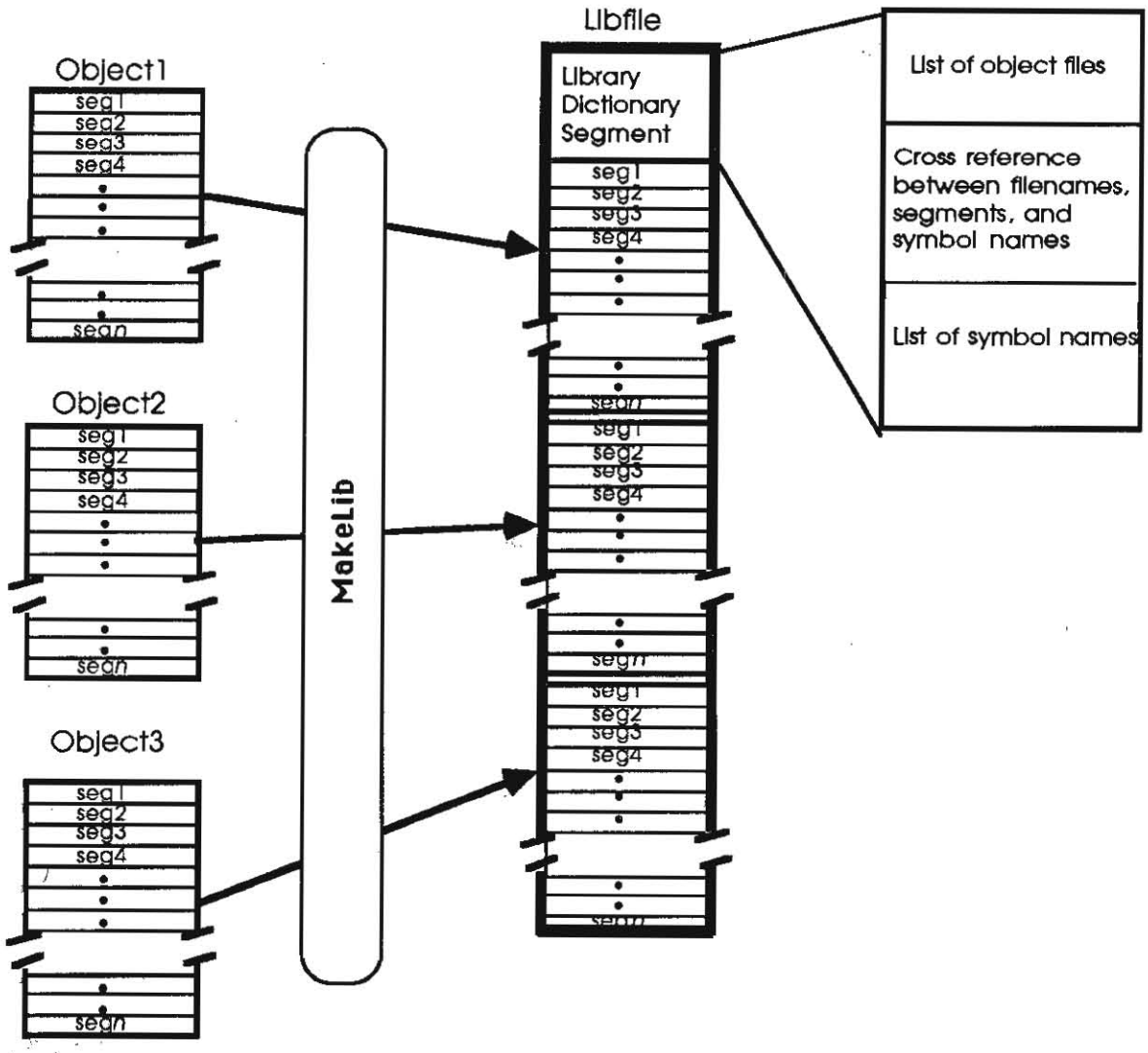


Figure 3.8. Creation of a Library File

To create an OMF library file using the APW Assembler, use the following procedure:

1. Write one or more source files in which each library subroutine is a separate segment. You might want to make the first segment of each file a dummy, to be discarded later as explained in the next two steps.
2. Assemble the programs, specifying a unique name for each program with the `KEEP` parameter in the `ASSEMBLE` command. Each multisegment program is saved as two object files: one with the extension `.ROOT` and one with the extension `.A`. The `.ROOT` file contains the first segment and the `.A` file contains all the rest. If you made the first segment a dummy, then the `.ROOT` file contains only the dummy segment.
3. Run the MakeLib utility, specifying each object file to be included in the library file. For example, if you assembled two files, creating the object files `LIBOBJ1.ROOT`, `LIBOBJ1.A`, `LIBOBJ2.ROOT`, `LIBOBJ2.A`, and your library file is named `LIBFILE`, then your command line should be as follows:

```
MAKELIB LIBFILE +LIBOBJ1.ROOT +LIBOBJ1.A +LIBOBJ2.ROOT +LIBOBJ2.A
```

If you made the first segment of each file a dummy, however, then you do not need to include the .ROOT files, and your command line should be as follows:

```
MAKELIB LIBFILE +LIBOBJ1.A +LIBOBJ2.A
```

4. Place the new library file in the LIBRARIES/ subdirectory. (You can accomplish this in step 3 by specifying /APW/LIBRARIES/LIBFILE for the library file, or you can use the MOVE command after the file is created.)

APW OMF library files and library-dictionary segments are described in the section "Object Module Format" in Chapter 7. The APW Linker is described in Chapter 5.

MOVE

MOVE [-C] *pathname1* [*pathname2*]
MOVE [-C] *pathname* [*directory*]

This internal command moves a file from one directory to another; it can also be used to rename a file.

-C If you specify -C before the first filename, then MOVE does not prompt you if the target filename (*pathname2*) already exists.

pathname1 The full pathname or partial pathname (including the filename) of the file to be moved. Wildcard characters may be used in this filename.

pathname2 The full pathname or partial pathname of the file you wish to move the file to. If you specify a target filename, the file is renamed when it is moved. Wildcard characters *cannot* be used in this pathname. If the prefix of *pathname2* is the same as that of *pathname1*, then the file is renamed only.

pathname The full pathname or partial pathname (including the filename) of the file to be moved. Wildcard characters may be used in this filename.

directory The pathname or partial pathname of the directory you wish to move the file to. If you do not include a filename in the target pathname, the file is not renamed. Wildcard characters *cannot* be used in this pathname.

If the file you wish to move and the target directory are on the same volume, APW calls ProDOS 16 to move the directory entry (and rename the file, if a target filename is specified). If the source and destination are on different volumes, the file is copied; if the copy is successful, the original file is deleted. If the file specified in *pathname2* already exists and you complete the move operation, then the old file named *pathname2* is deleted and replaced by the file that was moved.

MU

MU

This command is an alias for PREFIX 6 /APWU/UTILITIES. You can use this command when you are running APW from floppy disks to switch the utility prefix (prefix 6) to the utility subdirectory on the /APWU disk, which contains a full set of utility

programs and help files. Use the UMU command to switch the utility prefix back to the /APW disk.

> **Note:** The MU and UMU commands are created by ALIAS commands in the LOGIN file of the floppy-disk version of APW. These aliases are not included in the APW command table and are not set by the hard-disk version of the LOGIN file (that is, the LOGIN file put on a hard disk by the INSTALL command).

PREFIX

PREFIX [*n*] *directory*

This internal command sets any of the eight standard ProDOS 16 prefixes to a new subdirectory.

n A number from 0 through 7, indicating the prefix to be changed. If this parameter is omitted, 0 is used. This number must be preceded by one or more spaces.

directory The pathname or partial pathname of the subdirectory to be assigned to prefix *n*.

Prefix 0 is the current prefix; all shell commands that accept a pathname use prefix 0 as the default prefix if you do not include a slash (/) at the beginning of the pathname. Prefixes 1 through 6 are used for specific purposes by ProDOS 16 and APW; see the section "Standard Prefixes" earlier in this chapter for details. The default settings for the prefixes are shown in Table 3.1. Use the SHOW PREFIX command to find out what the prefixes are currently set to.

The prefix assignments are reset to the defaults each time APW is booted. To use a custom set of prefix assignments every time you start APW, put the PREFIX commands in the LOGIN file. (The LOGIN file is an Exec file that is executed automatically at load time if it is present. See the section "Exec Files" later in this chapter for instructions on writing an Exec file. See the section "Installing APW on a Hard Disk" in Chapter 2 for an example of a LOGIN file that uses PREFIX commands.)

PRODOS

PRODOS

This language command sets the APW Shell default language to ProDOS 16 text. ProDOS 16 text files are standard-ASCII files with ProDOS 16 file type $04; these files are recognized by ProDOS 16 as text files. APW text files, on the other hand, are standard-ASCII files with ProDOS 16 file type $B0 and an APW language type of TEXT. The APW language type is not used by ProDOS 16. See the *Apple IIGS ProDOS 16 Reference* for a discussion of ProDOS 16 file types.

QUIT

QUIT

This internal command terminates the APW program and returns control to ProDOS 16. If you called APW from another program, ProDOS 16 returns you to that program; if not, ProDOS prompts you for the next program to load.

RENAME

RENAME *pathname1 pathname2*

This internal command changes the name of a file. You can also use this command to move a file from one subdirectory to another on the same volume.

pathname1 The full pathname or partial pathname (including the filename) of the file to be renamed or moved. If you use wildcard characters in the filename, the first filename matched is used.

pathname2 The full pathname or partial pathname (including the filename) to which *pathname1* is to be changed or moved. You cannot use wildcard characters in the filename.

If you specify a different subdirectory for *pathname2* than for *pathname1*, the file is moved to the new directory and given the filename specified in *pathname2*.

Important: The subdirectories specified in *pathname1* and *pathname2* must be on the same volume. To rename a file and move it to another volume, use the MOVE command.

RUN

RUN [*option ...*] *file1* [*file2*] [...] [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2*] [...])] [*language1*=(*option ...*)
[*language2*=(*option ...*)] [...]]

This internal command compiles (or assembles) one or more source files, links one or more object and library files, and runs the resulting load file. See the ASML command for a list of options and a description of the parameters. See your compiler or assembler manual for the default values of the parameters and the language-specific options available.

The RUN and CMPLG commands are aliases for ASMLG.

SEARCH

SEARCH [+C|-C] [+L|-L] [+P|-P] *string pathname*

The Search utility searches a file or files for the string you specify.

`+C | -C` If you specify `+C`, a match is found only if the string found matches the search string exactly, including case. If you omit this option or specify `-C`, searches are not case sensitive.

`+L | -L` If you specify `+L`, search lists the line number and the contents of the line in which it found a match for the search string. If you omit this option or specify `-L`, only the name of the file in which a match was found is listed.

`+P | -P` If you specify `+P`, search displays progress information. Progress information consists of brief messages that tell you what the utility is currently working on; for example, which file is currently being searched. If you omit this option or specify `-P`, no progress information is displayed.

string The string for which you wish to search. To specify a string that includes spaces, enclose the string in double quotation marks (").

pathname The full pathname or partial pathname, including filename, of the file you want to search for *string*. You can include wildcard characters in this filename.

You can use this utility to search a text or source file for all occurrences of a string, or, by using wildcards in the filename, to search through several files to find out in which one the string occurs.

For example, to search the file `/APW/MYFILES/DONUT` for all occurences of the word `HOLE`, you could use the following command:

```
SEARCH -L HOLE /APW/MYFILES/DONUT
```

To search all files in the directory `MYFILES` to determine which files contain the string `Donut Hole`, making the search case-sensitive, you could use the following command:

```
SEARCH -C 'Donut Hole' /APW/MYFILES/=
```

SET

`SET [`*variable* `[`*value*`]]`

This internal command allows you to assign a value to a variable name. You can also use this command to obtain the value of a variable or a list of all defined variables.

variable The variable name you wish to assign a value to. Variable names are not case-sensitive, and only the first 255 characters are significant. If you omit *variable*, a list of all defined names and their values is written to standard output.

value The string that you wish to assign to *variable*. Values are case sensitive and are limited to 255 characters. All characters, including spaces, starting with the first nonspace character after *variable* to the end of the line, are included in *value*. If you include *variable* but omit *value*, the current value of *variable* is written to standard output.

The SET command can be used on a shell command line or in an Exec file. Use the UNSET command to delete the definition of a variable. Variables are valid only within the Exec file in which they are defined unless you use the EXPORT or EXECUTE commands. See the section "Exec Files" later in this chapter for a more complete discussion of the SET, EXPORT, and EXECUTE commands.

Important: Certain variable names are reserved. See the section "Variables" later in this chapter for a list of reserved variable names.

SHOW

```
SHOW [LANGUAGE] [LANGUAGES] [PREFIX] [TIME] [UNITS]
```

This internal command provides information about the system.

LANGUAGE Shows the current system-default language.

LANGUAGES Shows a list of all languages defined in the command table, including their language numbers.

PREFIX Shows the current subdirectories to which the ProDOS 16 prefixes are set. See the section "Standard Prefixes" later in this chapter for a discussion of APW prefixes.

TIME Shows the current time and date.

UNITS Shows the available units, including device names and volume names. Only those devices that have formatted ProDOS volumes in them are shown. To see the device names for all of your disk drives, make sure that each drive contains a ProDOS disk.

More than one parameter can be entered on the command line; to do so, separate the parameters by one or more spaces. If you enter no parameters, you are prompted for them.

TEXT

```
TEXT
```

This language command sets the APW Shell default language to APW TEXT. APW text files are standard-ASCII files with ProDOS 16 file type $B0 and an APW language type of TEXT. The TEXT language type is provided to support any text formatting programs that may be added to APW. TEXT files are shown in a directory listing as SRC files with a subtype of TEXT.

Use the PRODOS command to set the language type to ProDOS 16 text: that is, standard-ASCII files with ProDOS 16 file type $04. ProDOS 16 text files are shown in a directory listing as TXT files with no subtype. See the *Apple IIGS ProDOS 16 Reference* manual for a discussion of ProDOS 16 file types.

TYPE

TYPE [+N|-N] *pathname1* [*startline1* [*endline1*]]
[*pathname2* [*startline2* [*endline2*]] [...]]

This internal command prints one or more text or source files to standard output (usually the screen).

+N|-N If you specify +N, the shell precedes each line with a line number. The default is -N: no line numbers are printed. If you type more than one file, the line numbers are *not* reset at the start of each file.

pathname1 ... *pathname2* ... The full pathnames or partial pathnames (including the filenames) of the files to be printed. You can use wildcard characters in these filenames, in which case every text or source file matching the wildcard filename specification is printed. If you specify more than one pathname in the command, separate the pathnames with spaces.

startline1 The line number of the first line of *pathname1* to be printed. If this parameter is omitted, the entire file is printed.

endline1 The line number of the last line of *pathname1* to be printed. If this parameter is omitted, the file is printed from *startline* to the end of the file.

ProDOS 16 text files and APW source files (including APW TEXT files) can be printed with the TYPE command. For example, to write lines 2 through 5 of sourcefile MYPROG and 9 through 18 of text file RELEASE.NOTES, use the following command:

```
TYPE MYPROG 2 5 RELEASE.NOTES 9 18
```

To redirect output to a printer or file, use output redirection as described in the section "Redirecting Input and Output" later in this chapter. For example, to send to the printer the entire file MYFILE and the file YOURFILE from line 9 to the end of the file, preceding each line with a line number, use the following command:

```
TYPE +N MYFILE YOURFILE 9 >.PRINTER
```

UMU

UMU

This command is an alias for PREFIX 6 4/../UTILITIES. You can use this command when you are running APW from floppy disks to switch the utility prefix (prefix 6) from the utility subdirectory on the /APWU disk, which contains a full set of utility programs and help files, to the utility subdirectory on the /APW disk. Use the MU command to switch the utility prefix to the /APWU disk.

Note: The MU and UMU commands are created by ALIAS commands in the LOGIN file of the floppy-disk version of APW. These aliases are not included in the APW command table and are not set by the hard-disk version of the LOGIN file (that is, the LOGIN file put on a hard disk by the INSTALL command).

UNALIAS

UNALIAS *alias1* [*alias2* ...]

This internal command deletes aliases for commands.

alias1 alias2 ... The names of the aliases you wish to delete.

Use the ALIAS command to define an alias.

UNSET

UNSET *variable1* [*variable2* ...]

This internal command deletes the definition of a variable.

variable1 variable2 ... The names of the variables you wish to delete. Variable names are not case sensitive, and only the first 255 characters are significant.

Use the SET command to define a variable. See the next section, "Exec Files," for a more complete discussion of the SET command.

VERSION

VERSION

This external command displays the version number of the APW Shell program you are using.

Exec Files

You can execute one or more APW Shell commands from a command file called an *Exec file*. To create a command file, first set the system language to EXEC by typing EXEC and pressing Return, and then open a new file with the editor. Any of the commands described in this chapter can be included in an Exec file. The commands are executed in sequence, as if you had typed them from the keyboard.

To execute an Exec file, type the full pathname or partial pathname (including the filename) of the Exec file and press Return. You can also execute an Exec file using the EXECUTE command. The advantages of doing so are described in the section on the EXECUTE command later in this chapter.

You can place an Exec file in the UTILITIES/ subdirectory (prefix 6) and add it to the command table as a utility program. Then you can execute the program just by typing its name on the shell's command line (or by typing EXECUTE and the filename); in this case, the full pathname of the Exec file is not needed. The command table is discussed in the section "Command Types and the Command Table" earlier in this chapter.

When an Exec file terminates, it returns control to the Exec file that called it, or to the shell if it was executed from a shell command line. If you execute an interactive utility, such as

the APW Editor, from an Exec file, the utility operates normally, accepting input from the keyboard. If the utility name was not the last command in the Exec file, you are returned to the Exec file when you quit the utility.

Exec files are programmable: that is, APW includes several commands designed to be used within Exec files that permit conditional execution and branching. You can also pass parameters into Exec files by including them on the command line. These features are described in the following sections.

Exec files can call other Exec files. The level to which Exec files can be nested and the number of variables that can be defined at each level depend on the available memory.

You can put more than one command on a single line of an Exec file by separating the commands with semicolons (`;`).

The commands described in this section are usually used in Exec files; note, however, that any of these commands can also be used from a shell command line. For example, the following command line would delete from a directory all files that ended in the extensions `.OLD`, `.BAK`, and `.TEST`:

```
FOR EXT IN OLD BAK TEST ; DELETE =.{EXT} ; END
```

`FOR`–`END` loops are described later in this section.

The following subsections explain how to write Exec files. You are told how to pass parameters into Exec files, how to use variables in Exec files, and how to use each of the shell commands that provide conditional execution, branching, and other functions useful in Exec files.

Passing Parameters Into Exec Files

When you execute an Exec file, you can include the values of as many parameters as you wish by listing them after the pathname of the Exec file on the command line. Separate the parameters with spaces or tab characters. To specify a parameter value that has embedded spaces or tabs, enclose the value in quotation marks (`"`). Quotation marks embedded in a parameter string must be doubled.

For example, suppose you want to execute an Exec file named `FARM`, and you want to pass the following parameters to the file:

- `cow`
- `chicken`
- `one egg`
- `"Old" MacDonald`

In this case, you would enter the following command on the command line:

```
FARM cow chicken "one egg" """Old"" MacDonald"
```

Parameters are assigned to variables inside the Exec file as described in the next section.

Programming Exec Files

In addition to being able to execute any of the shell commands discussed in the "Command Descriptions" section of this chapter, Exec files can use several special commands that permit conditional execution and branching. This section discusses the use of variables in Exec files and the logic operators used to form Boolean (logical) expressions.

Variables

Any alphanumeric string up to 255 characters long can be used as a variable name in an Exec file. (If you use more than 255 characters, only the first 255 are significant.) All variable values and parameters are ASCII strings of 255 or fewer characters. Variable names are not case-sensitive, but the values assigned to the variables *are* case-sensitive.

To define values for variables, you can either pass them into the Exec file as parameters or include them in a `FOR` command or a `SET` command as described in the sections on those commands later in this chapter. To assign a null value to a variable (a string of zero length), use the `UNSET` command.

Curly brackets ({ }) around a variable name indicate the *value* of the variable. For example, if you execute the command `SET ECHO ON`, then `{Echo}` refers to the value `ON`.

Variables included in an `EXPORT` command on a shell command line can be used within any Exec file. Variables included in an `EXPORT` command within an Exec file are valid in any Exec files called by that file; they can be redefined locally, however. Variables redefined within an Exec file revert to their original values when that Exec file is terminated unless the file was called with an `EXECUTE` command.

The following variable names are reserved. Several of these variables may have number values; keep in mind that these values are literal ASCII strings. A null value (a string of zero length) is considered undefined. Use the `UNSET` command to set a variable to a null value. Several of the predefined variables are used to set up a printer. See the section "Using a Printer" in Chapter 2 for a discussion of printer initialization.

Variable Name	Value
`0`	The name of the Exec file being executed.
`1, 2, ...`	Parameters from the command line. Parameters are numbered sequentially in the sequence in which they are entered.
`#`	The number of parameters passed.
`CaseSensitive`	If you set this variable to any non-null value, string comparisons are case-sensitive. The default value is null.
`Command`	The name of the last command executed, exactly as entered, excluding any command parameters. For example, if the command was `/APW/MYPROG`, then `Command` equals `/APW/MYPROG`, while if the command was `EXECUTE /APW/MYEXEC`, then `Command` equals `EXECUTE`. The `Parameters` variable is set to the value of any parameters.

`Echo` If you set this variable to a non-null value, then commands within the Exec file are printed to the screen before being executed. The default value for `Echo` is null (undefined); use the `UNSET` command to set `Echo` to a null value (that is, to delete its definition).

`Exit` If you set this variable to any non-null value, and if any command or nested Exec file returns a nonzero error status, then execution of the Exec file is terminated. The default value for `Exit` is non-null (it is the ASCII string `true`). Use the `UNSET` command to set `Exit` to a null value (that is, to delete its definition).

`KeepType` A hexadecimal number (represented as an ASCII string) corresponding to a load file type. If `KeepType` is undefined or set to a nonvalid file type, $B5 (shell load file) is used. The most common alternative is $B3 (system load file). Valid load file types are $B3 through $BE.

`KeepName` If you set this variable to any non-null value and do not include a `KEEP` parameter on the shell command line when you compile or compile and link a program, then the shell uses this variable to name the output files. If you set `KeepName` to a non-null value, it overrides any `KEEP` directive in the source file. The default value for `KeepName` is null (undefined); use the `UNSET` command to set `KeepName` to a null value (that is, to delete its definition).

The `KeepName` variable can include the wildcard characters `%` and `$`. The percent sign (`%`) is replaced with the source filename. The dollar sign (`$`) is replaced with the source filename with the last extension removed. For example, if `{KeepName}` is set to `%.O` and you execute the command `CMPL MYFILE`, the shell uses the name `MYFILE.O.ROOT` for the object file and the linker uses the name `MYFILE.O` for the load file. Similarly, if `KeepName` is set to `$` and you execute the command `CMPL MYFILE.C`, the shell uses the name `MYFILE.ROOT` for the object file and the name `MYFILE` for the load file.

The `KeepName` variable is not used by the `LINK` command. See the description of the `LinkName` variable, below, for a way to set default load filenames.

> **Important:** Because ProDOS does not allow filenames longer than 15 characters, you must be careful not to use a source filename that will create an output filename longer than 15 characters. For example, if `KeepName` is set to `%.OUT` and the source filename is `LONGNAME`, the compile will fail when the shell tries to open the file `LONGNAME.OUT.ROOT`, which has 17 characters.
>
> Because the shell will not let you overwrite a source file with a load file, you cannot set `KeepName` to `%` and use it with a link. For example, if `KeepName` is set to `%` and you try to execute the command `CMPL MYFILE`, the link will

fail when the linker tries to write a load file named MYFILE.

`LinkName` If you set this variable to any non-null value and you do not include a KEEP parameter on the shell command line, then the shell uses this variable to name the load file. If you set LinkName to a non-null value, it overrides any KEEP command in the LinkEd source file. The default value for LinkName is null (undefined); use the UNSET command to set LinkName to a null value (that is, to delete its definition).

The LinkName variable can include the wildcard characters % and $. The percent sign (%) is replaced with the object file's root filename. The dollar sign ($) is replaced with the object file's root filename with the last extension removed. For example, if LinkName is set to %.O and you execute the command LINK MYFILE, the shell uses the name MYFILE.O for the load file. Similarly, if LinkName is set to $ and you execute the command CMPL MYFILE.C, the shell uses the name MYFILE for the load file.

If you name more than one object file on the command line and the LinkName variable includes a wildcard character, the shell applies the LinkName variable to the root name of the first object file linked.

Important: Because ProDOS does not allow filenames longer than 15 characters, you must be careful not to use a value for LinkName that will create a load filename longer than 15 characters. For example, if LinkName is set to %.LOADFILE and the root filename is LONGNAME.ROOT, the compile will fail when the shell tries to open the file LONGNAME.LOADFILE, which has 17 characters.

Because the shell will not let you overwrite a source file with a load file, you cannot set {LinkName} to % when the object file's root filename is the same as the source filename. For example, if {LinkName} is set to % and you try to execute the command CMPL MYFILE, the link will fail when the linker tries to write a load file named MYFILE.

`Parameters` The parameters of the last Exec file executed, exactly as entered, excluding the Exec file's pathname. For example, if you execute an Exec file with the command /APW/FARM COW DUCK, then {Parameters} equals COW DUCK.

`PrinterColumns` An ASCII number indicating the number of characters on a line. The printer driver assumes a new line has begun each time {PrinterColumns}+1 characters have been printed since the last carriage return. The printer driver uses this parameter to count lines on a page in case your printer automatically inserts a carriage return and line feed to wrap lines that are too long. If your printer stops printing at the end of the line, or returns to the start of the

line and overprints the line, then set `{PrinterColumns}` to `0` and the printer driver will count a new line only when a carriage return is sent.

`PrinterInit` The initialization string to be sent to your printer each time you send text to the printer. Use this string to set the printer options you want to use, such as character pitch, print quality, line spacing, or boldfacing. Precede a character with a tilde (~) to indicate a control character. Precede a character with a number sign (#) to indicate that the next character should have the most significant bit set. Precede the tilde with a number sign to indicate a control character with the most significant bit set.

To specify the number-sign character ($23), use the sequence ~#. To specify the tilde character ($7E), use the sequence ~~. To specify the tilde character with the most significant bit set ($FE), use the sequence #~~. A space is interpreted as a space character ($20).

> **Important**: The shell does no error checking on the initialization string; if you specify an illegal control character, the shell subtracts $40 from the character and sends it to the printer anyway. For example, if you specify ~g, the shell sends $27 to the printer.

The following command sends the string "Control-L Esc a 2" to the printer (for an Apple ImageWriter II printer, this string feeds the paper to the next top-of-form position and sets the printer to near-letter-quality mode):

```
SET PRINTERINIT ~L~[a2
```

The following command sends the sequence $1B $44 $80 $00 to the printer (for an Apple ImageWriter II printer, this sequence adds an automatic line feed after every carriage return):

```
SET PRINTERINIT ~[D#~@~@
```

See the manual that came with your printer for the options available and the codes necessary to set them.

`PrinterLineFeed` If this variable is not defined, no line-feed character ($0A) is inserted after a carriage return ($0D). If this variable is non-null, the printer driver automatically inserts a line feed after every carriage return. If no line feed is added when one is needed, the printer overprints every line of text without advancing the paper. If a line feed is added when one is not needed, the lines are double spaced.

`PrinterLines` An ASCII number indicating the number of lines to be sent to the printer before a form-feed character ($0C) is sent. If `{PrinterLines}` = `0`, no form-feed characters are sent.

`PrinterSlot` An ASCII number from `1` through `7` indicating the number of the slot containing your printer-driver PC board. The default value for `PrinterSlot` is `1`.

Important: If you specify the wrong slot number, the printer initialization string and output data are sent to the wrong slot, with consequences that depend on the device assigned to that slot. For example, the system might hang or reset.

`Status` The error status returned by the last command or Exec file executed. This variable is the ASCII character `0` ($30) if the command was executed successfully. For most commands, if an error occurred, the error value returned by the command is the ASCII string `65535` (representing the error code $FFFF).

Logic Operators

APW includes two operators that you can use to form Boolean (logical) expressions. String comparisons are case sensitive if `{CaseSensitive}` is not null (the default is for string comparisons to *not* be case sensitive). If an expression's result is true, the expression returns the character `1`. If an expression's result is not true, the expression returns the character `0`. There must be one or more spaces before and after the comparison operator.

The two Exec file logic operators are defined as follows:

str1 == *str2* String comparison: true if string *str1* and string *str2* are identical; false if not.

str1 != *str2* String comparison: false if string *str1* and string *str2* are identical; true if not.

Operations can be grouped with parentheses. For example, the following expression is true if one of the expressions in parentheses is false and one is true; the expression is false if both expressions in parentheses are true or if both are false:

```
IF ( COWS == KINE ) != ( CATS == DOGS )
```

Important: Every symbol or string in a logical expression must be separated from every other by at least one space. In the preceding expression, for example, there is a space beween the string comparison operator `!=` and the left parenthesis, and another space between the left parenthesis and the string `CATS`.

Entering Comments

To enter a comment into an Exec file, use the `COMMENT` command (or its alias, an asterisk (`*`)), followed by a space and the comment. See the section on the `COMMENT` command, later in this chapter, for details.

LOGIN Files

Each time you start APW, it looks for an Exec file named `LOGIN` in the APW system prefix (prefix 4). If it finds such a file, APW executes it before doing anything else. You can use `LOGIN` to set system variables such as `PrinterSlot`, to change default prefix

assignments, or even to execute a utility program. Any APW command described in this chapter can be used in a LOGIN file.

Any system variables set in a LOGIN file must be included in an EXPORT command to be exported to the shell command level and to other Exec files. To reexecute LOGIN without reloading APW (to reset system parameters to your selected defaults, for example), use the command EXECUTE 4/LOGIN.

Exec File Command Descriptions

The commands described in this section can be used in Exec files to control conditional execution and branching and to assign values to variables.

The following notation is used to describe commands:

UPPERCASE	Uppercase letters indicate a command name or an option that must be spelled exactly as shown. The APW Shell command interpreter is not case sensitive; that is, you can enter commands in any combination of uppercase and lowercase letters.
italics	Italics indicate a variable that you must replace with specific information, such as a pathname or file type.
[ ]	Parameters enclosed in square brackets are optional.
. . .	Ellipses indicate that a parameter or sequence of parameters can be repeated as many times as you wish.
. . .	Vertical ellipses indicate that any number of shell commands can be inserted between the two commands shown.

BREAK

BREAK

This internal command terminates the innermost FOR or LOOP statement currently executing. For example, if a FOR loop is executing inside an IF statement and a BREAK statement is encountered, control passes to the statement following the FOR loop's END statement. A BREAK statement can be used to terminate a LOOP loop.

COMMENT

COMMENT [*text*]

This internal command, or an asterisk (*), is used to enter comments into Exec files.

text The comment that you want to include in the file. All characters starting with the first nonspace character after the COMMENT or * command to the end of the line are part of the comment. To include a semicolon (;), vertical bar (|), greater-than sign (>), or less-than sign (<) in the comment, enclose *text* in double quotation marks ("). You *must* include a space after the asterisk or COMMENT, or the shell interprets the line as a pathname.

The asterisk and the word COMMENT are included in the command table as null commands. They are treated by the shell as commands that do nothing. Consequently, a semicolon terminates the comment; the text following the semicolon is interpreted as another command. If you include a redirection operator (> or <) or pipeline operator (|), the shell attempts to redirect the comment as it would any command.

As an example of the use of this command, the following Exec file sends a catalog listing to the printer:

```
CATALOG >.PRINTER
* Send a catalog listing to the printer
```

The following line uses a semicolon to place the comment on the same line as the CATALOG command:

```
CATALOG >.PRINTER  ;* Send a catalog listing to the printer
```

CONTINUE

CONTINUE

This internal command causes control to skip over following statements to the next END statement that isn't the END for an IF statement. It does not cause termination of the loop (unless the last value has been used in a FOR loop).

ECHO

ECHO *string*

This internal command lets you write messages to the screen.

string The string that you wish to print to the screen. All characters starting with the first nonspace character after the ECHO command to the end of the line are printed to the screen. If you include variables in the string, they are expanded—that is, their current value is substituted—before they are printed to the screen. To include leading space characters, enclose *string* in double quotation marks (").

EXECUTE

EXECUTE *pathname* [*paramlist*]

This internal command executes an Exec file, treating the commands in the file as if they were in the Exec file that contains the EXECUTE command. If this command is executed from the APW Shell command line, any variables defined in the Exec file are treated as if they were defined on the shell command line.

pathname The full or partial pathname of an Exec file. This filename cannot include wildcards.

paramlist The list of parameters being sent to the Exec file. Separate the parameters by one or more spaces.

Exec files can be *nested*: that is, one Exec file can include a statement that executes another Exec file, that file can in turn call a third, and so on until your Apple IIGS runs out of memory. Normally, variables defined within each Exec file are local to that file; that is, the values are not valid in any other Exec file (see the discussion of the EXPORT command later in this chapter for an exception to this rule). If you use an EXECUTE command, however, any commands executed in the Exec file *called* by the EXECUTE command (the nested file) are treated as if they were executed in the file that *contains* the EXECUTE command (the calling file). Consequently, any variables defined in the calling Exec file are valid in the nested Exec file, and any variables defined in the nested Exec file remain valid after the nested Exec file finishes executing.

As illustrated in Figure 3.9, when you execute an Exec file with the EXECUTE command, it's as if the commands in the nested Exec file are inserted into the calling Exec file. The Exec files illustrated in part B of this figure are exactly equivalent to those illustrated in part A. Note that EXEC4 is not called with an EXECUTE command, and so it does not share variable definitions with EXEC2 or EXEC1.

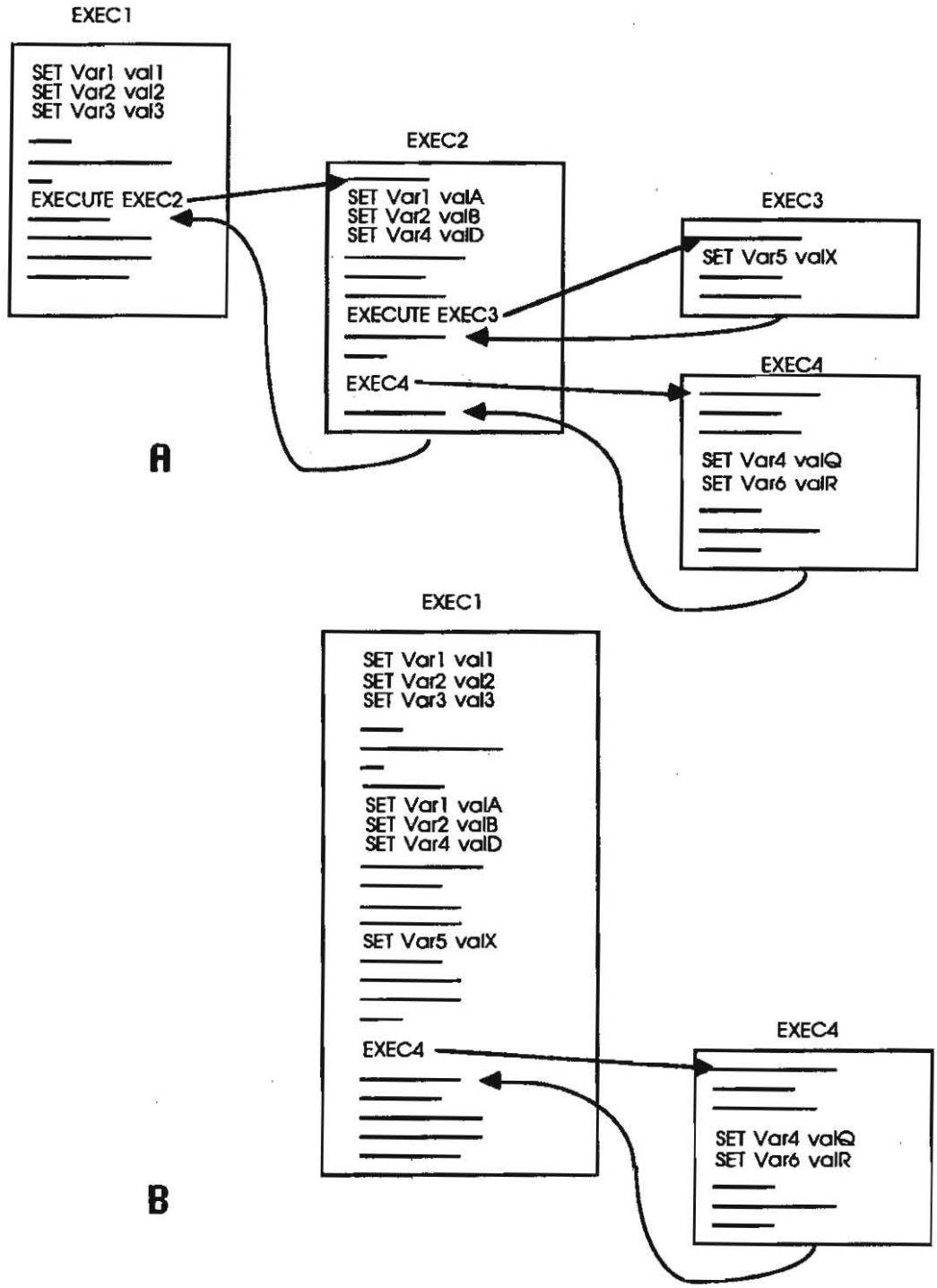


Figure 3.9. Effect of the EXECUTE Command

The definitions of variables resulting from the SET commands and Exec-file calling sequence shown in part A of Figure 3.9 are illustrated in Figure 3.10.

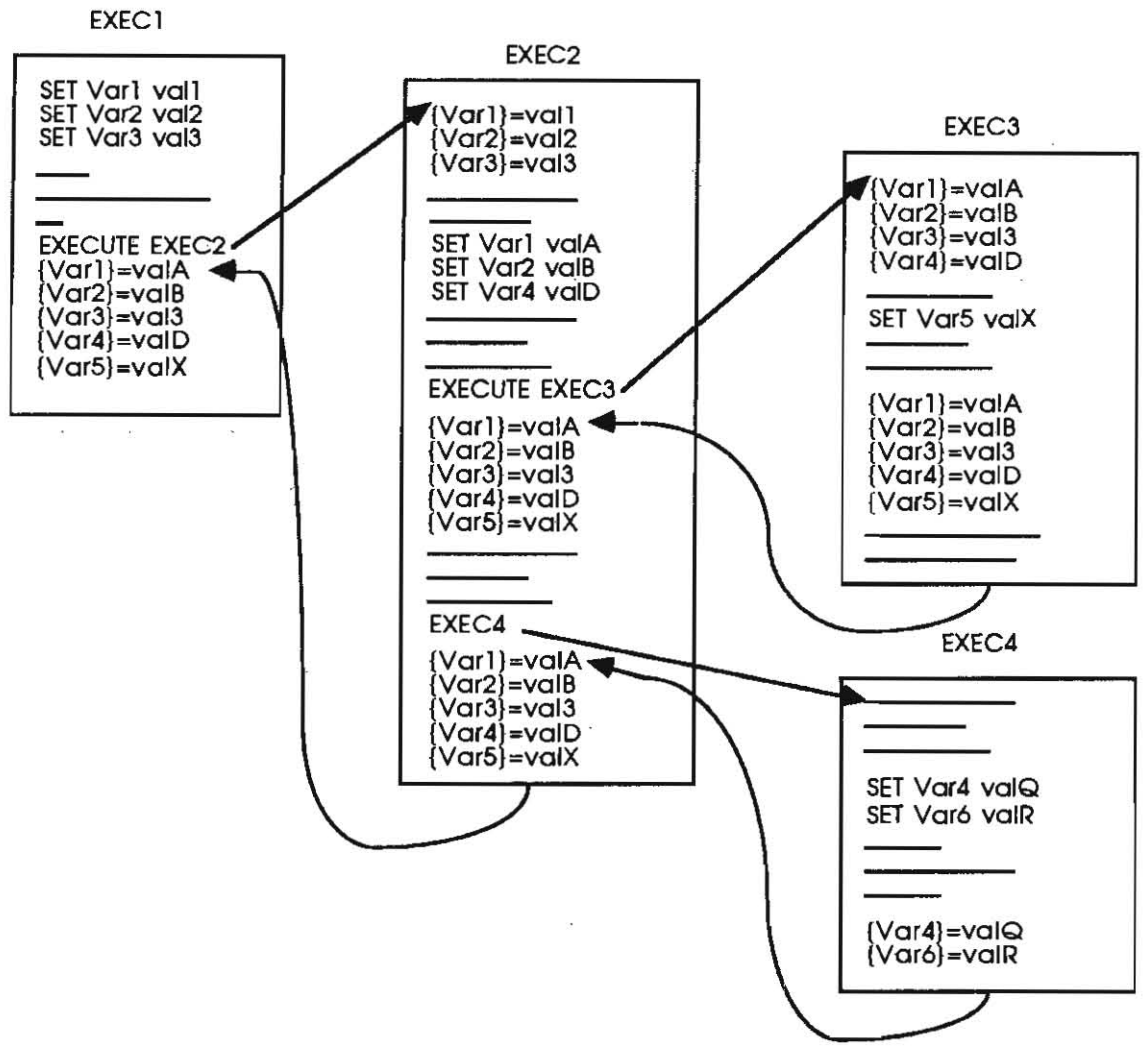


Figure 3.10. Variable Definitions and EXECUTE commands

Similarly, if you use an `EXECUTE` command on the shell command line to execute an Exec file, the variables defined in that Exec file are treated as if they were typed on the command line. For example, suppose you write an Exec file called `SETUP` that contains the following lines:

```
SET ECHO ON
SET PRINTERSLOT {1}
```

You can excute this Exec file from the command line with the following command:

```
SETUP 2
```

In this case, the variable `Echo` is set to `ON` and `PrinterSlot` is set to `2` (the value passed as a command-line parameter) only while the Exec file is executing. When the Exec file finishes, `Echo` and `PrinterSlot` return to their default values. To make the values

of `Echo` and `PrinterSlot` remain valid after the file `SETUP` has finished excuting, use the following command:

```
EXECUTE SETUP 2
```

In this case, the shell acts as if the commands `SET ECHO ON` and `SET PRINTERSLOT 2` were typed on the command line: that is, `{Echo}` is still set to `ON` and `{PrinterSlot}` is still set to 2 after the Exec file returns control to the shell.

> **Note:** When the APW Shell finds an Exec file named `LOGIN` in the APW system prefix (prefix 4) during system load, the shell automatically executes `LOGIN` immediately after loading APW. Use the `EXPORT` command in the `LOGIN` file to make variable definitions valid at the shell command level and in other Exec files. To reexecute `LOGIN` without reloading APW—to reset system parameters to your selected defaults, for example—use the command `EXECUTE 4/LOGIN` from a shell command line.

EXIT

`EXIT` [*number*]

This internal command terminates execution of the Exec file.

number This parameter is the error status with which the Exec file terminates. If you specify a value for *number* and the Exec file was executed from another Exec file, the predefined variable `Status` is set to *number*. This parameter is useful only for nested Exec files since it is used to terminate the calling Exec file if a nested Exec file terminates with an error.

EXPORT

`EXPORT` [*variable* ...]

This internal command makes the specified variables available to Exec files called by the current Exec file.

variable ... The names of the variables you wish to make available to enclosed Exec files. Variable names are not case sensitive, and only the first 255 characters are significant. If you omit *variable*, a list of all exported variables (for the current Exec file) is written to standard output.

The following statements describe the action of EXPORT commands:

- Variables included in EXPORT commands in a shell command line can be used within any Exec file called from the command line.
- Variables included in EXPORT commands in an Exec file can be used in any Exec file called by that file.
- Exported variable definitions are passed on to any Exec files enclosed at lower levels.
- An EXPORT command does not affect the values of variables in an Exec file that *called* the file that includes the EXPORT statement.
- Variables defined within an Exec file and not exported are local to that file.
- When a variable that has been exported is redefined, the new value is valid for all Exec files enclosed at lower levels without the necessity of reexporting the variable.
- Variables exported and redefined within an enclosed Exec file revert to their original values when the enclosed Exec file is terminated.
- Variables exported from the LOGIN file act as if they had been exported from the command level.

For example, suppose that you execute the Exec file EXEC.1 from the shell command line and that EXEC.1 calls EXEC.2, and EXEC.2 calls EXEC.3. In this case, the followng statements are true:

- A variable defined on the command line and specified in an EXPORT statement is valid in EXEC.1, EXEC.2, and EXEC.3.
- A variable specified in an EXPORT statement in EXEC.1 is valid in EXEC.2 and EXEC.3 but not on the command line.
- A variable exported from EXEC.2 is valid in EXEC.3, but not in EXEC.1 or on the command line.
- If a variable is defined in EXEC.1 and exported, and then redefined in EXEC.2, its value is changed in EXEC.2 and EXEC.3 but not in EXEC.1 or the command level.

The LOGIN file, which is executed at APW boot time, constitutes a special case of the use of EXPORT commands. Variables included in an EXPORT command in the LOGIN file are exported to the shell command level when the LOGIN file is executed at boot time. These variable definitions are valid at all levels of nested Exec files.

Note that you do not need to use an EXPORT command to use variable definitions in an Exec file that you call with an EXECUTE command. See the discussion of the EXECUTE command for details.

FOR-END

```
FOR variable [IN value1 value2 ... ]
   .
   .
   .
END
```

This command sequence creates a loop that is executed once for each parameter value listed.

variable The name of the variable whose value changes each pass through the loop. If *variable* has not been previously defined, this statement defines it.

IN *value1 value2* ... Each value or string listed after the optional parameter IN is assigned to *variable* for one pass through the loop. That is, the first time through the loop {*variable*} is equal to *value1*; the second time through the loop {*variable*} is equal to *value2*, and so forth. The values of *value* must be separated by one or more spaces.

If IN is omitted, the parameters listed after the Exec file pathname (when the Exec file is called) are used. The Exec file pathname itself (parameter 0) is not used as a value for *variable*.

END Each of the commands between FOR and END is executed once for each value of *value* (or for each parameter, if IN is not used). If *variable* appears in any of these statements, it takes on the current value of *value*.

For example, the following Exec file, named ERASE, would delete from a directory all files that ended in the extensions .OLD, .BAK, and .TEST. Note that the equal sign used here is a wildcard character in the DELETE command, not an Exec-file logic operator:

```
ERASE

FOR EXT IN OLD BAK TEST
DELETE =.{EXT}
END
```

The same result could be obtained by including the extensions as parameters on the command line and omitting them from the FOR command:

```
ERASE OLD BAK TEST

FOR EXT
DELETE =.{EXT}
END
```

IF–END

```
IF expression
   .
   .
   .
[ELSE IF expression]
   .
   .
   .
[ELSE]
   .
   .
   .
END
```

This command sequence provides conditional branching in Exec files. The expressions are tested until one evaluates as true, then the statements between that IF or ELSE IF and the following ELSE IF, ELSE, or END are executed. All other statements between the IF and END are skipped. If none of the expressions evaluate as true and if an ELSE statement is included, the statements between the ELSE and the END are executed.

expression Any expression formed with one of the logical operators discussed in the section "Logic Operators" earlier in this chapter.

LOOP–END

```
LOOP
    .
    .
    .
END
```

This command sequence defines a loop that repeats continuously until a BREAK or EXIT command is encountered. The loop is also terminated if any command in the loop returns a nonzero error status while {Exit} is not null (see the section "Variables" in this chapter for a discussion of Exit).

SET

SET [*variable* [*value*]]

This internal command allows you to assign a value to a variable name. You can also use this command to obtain the value of a variable or a list of all defined variables.

variable The variable name you wish to assign a value to. Variable names are not case-sensitive, and only the first 255 characters are significant. If you omit *variable*, a list of all defined names and their values is written to standard output.

value The string that you wish to assign to *variable*. Values are case-sensitive and are limited to 255 characters. All characters, including spaces, starting with the first nonspace character after *variable* and continuing to the end of the line, are included in *value*. If you include *variable* but omit *value*, the current value of *variable* is written to standard output.

Use the UNSET command to delete the definition of a variable. Variables defined within an Exec file and not exported are local to that file. See the discussions of the EXPORT and EXECUTE commands for ways to share variable definitions between Exec files.

Important: Certain variable names are reserved. See the section "Variables" earlier in this chapter for a list of reserved variable names.

UNSET

UNSET *variable*

This internal command deletes the definition of a variable.

variable The name of the variable you wish to delete. Variable names are not case-sensitive, and only the first 255 characters are significant.

Use the SET command to define a variable. Variables defined within an Exec file and not exported are local to that file. See the discussions of the EXPORT and EXECUTE commands for ways to share variable definitions between Exec files.

Example

When the following Exec file is executed, it attempts to assemble and link a source file. If the operation is unsuccessful, it attempts to assemble and link a different source file. If neither program can be assembled and linked, the Exec file writes a message to the screen. If either file can be assembled and linked, then that program is run.

```
UNSET EXIT                  ;*Don't abort the program if
*                             an assemble or link fails.
SET Message No luck!        ;*Message to send if we fail.
ASML PROG1 KEEP=TEST1       ;*Attempt to assemble and link
*                             the first program.
IF {Status} == 0            ;*If first prog was successful
TEST1                       ;*run the program and
EXIT                        ;*quit.
ELSE                        ;*If first prog failed
ASML PROG2 KEEP=TEST2       ;*attempt to assemble and link
*                             the second program.
IF {Status} == 0            ;*If second prog was successful
TEST2                       ;*run the program and
EXIT                        ;*quit.
ELSE                        ;*If both programs failed
ECHO {Message}              ;*send message.
END                         ;*End of second IF statement
END                         ;*End of first IF statement.
```

Note: When reading this example, remember that equal signs (=) can have three different functions in Exec files. They can function 1) as a wildcard character in a filename; 2) as part of an APW command parameter (for example, KEEP=TEST); 3) in the string-comparison operators == and !=.

Chapter 4

Editor

The APW Editor allows you to write and edit source and text files for use with APW assemblers, compilers, and utility programs. A brief introduction to the use of the editor is given in the section "Using the Editor" in Chapter 2. This chapter provides reference material on the editor. All editing commands are described in detail.

The first section in this chapter, "Modes," describes the different modes in which the editor can operate. The second major section, "Command Descriptions," describes each editor command and gives the keys or key combinations assigned to the command. The third major section, "Macros," describes how to create and use editor macros, which allow you to execute a string of editor commands with a single keystroke. The fourth section, "Setting Editor Defaults," describes how to set the defaults for editor modes and tab settings for each language.

An on-line help facility is available for the editor. To see the help file, press Apple-Slash (⌘-/) or Apple-Question Mark (⌘-?), then use the Up Arrow (↑), Down Arrow (↓), Apple-Up Arrow (⌘-↑), and Apple-Down Arrow (⌘-↓) keystrokes to scroll through the help file. Press Esc, Return, or Enter to return to the file you are editing.

Modes

The behavior of the APW Editor depends on the settings of several modes, as follows:

- insert
- escape
- auto indent
- select
- automatic wrap

Each of these modes has two possible states; you can toggle between the states while in the editor. All of these modes are described in this section. The commands for toggling modes are described in the section "Command Descriptions" later in this chapter. For example, to learn how to toggle wrap mode, look up "Toggle Wrap Mode."

The default settings for the auto-indent, select, and word-wrap mode depend on the language type of the file you are editing. You can change the default settings for a language, as described in the section "Setting Editor Defaults" in this chapter.

Insert

When you first start the editor, it is in overstrike mode; in this mode the characters you type replace any characters the cursor is on. If you press Control-E or Apple-E to toggle to insert mode, any characters you type are inserted at the left of the cursor while the cursor, the character the cursor is on, and any characters to the right of the cursor are moved to the right.

Although the editor can display only 80 columns of text, you can continue to insert characters into an 80-column line when the cursor is in any column other than the last. If you do so, the characters at the end of the line move off the screen to the right. The maximum length of a line in the APW Editor is 255 characters (including spaces). If you insert characters after the line is 255 characters long, the characters at the end of the line are lost. To bring characters beyond column 80 back into view, insert a carriage return near the end of the line; the characters are moved to the next line down.

> **Note:** If the editor is in insert mode and you continue typing when the cursor reaches column 80, each additional character is inserted at column 80 and the characters to the right of the cursor move off the screen to the right. As a result, the new characters you type are inserted in reverse order. For example, if you start typing in column 76 and type `12345` when the editor is in insert mode, the `5` is in column 80. If you then type `6`, the `6` is inserted at column 80 and the `5` moves off the screen to the right. If you continue with `789`, and then insert a carriage return before the `1` to move the string to the next line, you will find you have inserted `123498765` into the file.

If the editor is both in insert and automatic-wrap modes, when the cursor reaches the end-of-line marker (usually at column 80 as explained in the section "Setting Editor Defaults" later in this chapter), the editor inserts a carriage return before the word you are currently typing. The result is that the word that included column 80 and all remaining characters on the line (up to the 255th character) are moved to the next line down. See the section "Automatic Wrap" later in this chapter for an example.

To toggle from insert mode to overstrike mode, press Control-E or Apple-E one more time.

Escape

When you press the Esc key or Control-Underscore (Control-_), the editor enters escape mode. Escape mode has several special features:

- You can cause a command to be repeated automatically up to 32767 times while in escape mode by typing the number of repetitions before you execute the command. For example, the command Control-T deletes a line of text, so to delete 10 lines of text (starting with the line the cursor is on), type Esc 10 Control-T.
- If it is impossible for the editor to repeat a command as many times as you specify, it repeats it the maximum number of times possible. For example, if you type Esc 50 Up Arrow when you are only 20 lines from the top of the file, the cursor moves up 20 lines (to the top of the file) and stops.
- Although you can type letters and punctuation in escape mode as you can in edit mode, to type a numeral in escape mode you must hold down the Apple key.

To exit escape mode and return to edit mode, press Esc one more time or press Control-Apple-Underscore (Control-Ö-_).

Auto Indent

You can set the editor so that pressing Return moves the cursor to the first column of the next line (in this case, auto-indent mode is said to be off), or so that it follows indentations already set in the text (auto-indent mode is on). When you press Return while auto-indent mode is on, the editor puts the cursor on the first nonspace character in the next line. If the line is blank, the cursor is placed in the same column as the first nonspace character in the first nonblank line above the cursor. If the screen is blank, the cursor is placed in column 1.

Auto-indent mode is convenient for writing programs in some high-level languages, such as Pascal, in which lines are indented to help clarify the structure of the program.

Press Apple-Return, Apple-Enter, or Control-Apple-M to toggle auto-indent mode off or on.

Select

The Cut, Copy, and Delete commands require that you first select a block of text. The APW Editor has two modes for selecting text: line-oriented and character-oriented selects. As you move the cursor in line-oriented select mode, text or code is marked a line at a time. In the character-oriented select mode, you can start and end the marked block at any character. Line-oriented select mode is the default for assembly language; for text files and most high-level languages, character-oriented select mode is the default.

While in either select mode, the following cursor-movement and screen-scrolling commands are active:

- Bottom of Screen/Page Down
- Top of Screen/Page Up
- Cursor Down
- Cursor Up
- Screen Moves

In addition, while in character-oriented select mode, the following cursor-movement commands are active:

- Cursor Left
- Cursor Right
- Start of Line
- End of Line
- Tab
- Tab Left
- Word Right

- Word Left

As you move the cursor, the text between the original cursor position and the final cursor position is marked (in inverse characters). Press Return to complete the selection of text. Press Esc to abort the operation, leave select mode, and return to normal editing.

Press Control-Apple-X to toggle between line-oriented and character-oriented select modes.

Automatic Wrap

For line-oriented computer languages like assembly language, each program statement must fit on one line; for such languages, you may not want the editor to automatically break a line of text and keep entering text on the next line. For other languages and for text files, it is better if the editor continues entering text when you reach the end of the line by automatically inserting a carriage return and moving the cursor to the next line down. You can toggle the APW Editor between these two modes of operation by pressing Control-Apple-W.

In nonwrap mode, when you reach the end-of-line mark (usually at column 80 as explained in the section "Setting Editor Defaults" later in this chapter), any additional characters you type overwrite the last character on the line. In automatic-wrap mode, when you type one character too many to fit on the line, the entire word that that character is part of is wrapped to the next line. For example, suppose you are typing the word `pneumatolysis`, and the letter *t* falls on column 79. In nonwrap mode, the additional characters overwrite the last character on the line and the line ends with `pneumats`; in automatic-wrap mode, on the other hand, the entire word `pneumatolysis` is moved to the beginning of the next line.

> **Note:** The APW Editor does not have "soft" carriage returns; that is, once a line is broken by the automatic-wrap feature, there is a permanent carriage return at the end of the line. If you delete characters on the first line, the following line does *not* move back up to maintain the length of the first line. To remove the carriage return you must first enter insert mode, then move the cursor to the beginning of the second line, and finally execute a Delete Character Left command.

If the editor is in automatic-wrap mode, when the cursor reaches the end of the line (usually column 80), the editor inserts a carriage return before the word you are currently typing. If the editor is both in insert and automatic-wrap modes, the characters to the right of the cursor are pushed off the screen to the right until the cursor reaches the end of the line, and then the word that included column 80 and all remaining characters on the line (up to the 255th character) are moved to the next line down.

Note that the line does *not* wrap when the last character in the line reaches column 80 but when the *cursor* reaches column 80. For example, suppose you begin inserting characters in the following line. The editor displays only the first 80 characters on the line. Column numbers are shown above the line for purposes of illustration only. The cursor is shown as a solid square (■) at the end of the line.

```
.........1.........2.........3.........4.........5.........6.........7.........8
ted earlier, the minerals in this specimen appear to have pneumatolysis.■
```

Now, with insert and automatic-wrap modes active, you begin to type characters in column 60:

```
.........1.........2.........3.........4.........5.........6.........7.........8

ted earlier, the minerals in this specimen appear to have formed as a result of■
```

The *f* in *of* (the last character of the newly inserted text) has reached column 79, so the cursor is in column 80 and the line wraps as follows:

```
.........1.........2.........3.........4.........5.........6.........7.........8

ted earlier, the minerals in this specimen appear to have formed as a result
of pneumatolysis.■
```

If you paste characters into the line or insert spaces with the Insert Space command, the line doesn't wrap; instead, the characters at the end of the line move off the screen to the right. The maximum length of a line in the APW Editor is 255 characters (including spaces). If you insert text or spaces after the line is 255 characters long, the characters at the end of the line are lost. To bring characters beyond column 80 back into view, insert a carriage return near the end of the line; the characters are moved to the next line down.

Command Descriptions

This section describes the function of each of the editor commands. The keystrokes used for each command are shown with the command description. Note that for many of the commands, there is more than one keystroke that executes the command. You can use whichever keystroke you prefer; there is no functional difference between alternate ways of executing a given command.

If you are familiar with the commands and just need a summary of the keystrokes to use for each command, see Appendix B.

> **Note:** Screen-movement descriptions in this manual are based on the direction the display screen moves through the file, not the direction the lines appear to move on the screen. For example, if a command description says that the screen scrolls down one line, it means that the lines on the screen move *up* one line, and the next line in the file becomes the bottom line on the screen.

Beep the Speaker

Control-G

The ASCII control character BEL ($07) is sent to the output device. Normally, this causes the speaker to beep.

Begin Macro Definitions

See Define Macros.

Beginning of Line

Ó-,
Ó-<

The cursor is placed in column 1 of the line it is in.

Bottom of Screen / Page Down

Control-Ó-J
Ó-↓

The cursor moves to the last visible line on the screen, preserving the cursor's column position. If the cursor is already at the bottom of the screen, the screen scrolls down one screen's height. For example, if the screen is 22 lines high, the screen scrolls down 22 lines.

Change

See Search and Replace.

Clear

Ó-Delete

When you execute the Clear command, the editor enters select mode, as discussed in the section "Select" earlier in this chapter. Use any of the cursor-movement or screen-scroll commands to mark a block of text (all other commands are ignored) and then press Return. The selected text is deleted from the file. (To cancel the Clear operation without deleting the block from the file, press Esc instead of Return.)

> **Important:** The Undo Delete command does not work for text removed with the Clear command. Use the Cut command to remove a block of text from the document if you want to be able to restore it later.

Copy

Control-C
Ó-C

When you execute the Copy command, the editor enters select mode, as discussed in the section "Select" earlier in this chapter. Use cursor-movement or screen-scroll commands to mark a block of text (all other commands are ignored), and then press Return. The selected text is written to the file SYSTEMP in the work prefix. (To cancel the Copy operation without writing the block to SYSTEMP, press Esc instead of Return.) Use the Paste command to place the copied material at another position in the file.

Cursor Down

Control-J
↓

The cursor is moved down one line, preserving its column position. If it is on the last line of the screen, the screen scrolls down one line.

Cursor Left

Control-H
←

The cursor is moved left one column. If it is in column 1, the command is ignored.

Cursor Right

Control-U
→

The cursor is moved right one column. If it is on the end-of-line marker (usually column 80), the command is ignored.

Cursor Up

Control-K
↑

The cursor is moved up one line, preserving its column position. If it is on the first line of the screen, the screen scrolls up one line. If the cursor is on the first line of the file, the command is ignored.

Cut

Control-X
Ó-X

When you execute the Cut command, the editor enters select mode, as discussed in the section "Select" earlier in this chapter. Use cursor-movement or screen-scroll commands to mark a block of text (all other commands are ignored) and then press Return. The selected text is written to the file SYSTEMP in the work prefix and deleted from the file. (To cancel the Cut operation without cutting the block from the file, press Esc instead of Return.) Use the Paste command to place the cut text at another location in the file.

Define Macros

Ó-Esc

The editor enters the macro-definition mode. Press Option-Esc to terminate a definition, and then press Option to terminate macro-definition mode. The macro-definition process is described in the section "Macros" later in this chapter.

Delete

See Clear, Delete Character, Delete Character Left, Delete Line, Delete to EOL, Delete Word.

Delete Character

Control-F
Ó-F

The character that the cursor is on is deleted and put in the Undo buffer (see the description of the Undo Delete command). Characters to the right of the cursor are moved one space to the left to fill in the gap. The last column on the line is replaced by a space.

Delete Character Left

Delete
Control-D

The cursor is moved left one column, and a Delete Character command is executed. If the cursor is in column 1 and the overstrike mode is active, no action is taken. If the cursor is in column 1 and the insert mode is active, the line the cursor is on is appended to the line above and the cursor remains on the character it was on before the delete.

Delete Line

Control-T
Ó-T

The line that the cursor is on is deleted and the following lines are moved up one line to fill in the space. The deleted line is put in the Undo buffer (see the description of the Undo Delete command).

Delete to EOL

Control-Y
Ó-Y

The character that the cursor is on and all the characters to the right of the cursor to the end of the line are deleted and put in the Undo buffer (see the description of the Undo Delete command).

Delete Word

Control-W
Ó-W

When you execute the Delete Word command, the cursor is moved to the beginning of the word it is on, then Delete Character commands are executed for as long as the cursor is on a nonspace character. This command thus deletes the word plus all punctuation up to the next space character or the end of the line, whichever comes first. If the cursor is on a space when the command is executed, that space and all following spaces are deleted, up to the start of the next word. All deleted characters, including punctuation and spaces, are put in the Undo buffer (see the description of the Undo Delete command).

End Macro Definition

Option-Esc

When you are in macro definition mode, press Option-Esc to terminate a definition, and then press Option to terminate macro-definition mode. The macro-definition process is described in the section "Macros" later in this chapter.

End of Line

Ó-•
Ó->

If the last column on the line is not blank, the cursor moves to the last column. If the last column is blank, the cursor moves to the right of the last nonspace character in the line. If the entire line is blank, the cursor is placed in column 1.

> **Note:** The editor automatically deletes any space characters at the end of a line, so this command puts the cursor to the right of the last actual character on the line.

Enter Escape Mode

See Turn On Escape Mode.

Execute Macro

Option-*letter*

Use this command to execute a macro that you have defined. The macro-definition process is described in the section "Macros" later in this chapter.

Find

See Search.

Help

Ô-?
Ô-/

A window containing the contents of the SYSHELP file in the system prefix appears on the screen. Use the Up Arrow, Down Arrow, Apple-Up Arrow and Apple-Down Arrow to scroll through the file. Press Return, Enter, or Esc to return to the editor window. Any other key is ignored.

Insert Line

Control-B
Ô-B

A blank line is inserted at the cursor position, and the line the cursor was on and all subsequent lines are moved down to make room. The cursor remains in the same position on the screen.

Insert Space

Ô-Space bar

A space is inserted at the cursor position. Characters from the cursor to the end of the line are moved right to make room. Any character in column 255 on the line is lost. The cursor remains in the same position on the screen. Note that since spaces to the right of the last character on the line are not significant, the Insert Space command has no effect when the cursor is at the end of the line. Note also that the Insert Space command can extend a line past the end-of-line marker.

Paste

Control-V
Ô-V

The contents of the SYSTEMP file are copied to the current cursor position. If the editor is in line-oriented select mode, the line the cursor is on and all subsequent lines are moved down to make room for the new material. The cursor's column position is unchanged.

If the editor is in character-oriented select mode, the material is copied at the cursor's present position. The cursor remains in the same position on the screen. Characters from the cursor to the end of the line are moved right to make room.

It is best to use the same mode for pasting in text as you used when you cut or copied the text.

> **Warning:** If enough characters are inserted to make the line longer than 255 characters, the excess characters are lost.

If you attempt to execute the Paste command when no Cut or Copy command was executed (that is, there is no SYSTEMP file), the following error message appears on the screen:

```
ProDOS: File not found
```

Quit

Control-Q
Ó-Q

Exit to the editor's Quit menu. The following options are listed, followed by the prompt `Enter selection:`

R Return control to the editor. You are returned to same editing mode and to the same position in the file you were at when you quit it.

S Save the file to the current filename (shown at the top of the menu) and return to the Quit menu.

N Save the file to a new filename. You are prompted for a new filename, and the file is saved to that filename; then you are returned to the Quit menu. You can enter a full or partial pathname for the file, and you can use device names and prefix numbers as described in the section "Entering Commands" in Chapter 2.

If a text file or APW source file with the same name as the file you have specified already exists, you are prompted for verification before the old version is overwritten. If a file that is not a text or APW source file exists and has the name you have specified, you are not allowed to overwrite it. Instead, the following message appears at the bottom of the screen:

```
Incompatible file format.
Hit ESC to continue.
```

When you press Esc, the prompt `Enter selection:` reappears. Press N again to enter another filename.

L Load a file. You are prompted for a filename, and that file is loaded from disk. If the filename you specify is not on the disk, a new file is opened with that name. If you have not yet saved the changes to the file you just quit, you are asked to verify that you don't want to save those changes before the new file is loaded.

You can enter a full or partial pathname for the file, and you can use device names, prefix numbers, and wildcards as described in the section "Entering Commands" in Chapter 2. If you specify a wildcard character, the first filename matched is used. If this file is the wrong file type, the the following message appears at the bottom of the screen:

```
Incompatible file format.
Hit ESC to continue.
```

When you press Esc, the prompt `Enter selection:` reappears. Press L again to enter another filename.

When the file you specify is loaded, the editor places the cursor on the first character in the file. If the new file has the same language type as the previous one, the editor does not reset default modes and parameters; if you do change languages, the editor is set to the default parameters in the `SYSTABS` file for the new file's language.

E Leave the editor and return to the shell. If you have not yet saved the changes to the file you just quit, you are asked to verify that you want to quit the editor without saving changes.

Press the letter corresponding to the option you want and enter a pathname if prompted to do so. If you press Return without entering any other data in response to a prompt, the command is aborted and control returns to the menu.

Quit Macro Definitions

Option

When you are in macro definition mode, press Option-Esc to terminate a definition, and then press Option to terminate macro-definition mode. The macro-definition process is described in the section "Macros" later in this chapter.

Remove Blanks

Control-R
Ó-R

If the cursor is on a blank line, that line and all subsequent blank lines up to the next nonblank line are removed. If the cursor is not on a blank line, the command is ignored.

Repeat Count

1 to 32767

When in escape mode, you can enter any number from 1 to 32767 immediately before a command, and the command is repeated as many times as you specify (or as many times as is possible, whichever comes first). Escape mode is described in the section "Escape" earlier in this chapter.

Return

Return
Control-M

The Return key works in one of two ways, depending on the setting of the auto-indent mode toggle. Pressing the Return key can 1) move the cursor to column 1 of the next line; or 2) place the cursor on the first nonspace character in the next line or, if the next line is blank, move the cursor down one line and place it in the same column as the first nonspace character in the first nonblank line on the screen above the cursor. If the screen is blank, the cursor is placed on column 1 of the next line.

If the cursor is on the last line on the screen, the screen scrolls down one line.

Screen Moves

Ó-1 to Ó-9

The file is divided by the editor into eight approximately equal sections. Each of the screen-move commands Apple-2 (Ó-2) through Apple-8 (Ó-8) moves the display to one of the boundaries between two of these sections. The cursor remains in the same position (that is, the same line and column) on the screen. The command Apple-1 moves the cursor

to the first character in the file, and Apple-9 moves the cursor to the last character in the file.

Scroll Down One Line

Control-P
⌘-P

The editor moves down one line in the file, causing all of the lines on the screen to move up one line. The cursor remains in the same position on the screen. Scrolling can continue past the last line in the file.

Scroll Down One Page

See the Bottom of Screen/Page Down command.

Scroll Up One Line

Control-O
⌘-O

The editor moves up one line in the file, causing all of the lines on the screen to move down one line. The cursor remains in the same position on the screen. If the first line of the file is already displayed on the screen, the command is ignored.

Scroll Up One Page

See the Top of Screen/Page Up command.

Search Down

⌘-L

This command allows you to search through a file for a character or string of characters. When you execute this command, the prompt `Search string` appears at the bottom of the screen. If you have previously entered a search string, the previous string appears after the prompt as a default. Type in the string for which you wish to search, and press Return. Searches are not case-sensitive, and they include all occurrences of the string, whether it is embedded in a longer string or not. For example, if you search for the string NOT, any of the following strings could be found:

not

Note

prothonotary

Important: Any spaces at the end of the line in a search string are significant but not visible. Press ♁-> to move the cursor to the end of the line to see whether there are any trailing spaces in the search string.

The following editing commands are active when you are entering the search string:

←	Cursor Left
→	Cursor Right
♁-> or ♁-.	End of Line
♁-< or ♁-,	Start of Line
Delete	Delete Character Left
♁-Y or Control-Y	Delete to End of Line
♁-Z or Control-Z	Undo changes
♁-E or Control-E	Toggle Insert Mode

In addition, the following commands are used to terminate the search string:

Esc, Clear, or Control-X	cancel command without saving changes
Return or Enter	save changes and execute command

When you press Return, the editor looks from the cursor position toward the end of the file for the search string. If the string is found, the screen is moved so that the next occurrence of the string is on the top line. The cursor is placed on the first character of the target string. The search stops at the end of the file. To search between the current cursor location and the beginning of the file, use the Search Up command.

If the string is not found, the following message appears on the screen:

```
String Not Found
```

Search Up

♁-K

This command operates exactly like Search Down, except that the editor looks for the search string starting at the cursor and proceeding toward the beginning of the file. The search stops at the beginning of the file. To search between the current cursor location and the end of the file, use the Search Down command.

Search and Replace Down

♁-J

This command allows you to search through a file for a character or string of characters and to replace the search string with a replacement string. When you execute this command, the prompt `Search string` appears at the bottom of the screen. If you have previously

entered a search string, the previous string appears after the prompt as a default. Type in the string for which you wish to search, and press Return. Searches are not case-sensitive, and they include all occurrences of the string, whether it is embedded in a longer string or not.

When you enter the search string and press Return, the prompt `Replace string` appears at the bottom of the screen. If you have previously entered a replacement string, the previous string appears after the prompt as a default. Enter the string with which you want to replace the search string, and press Return.

If you press Return without entering any replace string, the prompt `Replace with null string (Y N Q)?` appears. Press Y to delete each occurrence of the search string. Press N to return to the `Replace string` prompt. Press Q to quit the Search and Replace operation and return to editing the file.

After you enter a replace string and press Return, or press Y in response to the `Replace with null string` prompt, the prompt `Auto or Manual (A M Q)?` appears.

`A` Press A to cause all occurrences of the search string from the cursor position to the end of the file to be replaced automatically. The cursor returns to the starting point when the replacement is done.

`M` If you Press M, then when the search string is found, it is highlighted on the top line of the screen and the prompt `Replace (Y N Q)?` appears at the bottom of the screen. Press Y to replace the string and search for the next occurrence; N to leave this occurrence of the string unchanged and search for the next occurrence; or Q to leave the string unchanged and terminate the search and replace operation. When the operation is finished, the cursor returns to its starting point.

`Q` Press Q to terminate the search and replace operation and return to the file you are editing.

Important: Any spaces at the end of the line in a search string or replacement string are significant but not visible. Press ♁-> to move the cursor to the end of the line to see whether there are any trailing spaces in the search or replacement strings.

The following editing commands are active when you are entering text in response to the `Search String` and `Replace String` prompts.

←	Cursor Left
→	Cursor Right
♁-> or ♁-.	End of Line
♁-< or ♁-,	Start of Line
Delete	Delete Character Left
♁-Y or Control-Y	Delete to End of Line
♁-Z or Control-Z	Undo changes
♁-E or Control-E	Toggle Insert Mode

In addition, the following commands are used to terminate the search and replace strings:

Esc, Clear, or Control-X	cancel command without saving changes
Return or Enter	save changes and go on to next prompt

When you enter a replacement string and press A or M, the editor looks from the cursor position toward the end of the file for the search string. The search stops at the end of the file. To search between the current cursor location and the beginning of the file, use the Search and Replace Up command.

Search and Replace Up

Ó-H

This command operates exactly like Search and Replace Down, except that the editor looks for the search string starting at the cursor and proceeding toward the beginning of the file. The search stops at the beginning of the file. To search between the current cursor location and the end of the file, use the Search and Replace Down command.

Set and Clear Tabs

Ó-Tab
Control-Ó-I

If there is a tab stop in the same column as the cursor, this command clears it; if there is no tab stop in the cursor column, this command sets one.

Tab settings remain in effect only as long as you are editing the current file. Tab settings are not saved with a file. If you close the current file and open a new file, the default tab settings are used.

Start of Line

See Beginning of Line.

Tab

Tab
Control-I

The cursor is moved to the next tab stop. If there are no more tab stops, the cursor is moved to the end of the line. If the editor is in insert mode, space characters are inserted from the cursor's starting location to the tab stop; any characters to the right of the cursor are moved to the right to make room. If the editor is in overstrike mode, on the other hand, the tab acts only as a cursor-control command: no space characters are inserted.

Note that spaces to the right of the last nonspace character on the line are not significant; that is, the editor never puts spaces at the end of a line.

Tab Left

Control-A
Ó-A

The cursor is moved to the previous tab stop, or to the beginning of the line if there are no more tab stops to the left of the cursor. This command does not enter any characters in the file.

Toggle Auto Indent Mode

Ó-Return
Ó-Enter
Control-Ó-M

If the editor is set to put the cursor on column 1 when you press Return, it is changed to put the cursor on the first nonspace character in the next line. If the editor is set to move the cursor to the first nonspace character on the next line, it is changed to put the cursor on column 1. The auto-indent mode is described in the section "Auto Indent" earlier in this chapter.

Toggle Escape Mode

Esc

If the editor is in the edit mode, it is put in escape mode; if it is in escape mode, it is put in edit mode. See also the Turn On Escape Mode and Turn Off Escape Mode commands. Escape mode is described in the section "Escape" earlier in this chapter.

Toggle Insert Mode

Control-E
Ó-E

If insert mode is active, the editor is changed to overstrike mode. If overstrike mode is active, the editor is changed to insert mode. Insert and overstrike modes are described in the section "Insert" earlier in this chapter.

Toggle Select Mode

Control-Ó-X

If the editor is set to select text for the Cut, Copy, and Delete commands in units of one line, it is changed to select individual characters instead; if it is set to character-oriented selects, it is toggled to select whole lines. See the section "Select" earlier in this chapter for more information on select mode.

Toggle Wrap Mode

Control-Ö-W

If the editor is set to stop at the end of a line and ignore addtional characters, it is changed to insert a carriage return after the last full word in the line and continue entering text on the next line. If it is set to wrap lines, it is changed to stop at the end of the line. The wrap mode is described in the section "Automatic Wrap" earlier in this chapter.

Top of Screen / Page Up

Control-Ö-K
Ö-↑

The cursor moves to the first visible line on the screen, preserving the cursor's horizontal position. If the cursor is already at the top of the screen, the screen scrolls up one screen's height (for example, if the screen is 22 lines high, the screen scrolls up 22 lines). If the cursor is at the top of the screen and less than one screen's height from the beginning of the file, then the screen scrolls to the beginning of the file.

Turn Off Escape Mode

Control-Ö-_

If the editor is in escape mode, it is put in edit mode. If the editor is in edit mode, this command does nothing. This command is especially useful in editor macros, where you can use it to assure that edit mode is turned on. See also the Turn On Escape Mode and Toggle Escape Mode commands. Escape mode is described in the section "Escape" in this chapter.

Turn On Escape Mode

Control-_

If the editor is in edit mode, it is put in escape mode. If the editor is in escape mode, this command does nothing. This command is especially useful in editor macros, where you can use it to assure that escape mode is turned on. See also the Turn Off Escape Mode and Toggle Escape Mode commands. Escape mode is described in the section "Escape" earlier in this chapter.

Undo Delete

Control-Z
Ö-Z

The last character or block of characters deleted using the Delete Character, Delete Character Left, Delete Line, Delete to EOL, or Delete Word commands is inserted at the

cursor position. If the cursor has not been moved, the file is restored to its state before the delete.

> **Important:** The Undo Delete command does not work for blocks of text deleted with the Cancel command. Use the Cut command to remove a block of text from the document if you want to be able to restore it later.

The Undo buffer functions as a stack, so multiple undos are possible. For example, suppose you delete the errors (shown in boldface) in the following text, in the order in which they appear (that is, first the `e`, then the `l`, and so on):

```
Ite would appear that an appppeal to reason would not go
unAanswered.
```

When you execute the Undo Delete command one time, the text deleted last is restored—in this case, an `a`. If you execute a second Undo Delete command, the text deleted before that, `pp`, is restored, and so on. In this example, four Undo Delete commands in a row would put the following text on the screen:

```
Apple
```

A maximum of 10240 characters can be stored in the Undo buffer. No warning is issued if you delete more than 10240 characters.

Word Left

⌥-←
Control-⌥-H

The cursor is moved to the beginning of the next nonblank sequence of characters to the left of its current position. If there are no more words on the line, the cursor is moved to the last word in the previous line, or if the previous is blank, to the last word in the first nonblank line preceding the cursor.

Word Right

⌥-→
Control-⌥-U

The cursor is moved to the start of the next nonblank sequence of characters to the right of its current position. If there are no more words on the line, the cursor is moved to the first word in the next nonblank line.

Macros

You can define up to 26 macros for the APW Editor, one for each letter on the keyboard. A macro allows you to substitute a single keystroke for up to 128 predefined keystrokes. A macro can contain both editor commands and text and can call other macros.

To define a macro, press Apple-Esc. The first ten of the current macro definitions appear on the screen. To see the next ten macros, press the Right Arrow key. Press the Right Arrow key again to see the final six macros, or press the Left Arrow key to see the previous screen of macro definitions.

Before you can redefine a macro, you must first display the current definition of that macro on the screen. After pressing Apple-Esc and using the arrow keys (as necessary) to display the macro, press the letter key that corresponds to that macro and then type in the new macro definition. Press Option-Esc to terminate the macro definition. You can include Control-*key* combinations (where *key* represents any key), Apple-*key* combinations, Option–*key* combinations, and the Return, Enter, Esc, Delete, and arrow keys. The conventions in Table 4.1 are used by the editor to display keystrokes in macros:

Table 4.1. Conventions for Displaying Keystrokes in Editor Macros

Keystroke	**Convention Used to Display the Keystroke**
Control–*key*	The uppercase character corresponding to *key* is shown in inverse video.
♁–*key*	An inverse A followed by *key* (for example, A K) .
Option–*key*	An inverse B followed by *key* (for example, B K) .
Esc	An inverse left bracket (this command is equivalent to Control-[ ).
Return	An inverse M (Control-M).
Enter	An inverse M (Control-M).
↑	An inverse K (Control-K).
↓	An inverse J (Control-J) .
←	An inverse H (Control-H).
→	An inverse U (Control-U).
Delete	A block (■).
Clear	An inverse X (Control-X).

Note: Each ♁-*key* combination or Option-*key* combination counts as two keystrokes in a macro definition. Although an ♁-*key* combination looks (in the macro definition) like a Control-A followed by *key*, and an Option-*key* combination looks like a Control-B followed by *key*, you cannot enter Control-A when you want an ♁ or Control-B when you want an Option key.

If you make a mistake while entering a macro definition, press Option-Delete to delete the character to the left of the cursor.

When you are finished entering macros, press Option-Esc to terminate the last option definition, and then press Option to end macro entry. The following prompt appears on the screen:

```
Write macros to disk?
```

Press Y to save the new macro definitions on disk. Press N to return to the file without saving the macros. Macros are saved on disk in the file SYSEMAC in the APW system prefix (prefix 4).

The commands used to create and edit macro definitions are summarized in Table 4.2.

Table 4.2. Commands Used for Defining Editor Macros

♁-Esc	Begin macro definitions.
→	Display the next screen of macro definitions.
←	Display the previous screen of macro definitions.
letter	Begin defining the macro corresponding to the letter-key *letter*. Note that *letter* must be displayed on the screen before you begin to define it.
Option-Delete	Delete the character to the left of the cursor.
Option-Esc	Terminate the macro definition.
Option	Stop defining macros and return to editing the file. If you are currently defining a macro, press Option-Esc first to terminate the macro definition, and then press Option to return to the file.

To execute a macro while in the editor, hold down Option and press the key corresponding to that macro.

For example, assume you want to define a macro that draws a box such as the one in Figure 4.1. The macro must insert the box into the file regardless of what text surrounds it, and leave the cursor in the top left corner of the box.

Figure 4.1. Output of an Editor Macro

Use the following procedure to define this macro:

1. Open an editor file and press Apple-Esc to enter macro-definition mode. The current definitions of macros A through J are now displayed on the screen. To see the macros defined for the other letter keys, press the Right Arrow key.
2. We will assign macros to the letters A, B, and C to accomplish our task. Use the Left-Arrow key to return to macros A through J.

3. Press A. The editor clears the macro definition for the letter A and places the cursor just after the `A:` near the top of the screen.
4. Type in the following command sequence, being sure to include a space between the Apple-< and the first hyphen. If you make a mistake while typing in the definition, press Option-Delete to delete the character to the left of the cursor:

   ⌘-B ⌘-< ----------

   The macro definition for the letter A now should appear as shown in Figure 4.2. This command sequence inserts a blank line in the file, moves the cursor to the left margin, and inserts a space followed by 27 hyphens.
5. Press Option-Esc to terminate the definition of macro A.
6. Press B to begin definition of macro B and then type in the following command sequence, being sure to include a space between the 27 and the Esc:

   ⌘-B ⌘-< | Control-_ 27 Esc |

   The macro definition for the letter B now should appear as shown in Figure 4.2. This sequence inserts a blank line in the file, moves the cursor to the left margin, inserts a vertical bar, enters escape mode, inserts 27 spaces, leaves escape mode, and inserts another vertical bar.

   We use the Control-_ command here to turn on escape mode because this command will do nothing if escape mode is already on. If we used Esc instead and escape mode were already on, the command would toggle escape mode *off*, and the macro would not work. Note that when the macro is finished executing, escape mode will be off, whether it was on or off when the macro was called.
7. Press Option-Esc to terminate the definition of macro B.
8. Press C to begin definition of macro C and then type in the following command sequence:

   Option-A Option-B Option-B Option-B Option-B Option-A ⌘-< ↓ →

   The macro definition for the letter C now should appear as shown in Figure 4.2. This sequence executes macro A to insert a line of dashes, executes macro B four times to insert four blank lines bracketed by vertical bars, then executes macro A again, and finally moves the cursor to the left margin, down one line, and one space to the right.
9. Press Option-Esc to terminate the definition of macro C and then press the Option key to terminate macro-definition mode. When the prompt `Write macros to disk?` appears, press Y to save the macro definitions and return to the file you were editing.

```
A: ABA< ----------
B: ABA<| 27 |
C: BABBBBBBBBA A< JU
```

Figure 4.2. Macro Definitions

Now when you press Option-C, the following sequence occurs:

1. The editor calls macro A, which inserts a blank line in the file, moving the line the cursor was on and all subsequent lines down to make room, and then puts a space in column 1 followed by a string of hyphens.
2. The editor calls macro B four times in a row. Each time macro B is executed, the last line written is pushed down out of the way and a new line is written consisting of two vertical bars separated by a string of spaces.
3. The editor calls macro A again, which inserts another blank line at the top of the four lines just written and then writes another string of hyphens.
4. The cursor moves down one line and right one column, to the first blank space in the box just created (see Figure 4.1).

Setting Editor Defaults

When you start the APW Editor, it reads the file named `SYSTABS`, which is located in the APW system prefix, and which contains the default tab-stop and editor-mode settings for each language. Because the `SYSTABS` file is an ASCII text file that you can edit with the APW Editor, you can change these defaults at any time. Note also that you can change tab settings and toggle editing modes while in the editor; the defaults set by the `SYSTABS` file only determine the configuration of the editor when a file is opened.

Each language recognized by APW is assigned a language number. The `SYSTABS` file has three lines associated with each language:

1. The language number.
2. The default settings for auto-indent, select, and word-wrap modes.
3. The default tab and end-of-line (EOL) settings.

For a discussion of APW languages, see the section "Command Types and the Command Table" in Chapter 3. A complete list of APW languages and language numbers is given in Appendix B.

The first line of each set of lines in the `SYSTABS` file specifies the language that the next two lines apply to. APW languages can have numbers from 0 to 32767 (decimal). The language number must start in the first column; leading zeros are permitted and are not significant, but leading spaces are not allowed.

The second line of each set of lines in the `SYSTABS` file sets the defaults for various editor modes, as follows:

1. The first column sets auto-indent mode. If the first column contains a 0, auto-indent mode is off when the file is opened; if it's a 1, auto-indent mode is on.
2. The second column sets select mode. If the second character is 0, the editor is set to line-oriented selects; if 1, it is set to character-oriented selects.
3. The third column sets automatic wrap mode. If the third character is 0, the cursor stops when it reaches the end of a line; if 1, the editor inserts a carriage return and wraps to the next line.
4. The fourth character is reserved for future enhancements. It should be blank or 0.
5. The fifth character is reserved for future enhancements. It should be blank or 0.

6. The sixth and any additional characters are ignored. They should be blank or 0.

The third line of each set of lines in the SYSTABS file sets default tab stops. There are 80 zeros and ones in this line, representing the 80 columns on the screen. The ones indicate the positions of the tab stops. A two in any column of this line sets the end of the line. The column containing the two then replaces column 80 as the default right margin when the editor is set to that language.

For example, the following lines define the defaults for APW 65816 assembly language and APW C:

```
3
000
00000000010000010000000000000000000000010000001000000010000000100000002
10
101
00010001000100010001000100010001000100010001000100010001000100010002
```

The first three lines in this example set the defaults for the language with language number 3: that is, APW 65816 assembly language. The second line sets auto-indent mode off, sets line-oriented selects, and sets word-wrap mode off. The third line sets tab stops in columns 10, 16, 41, 48, 56, 64, and 72, and sets the end of the line at column 80. The next three lines set the defaults for language number 10: APW C. The fifth line sets auto-indent mode on, sets line-oriented selects, and sets word-wrap mode on. The sixth line sets tab stops at every fourth column and the end of the line at column 80.

If no defaults are specified for a language (that is, there are no lines for that language in the SYSTABS file), the editor assumes the following defaults:

- Auto-indent mode off.
- Line-oriented selects.
- No word wrapping: the cursor stops at the end of the line.
- There is a tab stop every eighth column.
- The end of the line is at column 80.

Chapter 5

Linker

This chapter describes the APW Linker, including its input, output, options, and commands.

A linker is a program that locates individual program segments, resolves references between segments, and combines them into a complete, executable program. The APW Linker is independent of source-code language. It is capable of extracting specific code segments from multiple library and object files, and can create segmented load files.

The APW Linker works with any assembler or compiler that generates files conforming to the Apple IIGS object module format (OMF). The linker can join separate files produced by Apple IIGS-compatible assemblers and compilers and convert them into the form needed by the System Loader for loading into the computer. Together, these three components (assembler or compiler, linker, and loader) provide a very powerful and flexible programming facility.

Although the APW Linker is a single program, conceptually there are two APW linkers. Normally, the linker is called directly by a shell command (such as `LINK` or `ASML`). These commands provide a limited number of linker options; most linker options are either not available or are set to default values. In this manual, this aspect of the linker is referred to as the *standard linker*. Alternatively, all functions of the APW Linker can be controlled by compiling a file of linker commands, called a *LinkEd file*. In this manual, the aspect of the linker controlled by LinkEd files is referred to as the *advanced linker*.

The advanced linker is provided for programmers who require maximum flexibility from the system; for most purposes, the standard linker is completely adequate. When a statement in this book applies equally to the standard and advanced aspects of the APW Linker, the terms *APW Linker* or *linker* are used.

Operations you can perform through LinkEd commands include the following:

- selecting specific segments from an object file
- assigning object-file segments to specific load-file segments
- assigning load-file segments as static or dynamic
- specifying the exact order in which to search libraries
- controlling the diagnostic output of the linker

Most users will never need the options provided by LinkEd. The first several sections of this chapter describe features common to the standard linker and advanced linker, with emphasis on the standard linker. The advanced linker is described in detail at the end of this chapter.

The principal tasks of a linker are to bring together the segments needed for a program and to resolve global references. Because most Apple IIGS code is relocatable, the APW Linker must work together with the System Loader to resolve and relocate global references. The linker provides the relocation information necessary for the loader to relocate all references after loading. Much of the work of the linker therefore consists of constructing tables of information for the loader to interpret, so that it may load and relocate the linker's output correctly.

Operation of the Linker

This section describes

- the formats and types of input files (object files) to the linker
- the formats and types of output files (load files) that it produces
- the diagnostic output from the linker

Object Files: Input to the Linker

Object files are the output from an assembler or compiler and the input to a linker. Although both object files and load files conform to the Apple IIGS object module format (OMF), only object files can be processed by the linker. Only object-file information specifically related to the operation of the linker is discussed in this chapter; see Chapter 7 for more detailed information on the Apple IIGS object module format.

Object files (ProDOS 16 file type $B1) contain data or program code that has been translated (assembled or compiled) into machine language but that may contain unresolved references to external subroutines or data. The linker processes object files, resolves external references, and produces load files. Load files contain all the information necessary to relocate external references, and are ready to be loaded into the computer by the System Loader.

> **Note:** The default file type for the load files the linker creates is set by the APW Shell's `KeepType` variable; if `KeepType` is not set, the file type is $B5, shell load file. If you are using the advanced linker, you can use the LinkEd KEEPTYPE command to set the file type of the load file. To change the file type of an existing load file, use the shell's `FILETYPE` command. Use the shell's `SET` command to change the value of the `KeepType` variable.

Each object file consists of segments. Each segment is a separate entity that contains all the information necessary to link it with other segments. A segment consists of a header followed by a body; the header contains name, size, type, and other information about the segment, while the body consists of sequential **records**, each one of which consists of either program code or information for the linker or loader. Segments are discussed in the section "Program Segmentation" in Chapter 1 and are fully described in Chapter 7.

Library Files

Library files (ProDOS 16 file type $B2) contain object segments useful to many programs. The linker can search library files to resolve references unresolved within the program

source code. Library files are normally kept in the APW library prefix (prefix 2). When you use the standard linker, it first links the source code and any library files you specify, and then if there are any remaining unresolved references, it automatically searches the files in the library prefix until all references are resolved. The advanced linker searches only those library files that you specify in the LinkEd file.

Library files differ from object files in that each library file includes a segment called the *library dictionary segment* (segment-type KIND = $08). The library dictionary segment contains the names and locations of all segments in the library file. The linker can look through the library dictionary segment for the names of segments it needs, so the library dictionary segment allows the linker to find segments much more quickly than if it had to scan through the entire file. Library files are created from object files by the MakeLib utility program (described in the section "Command Descriptions" in Chapter 3). Each library file can be created from any number of object files.

> **Important:** Once a library file has been searched, it is not returned to by the APW Linker. Therefore, a reference in a library file cannot refer to a segment in a library file that precedes it in the directory. You can, however, use the MAKELIB program to combine as many object files into a single library file as you choose, and there are no restrictions on segments referencing each other within a single library file. The order of subroutines within a single library file can affect the time necessary to complete a link but is otherwise not important.

Partial Assemblies and Filename Conventions

When you assemble or compile a program, you can use a KEEP directive (or the equivalent for the language you are using) in the source code or the KEEP parameter in the command line to specify a filename for the output. If you are assembling or compiling the entire program, and the program consists of more than one segment, then the first segment to be executed when the program is run is placed in one file and the remaining segments are placed in a second file. If the filename you specify is MYPROG, the first file is named MYPROG.ROOT and the second one is named MYPROG.A.

> **Important:** The root filename cannot be longer than 10 characters for files to which the .ROOT extension will be appended because ProDOS 16 limits the entire filename to 15 characters. Using more than 10 characters in such a filename will result in a fatal assembler or compiler error (`Unable to open output file`).

There are two circumstances under which a file with a higher alphabetic suffix (.B, .C, and so on) is created, as follows:

- If the compile involves more than one language, the first compiler or assembler usually creates the .ROOT and .A files, the second compiler creates the .B file, and so on.
- If you include a NAMES parameter on the command line, a partial assembly or compile is performed. In this case, only the segments named are compiled, and they are placed in a file with the next available alphabetic extension. Partial assemblies are described in the section "Partial Assemblies or Compiles" in Chapter 3.

Note: You can use the CRUNCH command described in Chapter 3 to combine all the alphabetic-extension files into one .A file.

The advanced linker processes segments in the order specified by the LinkEd commands. The standard linker selects the object files to process as follows:

1. The linker first scans the output disk for a filename with the proper extension (MYPROG.ROOT in this example). The object segment in that file will become the first segment in the output (load) file.
2. The linker then looks for a .A file. If it finds one, the linker looks for a .B file, and so on, until it locates the last object file created by finding the file (with name MYPROG) with the alphabetically highest extension.
3. It takes subroutines from this file in the order encountered, links them, and places them in the load file.
4. The linker then looks at the file with the next highest extension. If it finds a subroutine that has not yet been linked, it adds it to the load file. Any subroutines with the same labels as those of already linked subroutines are assumed to be older versions and are ignored.
5. The linker continues in reverse alphabetical order through the files until they all have been searched. If there are still unresolved references, the linker assumes that they are references to library files.
6. The linker automatically searches the library directory for library files. Each library file is searched in the order in which it appears in the directory. Any library segment that corresponds to an unresolved reference is extracted, processed, and placed in the load file.

Once all the necessary segments have been located, the linker proceeds to a second pass through the file. The result of pass two is a load file (ProDOS 16 file type $B5 unless you have set the shell KeepType variable to another value), ready for loading by the System Loader. Load files are described in this section.

Load Files: Output From the Linker

Load files (types $B3 through $BE) are the result of the processing of object files by the linker (and, optionally, the shell's FILETYPE command). They contain segments that are ready to be loaded into memory by the System Loader. Load files conform to a subset of the Apple IIGS object module format and do not contain any unresolved symbolic references.

Both object files and load files are segmented, but a load segment may contain more than one object segment. In assembly language, both the object-segment name and the name of the load segment to which that object segment is to be assigned can be specified with a START, DATA, PRIVATE, or PRIVDATA directive. APW C provides the overlay function to allow you to assign subroutines to specific load segments. As a default, some APW compilers assign one load-segment name (a string of spaces) to all code segments, and another (~global) to all global variables.

When you call the linker by using an APW Shell command, the linker assigns object-file segments to load-file segments based on the load-segment names. All object-file segments with the same load-segment name are collected into a single static load segment.

The linker may produce a single load file from a single object file or from several object files, as described in the discussions of the LINK command in Chapter 3 and LinkEd command files in this chapter.

For a complete description of load files and the function of the System Loader, see the section "Object Module Format" in Chapter 7 and the description of the System Loader in the *Apple IIGS ProDOS 16 Reference* manual.

Diagnostic Output

In addition to the load file itself, the linker produces diagnostic output to show what it has done and to aid debugging. Output is sent to standard output (usually the screen). Most of the output can be suppressed, if desired, with command-line parameters. Each of the types of information output by the linker is described in the following sections. Figure 5.1 shows the sample output of a LinkEd command file.

```
Link Editor V1.0

   1 KEEP LINKTEST
   2.SOURCE ON
   3 SYMBOL ON
   4.LIST ON
   5 LINK/ALL TEST
   6 LIBRARY *

0 errors found in source file.


00000000 00000020 Code: MAIN
00000020 0000001B Code: SECOND
0000003B 0000001C Data: DATA
00000057 00000034 Code: ~COUT
0000008B 00000002 Code: STOUT



Global symbol table:

0000003B 01 DATA          00000000 00 MAIN          0000003B 01 MSG1
00000049 01 MSG2          00000049 01 MSG3          00000057 01 MSG4
00000020 00 SECOND        0000008B 00 STOUT         00000057 00 ~COUT



Segment Information:

  Number     Type      Length       Org

     1       $00       $0000008D    Relocatable

There is 1 segment, for a length of $0000008D bytes.
```

Figure 5.1. Sample Output of a LinkEd Command File

Error Messages

Errors can be caused by source-code errors in a LinkEd file, by mistakes in the command line, or by problems encountered whie trying to link an object file. Appendix C gives a full list of error messages and their meanings. Error messages cannot be suppressed.

Link Map and Source Listing

If you use the +L command-line parameter or the LinkEd `LIST ON` command, as the linker processes each segment or subroutine, it writes the starting address of the segment, the length in bytes (hexadecimal) of the segment, the segment type (code or data), and the name of the segment. If the program is relocatable, the starting-address calculation is based on the assumption that the program starts at $000000.

If you call the linker from a LinkEd file and use the +L command-line parameter or the LinkEd `SOURCE ON` command, the LinkEd source code is written to standard output. A sample LinkEd output listing is shown in Figure 5.1.

Symbol Table

If you use the +S command-line parameter or the LinkEd `SYMBOL ON` command, an alphabetized global-symbol table is printed. The table presents the following information for each symbol:

- assigned value (hexadecimal)
- classification number
- symbol name

The classification number is a pair of hexadecimal digits. If it is $00, the symbol is a global label or subroutine name; if the number is nonzero, the symbol is a data label and the value of the digit is the number of the data segment that defined it.

A sample symbol table is shown as part of the output in Figure 5.1.

> **Symbol Types:** The Apple IIGS object module format defines three types of symbols: global, private, and local. Global symbols can be referenced in any segment. For APW assembly-language programs, for example, global symbols include object-segment names defined by `START` and `DATA` directives and any symbols defined in an `ENTRY` or `GEQU` directive. Private symbols are available to any segment in the same object file, but not to segments in other object files that are part of the same program. For APW assembly-language programs, private symbols include object-segment names defined by `PRIVATE` and `PRIVDATA` directives. Local symbols are labels that are defined only within individual code or data segments.
>
> Local symbols are normally accessible only within the segment in which they appear. However, a segment may gain access to local symbols in another *data* segment by issuing a `USING` assembler directive. The `USING` directive cannot refer to a code segment.

Be sure that no two global symbols (or local symbols in data segments) with the same name appear anywhere in the program. Two private symbols with the same name cannot appear in the same object file but can appear in separate object files that are part of the same program.

The assembler or compiler resolves local references, so the linker never sees them. Therefore, local symbols never appear in the symbol table, with the exception of local labels in a data segment named in a `USING` directive.

Summary Table

When it finishes, the linker prints a summary giving the number of errors detected (if any) and the highest error level encountered (see Appendix C). A table of load segments is printed, indicating the segment number and type of each load segment created, along with its length and absolute origin (if any; see Figure 5.1). The last line tells how many segments there are, and how many bytes long the program is (in hexadecimal).

Using the Standard Linker

You can call the APW Linker by executing an APW Shell command. The following commands allow you to call the linker without having to execute a LinkEd command file:

- `ASML`
- `ASMLG`
- `CMPL`
- `CMPLG`
- `RUN`
- `LINK`

The `LINK` command differs from the other five commands in several ways. First, the `LINK` command lets you perform a link separate from the compile or assembly. The other commands call the linker automatically after a successful assembly or compile has been completed. Each of these commands lets you print the link map and symbol table; for all but the `LINK` command, however, you can print the link map only if you also print the source listing of the assembler or compiler. Finally, the `LINK` command lets you specify a name for the load file, whereas the other commands let you specify a root filename for the object files, which is then also used as the name of the load file.

Important: If you are linking object files with the root name *rootname*, make sure there are no other files in the same prefix as *rootname* with the same rootname and an alphabetic extension. For example, if you are linking `MYFILE.ROOT` and `MYFILE.A`, make sure there are no files named `MYFILE.B` or `MYFILE.C` in the same directory before linking.

The following linker defaults are used when you execute one of these APW Shell commands:

- Load-segment names are used to determine which object segments to put in which load segments: all object segments with the same load-segment name are placed in the same load segment. In assembly language, for example, you can specify the load-segment name as the operand of a `START`, `DATA`, `PRIVATE`, or `PRIVDATA` directive. Most APW compilers use a string of spaces for the load-segment name of all code segments, and thus put all global label definitions and data in segments with the load-segment name `~global`.
- Object segments are scanned in the sequence in which they appear in the object file. Load segments are placed in the load file in the order of the load-segment name's first appearance in the object file. The `LINK` command lets you specify more than one object file to be included in the link.
- If segment `KIND`s are specified in the source file and the `KIND`s of the object segments placed in a given load segment are not all the same, the segment `KIND` of the resulting load segment is unpredictable.
- Any library files specified on the command line are searched in the order in which they are listed. If any references remain unresolved after all the object and library files listed in the command line have been linked, the library files in the library prefix (prefix 2) are searched.
- The load address of absolute code must be specified in the source file; there is no command-line parameter to set a load address.
- No load file is saved to disk unless the `KEEP` parameter is used on the command line, or the `KEEP` directive is used in the source file. (If you use the `LINK` command, you must use the `KEEP` parameter on the command line to save the load file.)

If you need to have more control over the link, use a LinkEd file, as described in the following section. All of the APW Shell commands are described in the section "Command Descriptions" in Chapter 3. The file type of load files produced by the standard linker is set by the `KeepType` shell variable; the default is ProDOS 16 file type $B5. You can use the shell's `FILETYPE` command to change the file type of an existing load file or the shell's `SET` command to change the value of the `KeepType` variable.

Using the Advanced Linker

You can control every aspect of a link by using a LinkEd command file. LinkEd files are APW source files with a language type of `LINKED` (see the section "Language Types" in Chapter 2 for instructions on assigning a language type to a source file). To execute a LinkEd file, use one of the following APW Shell commands:

- `ALINK`
- `ASSEMBLE`
- `COMPILE`

Note: These are all aliases for the same command, which checks the language type of the file and calls the linker for files with language type `LINKED`.

Alternatively, you can append the LinkEd file to the last source-code file; when the compiler or assembler gets to the LinkEd file, it returns control to the APW Shell, which calls the APW Linker. If you append the LinkEd file to the last file of the source code, the

file is linked automatically every time the program is compiled or assembled. When the linker finishes processing the file, it tells the APW Shell not to call another compiler or the linker. For this reason, you can use the ASML, ASMLG, CMPL, CMPLG, and RUN commands with a LinkEd file without causing any errors. This also means, however, that LinkEd must be the last language called. All of the APW Shell commands are described in the section "Command Descriptions" in Chapter 3.

The Structure of a LinkEd File

A LinkEd file is more than a set of linker parameters stored in a file; it is a set of commands that give you a high degree of control over the link process. The following rules summarize the structure of a LinkEd file:

- LinkEd commands are processed sequentially from the beginning of the file. Because only one pass is made through the LinkEd file by the linker, the order of the commands is important.
- The name of the load file must be specified before any output is generated for it. If you have not specified a name for the load file with a KEEP parameter on the command line or by specifying a default load filename with the KeepName shell variable, then you must include a KEEP command in the LinkEd file and it must be placed before the first SEGMENT, LINK, or LIBRARY command.
- The name of the load segment must be specified before any output is generated for it. Load segment names are specified with the SEGMENT command.
- The commands that extract segments from object and library files (LIBRARY, LINK, LOADSELECT, and SELECT) may leave label references unresolved; these references can be resolved when segments are extracted by later commands. An error results only if a label remains unresolved after all the commands in the file have been executed.

LinkEd Command Descriptions

LinkEd source files consist only of LinkEd commands and comments. Each command must be on a separate line. Comments consist of either blank lines or lines that start with an asterisk (*) or semicolon (;).

LinkEd commands are case insensitive. Any combination of uppercase and lowercase letters may be used when writing commands. In the examples shown here all commands are in uppercase to help set them apart from comments and text.

> **Important:** Segment names *are* case sensitive. For case-sensitive languages (such as C), segment names must be entered in LinkEd commands exactly as they are listed in the source code, including case. For case-insensitive languages, the compiler normally writes all segment names to object files as all uppercase, so for such languages, segment names must be entered in uppercase.

The linker can produce diagnostic output to show what it has done and to aid debugging. Output is sent to standard output (usually the screen). Except for error messages, the output can be turned on or off with LinkEd commands. Where conflicting command-line parameters and LinkEd commands are used, the command line takes precedence.

The following notation is used to describe commands:

UPPERCASE	Uppercase letters indicate a command name or an option that must be spelled exactly as shown.
italics	Italics indicate a variable that you must replace with specific information, such as a pathname or address.
pathname	This parameter indicates a full pathname, including the prefix and filename, or a partial pathname, in which the current prefix is assumed. For example, if a file is named `FILE` in the subdirectory `DIRECTORY` on the volume `VOLUME`, the *pathname* parameter would be: `/VOLUME/DIRECTORY/FILE`. If the current prefix were `/VOLUME/`, you could use `DIRECTORY/FILE` for *pathname*. A full pathname (including the volume name) must begin with a slash (/); do *not*, however, precede *pathname* with a slash if you are using a partial pathname. The device names `.D1`, `.D2`, . . . `.Dn` can be used for volume names and ProDOS 16 prefix numbers or double periods (..) can be used instead of a prefix.
|	A vertical bar indicates a choice. For example, `LIST ON|OFF` indicates that the command can be entered as either `LIST ON` or as `LIST OFF`.
A|B	An underlined choice is the default value.
[ ]	Parameters enclosed in square brackets are optional.
. . .	Elipses indicate that a parameter or sequence of parameters can be repeated as many times as you wish.

APPEND

`APPEND` *linkedname*

LinkEd appends the LinkEd file with the pathname *linkedname* to the present LinkEd source file. Any statements after the `APPEND` command in the present LinkEd file are ignored.

linkedname The full or partial pathname of the LinkEd file you want to append.

COPY

`COPY` *linkedname*

LinkEd stops processing the present LinkEd file temporarily and processes all statements in the LinkEd file specified by *linkedname*. LinkEd then resumes processing the present file at the statement immediately following the `COPY` command.

linkedname The full or partial pathname of the LinkEd file to which you want to transfer control.

Copied files can copy other files, with no fixed limit to the number of nested levels. The only constraint is the amount of available memory; it is generally safe to assume that you may copy eight levels deep.

EJECT

`EJECT`

This command controls printer output. If output is to a printer, `EJECT` causes the printer to skip to the top of the next page. If output is to a CRT screen, `EJECT` has no effect.

KEEP

`KEEP` *loadname*

The typical output file produced by LinkEd is a relocatable load file, ready for loading and executing at any free memory location. A load file may contain several segments (see the discussion of the `SEGMENT` command, later in this chapter), each of which can be loaded independently and automatically during program execution.

loadname The full or partial pathname of the load file you want to create.

The `KEEP` command opens the output file (load file) specified by *loadname*. All segments subsequently processed by LinkEd are placed in *loadname*, in the order in which they are encountered. The `KEEP` command must be placed before the first statement that creates output: that is, before the first `SEGMENT`, `LINK`, or `LIBRARY` command.

The load filename is determined first by the `KEEP` parameter on the command line. If there is no KEEP parameter, the `KeepName` shell variable is used. The LinkEd `KEEP` command is used only if neither the `KEEP` parameter nor the `KeepName` shell variable is specified. Notice that the `LinkName` shell variable is used only by the `LINK` command and has no effect on LinkEd files.

Important: You cannot use a LinkEd `KEEP` command if you append the LinkEd file to your source code and the source code includes a `KEEP` directive (or equivalent).

Use the `KEEPTYPE` command to set the file type of a load file.

KEEPTYPE

`KEEPTYPE` *filetype*

This command sets the file type of the load file produced by the linker.

filetype The ProDOS 16 file type to which you want to set the load file. Use one of the following three formats for *filetype*:

- A decimal number 179–191.
- A hexadecimal number $B3–$BF.
- The three-letter abbreviation for the file type used in disk directories, as shown in Table 5.1.

The position in the LinkEd file of the KEEPTYPE command is not important.

The default file type of load files produced by the APW Linker is set by the KeepType shell variable; if this variable is null, the default is ProDOS 16 file type $B5. You can use the shell's FILETYPE command to change the file type of an existing load file or the shell's SET command to change the default file type.

Table 5.1. File Types of ProDOS Load Files

Decimal	Hex	Abbreviation	File Type
179	$B3	S16	ProDOS 16 system load
180	$B4	RTL	Run-time library
181	$B5	EXE	Shell load
182	$B6	STR	Startup load
184	$B8	NDA	New desk accessory
185	$B9	CDA	Classic desk accessory
186	$BA	TOL	Tool set file

LIBRARY

LIBRARY *libname*
LIBRARY/LOADSELECT *libname lseg*

A library file is a file of ProDOS 16 file type $B2 containing object segments, such as general utilities, that may be called by other programs. The LIBRARY command causes the linker to search the library file specified by *libname* for segments that have been referenced by a source file; any that are found are included in the output load file. See the discussion of the MakeLib utility in Chapter 3 for instructions on creating your own library files.

libname The full or partial pathname of the library file you want to search. If you use an asterisk (*) for *libname*, the linker scans all the files in the current APW library prefix (prefix 2).

/LOADSELECT If you include the /LOADSELECT parameter, only those segments with the load-segment name specified by *lseg* are searched. If the /LOADSELECT parameter is omitted, the linker ignores load segment names in library files. There cannot be any spaces between the LIBRARY command and the /LOADSELECT parameter.

lseg The load-segment name of the object segments that you want to search. To search all object segments with a blank load segment name, use an asterisk (*) for *lseg*. In case-sensitive languages, segment names must be entered exactly as they appear in the source code. Segment names in case-insensitive languages must be entered as all uppercase characters.

For example, suppose your library file MYLIB contains the object segments PETER, PAUL, and MARY, and each of these object segments is assigned either to the load segment WHITE or the load segment BLACK, as follows:

0. Object-segment name: PETER
   Load-segment name: WHITE
1. Object-segment name: PAUL
   Load-segment name: BLACK
2. Object-segment name: MARY
   Load-segment name: WHITE

The following LIBRARY command searches the file MYLIB. If an unresolved reference exists to any of the segments in MYLIB or to any of the labels in those segments, the referenced segments are extracted and linked into load segment GRAY.

```
SEGMENT GRAY
LIBRARY MYLIB
```

Suppose, on the other hand, that you use the following commands in your LinkEd file:

```
SEGMENT GRAY
LIBRARY/LOADSELECT MYLIB WHITE
```

In this case, only the object segments PETER and MARY are searched, since each of these segments has the load-segment name WHITE. If an unresolved reference exists to either of these segments or to any of the labels in these segments, that segment is extracted and linked into load segment GRAY.

The following command causes a search of all the segments with blank load-segment names in all of the files in the library prefix:

```
LIBRARY/LOADSELECT * *
```

LINK

LINK[/ALL] *objname*

This command causes the object file specified by *objname* to be included in the output file. All segments of the file specified by *objname* not already included are added to the program. If the LINK command follows a SEGMENT command, all the object segments in *objname* are placed in the load segment defined by the SEGMENT command. If the LINK command does not follow a SEGMENT command, all object segments are placed in a load segment whose name consists of ten space characters. The LINK command ignores source-code load-segment names, such as those specified by the operand of an APW Assembler START directive.

Use the SELECT command to link individual object segments from a given file.

/ALL — If you use the /ALL qualifier, all files with the root filename specified by *objname* and .ROOT or alphabetic filename extensions are searched to make sure the most recently assembled version of each file segment is included (see the section "Partial Assemblies and Filename Conventions" earlier in this chapter). There cannot be any spaces between the LINK command and the /ALL parameter.

objname — The full or partial pathname of the object file you want to include.

For example, suppose you use the following command:

```
LINK/ALL MYFILE
```

If files MYFILE.A and MYFILE.B are in the current directory, the linker first searches MYFILE.ROOT, then MYFILE.B, and finally MYFILE.A.

Important: If you are linking object files with the root name *rootname*, make sure there are no other files in the same prefix as *rootname* with the same rootname and an alphabetic extension. For example, if you are linking MYFILE.ROOT and MYFILE.A, make sure there are no files named MYFILE.B or MYFILE.C in the same directory before linking.

If you do not include the /ALL qualifier, you must specify the full pathname (including filename extension, if any).

Note: The LinkEd LINK command does *not* automatically search library files in the library prefix (prefix 2). If any of the references in your program refer to labels in library files, you must use LinkEd LIBRARY commands to specify which libraries to search.

LIST

LIST ON|OFF

The LIST command controls the output of the link map.

ON|OFF LIST ON causes all subsequent segment names to be sent to standard output; LIST OFF suppresses output (unless an error occurs).

The link map is a listing of each segment, with its starting address and length, followed by a summary table showing the segment number, segment type (the KIND field in the segment header), the length of the segment, and the origin address (if any). This command is overridden by the L option in the shell's ASSEMBLE and COMPILE command lines.

LOADSELECT

LOADSELECT[/SCAN] *objname lseg*

This command causes the object segments that have the load segment name *lseg* in the object file specified by *objname* to be included in the output file. If the LOADSELECT command follows a SEGMENT command, the object segments are placed in the load segment defined by the SEGMENT command. If the LOADSELECT command does not follow a SEGMENT command, all object segments are placed in a load segment with a blank segment name (its name consists of ten space characters).

If the LOADSELECT command is not used, the linker ignores source-code load-segment names, such as those specified by the operand of an APW Assembler START directive.

/SCAN If you include the /SCAN parameter, all files with the root filename of the file specified by *objname* and .ROOT or alphabetic filename extensions are searched to make sure the most recently assembled version of each file segment is included (see the section "Partial Assemblies and Filename Conventions" earlier in this chapter). There cannot be any spaces between the SELECT command and the /SCAN parameter.

objname The full or partial pathname of the object file you want to search. If you do not include the /SCAN parameter in the command, you must use the complete filename, including the filename extension. If you do include the /SCAN parameter, do *not* include the filename extension in *objname*.

lseg The load-segment name of the object segments that you want to extract. To select all object segments with a blank load-segment name, use an asterisk (*) for *lseg*. In case-sensitive languages, segment names must be entered exactly as they appear in the source code. Segment names in case-insensitive languages must be entered as all uppercase characters.

For example, suppose your object file MYFILE.A contains the object segments PETER, PAUL, and MARY, and each of these object segments has either the load-segment name WHITE or the load-segment name BLACK, as follows:

0. Object-segment name: PETER
   Load-segment name: WHITE
1. Object-segment name: PAUL
   Load-segment name: BLACK
2. Object-segment name: MARY
   Load-segment name: WHITE

Furthermore, suppose you use the following commands in your LinkEd file:

```
SEGMENT GRAY
LOADSELECT MYFILE.A WHITE
```

This command extracts the object segments PETER and MARY, each of which has the load-segment name WHITE, and places them in the load segment GRAY. Note that the object segments with the load-segment name WHITE are not actually put in a load segment named WHITE unless you also use that load-segment name in the SEGMENT command, as in the following set of commands:

```
SEGMENT WHITE
LOADSELECT MYFILE.A WHITE
```

As an example of the use of the /SCAN parameter, suppose files MYFILE.ROOT, MYFILE.A, and MYFILE.B are in the current directory and you use the following command:

```
LOADSELECT/SCAN MYFILE WHITE
```

In this case, the linker first searches MYFILE.ROOT, then MYFILE.B, and finally MYFILE.A for object files that have the load-segment name WHITE.

OBJ

OBJ *val*

OBJ sets the value of the program counter (PC, a pseudo-address for the next line of code), so that subsequent lines of code will be linked as if the sequence had started at the address *val*.

val The value to which you want to set the program counter.

Unlike ORG, OBJ has no effect on the actual physical location at which the code is initially loaded; instead, OBJ is used when part of a program must be moved (to *val*) before execution.

Code produced in this way is not relocatable by the System Loader because references within it are to absolute addresses, starting at *val*. Such code may, however, be included in a segment that is relocatable. Use the OBJEND command to end the effect of the OBJ command.

> **Note:** This command is provided for those programs that have their own routines to move segments to specific absolute addresses. We strongly recommend that you not use this command, but take advantage of the capabilites of the Apple IIGS System Loader and Memory Manager instead. Programs that do their own loading and memory management are very unlikely to work successfully with any other Apple IIGS routines.

OBJEND

OBJEND

OBJEND resets the program counter to the current physical address in the file. The program counter and the physical address always match unless an OBJ command has been given.

ORG

ORG *val*

The ORG command sets the value of the program counter.

val The value at which you want to set the program counter.

The operation of ORG depends on where it is used, as follows:

- If the ORG command is used before any code segments in the current load segment have been processed, the load segment is given a fixed start location equal to *val*, and all code is linked for execution starting at the address *val*.
- If the ORG command is used after a code segment has been processed, LinkEd inserts zeros from the present location until the specified location is reached. If *val* is smaller than the current value of the program counter, the bytes between *val* and the program counter are deleted. If *val* is smaller than the program-counter value at the start of the code segment, an error is returned. An ORG command cannot be used within a load segment unless another ORG command was used at the beginning of the load segment.

Important: An ORG command in a LinkEd file does not override an ORG directive in the source code; rather, the linker processes all ORGs in the order in which it encounters them.

The parameter *val* can be specified as either a decimal number (for example, 126720) or a hexadecimal number (for example, $01EF00).

Note: We strongly recommend that you not use this command, but take advantage of the capabilites of the System Loader and Memory Manager instead. Programs that do their own loading and memory management are very unlikely to work successfully with any other Apple IIGS routines.

PRINTER

PRINTER ON|OFF

The PRINTER command controls output to the printer.

ON|OFF PRINTER ON sends the LinkEd source listing and symbol table to the printer; PRINTER OFF stops output. The default value is OFF.

This command overrides any output redirection used in the APW Shell's ASSEMBLE, COMPILE, or ALINK command line.

SEGMENT

SEGMENT[/DYNAMIC|/*kind*] *segname*

The SEGMENT command defines the beginning of a new load segment in the current load file, giving it the load-segment name *segname*. You can use the LINK, LOADSELECT, and SELECT commands to put any number of object segments in a load segment. Load-file segments may be loaded independently by the System Loader, as required.

Note: If the LINK or SELECT commands are used before any SEGMENT command, all object segments are placed in a load segment whose name consists of ten space characters. LinkEd ignores any load-segment assignments in your source code unless you use the LOADSELECT command.

Important: Some languages (such as C) are case sensitive; segment names for such a language must be entered in LinkEd commands exactly as they are listed in the source code, including case. For case-insensitive languages, segment names must be entered in uppercase.

/DYNAMIC The linker automatically flags segments as static. However, adding the /DYNAMIC qualifier to the SEGMENT command makes the segment dynamic. There cannot be any spaces between the SEGMENT command and the /DYNAMIC parameter. You cannot use both the /DYNAMIC and */kind* qualifiers in the same SEGMENT command.

Note: Dynamic segments are supported so that you can write programs that make highly efficient use of memory. Keep in mind, however, that any code that is needed at all times (or frequently) by the program cannot be dynamic. See the following note on load segments.

/kind The Apple IIGS object module format defines several segment types in addition to static, dynamic, code, and data. The segment type is specified in the KIND field of the segment header. You can use the */kind* qualifier to specify a special segment type for a load segment. There cannot be any spaces between the SEGMENT command and the */kind* parameter. Precede the number with a dollar sign ($) to indicate a hexadecimal number.

The linker presently generates files that conform to OMF Version 1.0, so you must enter a 1-byte Version 1.0 KIND in this parameter. In OMF Version 2.0, the KIND field is 2 bytes long. OMF Version 1.0 and 2.0 KIND fields are described in Chapter 7. You can convert the load file to OMF 2.0 by using the Compact utility, as described in Chapter 3. The Version 1.0 KIND field does not define the No Special Memory and Reload segment types; however, the Compact utility adds the appropriate bits to the Version 2.0 KIND field if you set the Version 1.0 segment type as follows:

KIND	Segment Type
$1E	cannot be loaded in special memory
$1F	reload segment

You cannot use both the /DYNAMIC and */kind* qualifiers in the same SEGMENT command.

segname The name of the load segment into which you want to link object segments. Segment names are case sensitive.

Examples of SEGMENT commands are shown in the section "Sample LinkEd Files" at the end of this chapter.

The end of a load segment is marked by

- another SEGMENT command
- the end of the source file

Load Segments: Each load file has at least one segment—the main segment—which, along with all other static segments, is loaded first by the System Loader and is never removed from memory. It is usually the first segment in the file. Segments may directly access data in themselves and in any static segment, but they cannot directly access data in dynamic segments.

If a segment calls a subroutine in a dynamic segment, and that segment is not in memory, then the System Loader loads that segment. If there is not enough memory to hold the segment, the Memory Manager attempts to free memory by unloading dynamic segments that an application has made purgeable (if this attempt fails, a system error is returned). Note that this means that the values of variables in dynamic segments may not be preserved between calls. Intersegment calls must be made with a long subroutine jump (`JSL`), which uses a 3-byte address, rather than the "regular" subroutine jump (`JSR`), which uses a 2-byte address; because the loader may put a segment into any bank of memory, the `JSR` instruction would be useless because it can access only the current bank. For more information on segment loading and dynamic segment referencing, see the *Apple IIGS ProDOS 16 Reference* manual.

Both static and dynamic segments are automatically considered by the linker to be relocatable, unless they contain an `ORG` assembler directive or are preceded by an `ORG` LinkEd command.

SELECT

`SELECT[/SCAN]` *objname* `(`*seg1*`[,`*seg2*`[,...]])`

This command causes the named segment(s) (*seg1, seg2,...*) from the object file specified by *objname* to be included in the output file. The segments are added in the order listed in the command. If the `SELECT` command follows a `SEGMENT` command, the object segments specified in *objname* are placed in the load segment defined by the `SEGMENT` command. If the `SELECT` command does not follow a `SEGMENT` command, all object segments are placed in a load segment whose name consists of ten space characters. The `SELECT` command ignores source-code load-segment names, such as those specified by the operand of an APW Assembler `START` directive.

Use the `LINK` command to link all the segments in a file.

`/SCAN` If you include the `/SCAN` parameter, all files with the root filename of the file specified by *objname* and `.ROOT` or alphabetic filename extensions are searched to make sure the most recently assembled version of each file segment is included (see the section "Partial Assemblies and Filename Conventions" in this chapter). There cannot be any spaces between the `SELECT` command and the `/SCAN` parameter.

objname The full or partial pathname of the object file you want to search. If you do not include the `/SCAN` parameter in the command, you must use the complete filename, including the filename extension. If you do include the `/SCAN` parameter, do *not* include the filename extension in *objname*.

seg1, seg2, . . . The names of the object segments that you want to extract. To extract all the object segments from an object file, use the `LINK` command. In case-sensitive languages, segment names must be entered exactly as they appear in the source code. Segment names in case-insensitive languages must be entered as all uppercase characters.

For example, suppose you use the following command:

```
SELECT/SCAN MYFILE (main,globals)
```

If files MYFILE.ROOT, MYFILE.A, and MYFILE.B are in the current directory, the linker first searches MYFILE.ROOT, then MYFILE.B, and finally MYFILE.A. It extracts only the most recent versions of the object segments main and globals from these files.

SOURCE

SOURCE ON|OFF

This command controls the output of LinkEd source code.

ON|OFF SOURCE ON causes all subsequent lines of LinkEd source code to be sent to standard output. SOURCE OFF suppresses output, unless an error is encountered.

This command is overridden by the L option in the shell's ASSEMBLE and COMPILE command lines and by the LIST assembler directive.

SYMBOL

SYMBOL ON|OFF

The SYMBOL command controls output of the symbol table.

ON|OFF SYMBOL ON causes the symbol table to be sent to standard output; SYMBOL OFF suppresses output.

The symbol table is an alphabetical listing of all symbolic references (labels). All segments share the same symbol table. This command is overridden by the S option in the shell's ASSEMBLE and COMPILE command lines.

Sample LinkEd Files

The listings below are all valid LinkEd files. Here all commands are written in uppercase to follow the convention used in this book. Note, however, that segment names for languages (such as C) that are case sensitive must be entered exactly as they are listed in the source code. For case-insensitive languages, segment names must be entered in uppercase.

1. The following routine opens an output file called OUTFILE, includes all files within the current subdirectory that have the root filename MYFILE, and performs a library search on the current system library. It is equivalent to calling the linker with the APW Shell command LINK MYFILE KEEP=OUTFILE, except that any source-code load-segment names are ignored.

```
KEEP OUTFILE
LINK/ALL MYFILE
LIBRARY *
```

2. This routine creates an object file with three segments, one of which is dynamic. The first load segment is created by the LINK statement that precedes the first SEGMENT statement and has a load segment name consisting of ten space characters.

The second static load segment is created by the first SEGMENT command. The dynamic load segment is created by the SEGMENT/DYNAMIC command.

```
KEEP MYPROG
LINK/ALL MAINSUBS
LIBRARY *
SEGMENT SEG1
   LINK/ALL SUBS1
   LIBRARY *
SEGMENT/DYNAMIC SEG2
   LINK/ALL SUBS2
   LIBRARY *
```

3. In this routine, both the library file MYFILE2 and the system libraries are searched for needed subroutines.

```
KEEP MYPROG
LINK MYFILE
LIBRARY MYFILE2
LIBRARY *
```

4. In this example we assume we have written a program in two parts, one part in C called DEMO.C, and one in assembly language called DEMO.ASM. The object file START.ROOT, which is located in the library prefix (prefix 2), must be linked first, as it contains initialization routines that we use for all our C programs. We want to include routines from the standard C libraries in the library prefix, but we have created two additional library files, called NEWLIB1 and NEWLIB2, that modify some of the standard library routines. If we link those libraries *before* the standard C libraries, the linker will have already resolved any references to the routines in NEWLIB1 and NEWLIB2 and will ignore any routines with the same names in the standard C libraries. NEWLIB1 and NEWLIB2 are in the current prefix.

   Each of the object segments in the C libraries has one of the following load-segment names in its segment header:

```
main
~globals
~arrays
        (all space characters)
```

   In addition, the file START.ROOT contains object segments with the load-segment name main, and the files DEMO.C.ROOT, DEMO.ASM.ROOT, and DEMO.ASM.A contain object segments with the load-segment names LSeg1, LSeg2, and LSeg3.

   Before looking at a LinkEd file to link this program, first consider the effect of using the following command:

```
LINK 2/START DEMO.C DEMO.ASM NEWLIB1 NEWLIB2 KEEP=SAMPLE
```

   When this LINK command is executed, the file START.ROOT is linked first, followed by DEMO.C.ROOT, DEMO.ASM.ROOT, and DEMO.ASM.A. Next, the library files NEWLIB1 and NEWLIB2 are searched for any unresolved references. Finally, if any unresolved references remain, the library files in prefix 2 are

searched. The resulting load file (named SAMPLE) contains the following segments (in the sequence in which segments with these load-segment names were first found in the object files). All these segments are static.

```
main
        (all space characters)
~globals
~arrays
LSeg1
LSeg2
LSeg3
```

Now, consider what happens when we link the program with the following LinkEd file:

```
   KEEP SAMPLE
* The following command starts the first load segment, named LSEG1.
*  This is a static load segment.
   SEGMENT LSEG1
* The following commands extract object segments with the load-
*  segment names main and LSeg1 from the object files:
      LOADSELECT/SCAN 2/START main
      LOADSELECT/SCAN DEMO.C LSeg1
      LOADSELECT/SCAN DEMO.ASM LSeg1
* The following commands extract object segments with the load-
*  segment name main from the library files:
      LIBRARY/LOADSELECT NEWLIB1 main
      LIBRARY/LOADSELECT NEWLIB2 main
      LIBRARY/LOADSELECT * main
* The following command extracts object segments with blank load-
*  segment names from the standard library files:
      LIBRARY/LOADSELECT * *
* The following command starts the second load segment, named LSEG2.
*  This is a dynamic load segment.
   SEGMENT/DYNAMIC LSEG2
* The following commands extract object segments with the load-
*  segment name LSeg2 from the object files:
      LOADSELECT/SCAN DEMO.C LSeg2
      LOADSELECT/SCAN DEMO.ASM LSeg2
* The following command starts the third load segment, named LSEG3.
*  This is a dynamic load segment.
   SEGMENT/DYNAMIC LSEG3
* The following commands extract object segments with the load-
*  segment name LSeg3 from the object files:
      LOADSELECT/SCAN DEMO.C LSeg3
      LOADSELECT/SCAN DEMO.ASM LSeg3
* The following command starts the fourth load segment, named GLOBALS.
*  This is a static load segment.
   SEGMENT GLOBALS
* The following commands extract object segments with the load-
*  segment name ~globals from the object files.  These object segments
*  contain global variables called with short addresses in C routines:
      LOADSELECT/SCAN 2/START ~globals
      LOADSELECT/SCAN DEMO.C ~globals
      LIBRARY/LOADSELECT NEWLIB1 ~globals
      LIBRARY/LOADSELECT NEWLIB2 ~globals
```

```
      LIBRARY/LOADSELECT * ~globals
* The following command starts the fifth load segment, named ARRAYS.
*  This is a static load segment.
   SEGMENT ARRAYS
* The following commands extract object segments with the load-
*  segment name ~arrays from the object files.  These object segments
*  contain global arrays called with long addresses in C routines:
      LOADSELECT/SCAN 2/START ~arrays
      LOADSELECT/SCAN DEMO.C ~arrays
      LIBRARY/LOADSELECT NEWLIB1 ~arrays
      LIBRARY/LOADSELECT NEWLIB2 ~arrays
      LIBRARY/LOADSELECT * ~arrays
```

When this LinkEd routine is executed, the object and library files are searched in the sequence specified by the commands in the file, as indicated by the comments in the file. The file `DEMO.C`, for example, is opened and searched five separate times: first for object segments with the load-segment name `LSeg1`, then for object segments with the load-segment name `LSeg2`, then `LSeg3`, `~globals`, and finally `~arrays`. The final load file (also named `SAMPLE`) includes the following segments:

```
LSEG1
LSEG2
LSEG3
GLOBALS
ARRAYS
```

Segments `LSEG2` and `LSEG3` are dynamic. Unlike the segments created by the standard linker, these segments are placed in the load segment in the sequence you specified. The object segments with load-segment names `main` and blank (all space characters) are incorporated into segment `LSEG1`.

In contrast to the standard linker, LinkEd files let you control the order in which object and library files are searched for each object segment and the sequence in which load segments are placed in the load file. LinkEd files let you specify whether a segment is static or dynamic, regardless of any segment-type specifications in the source file. LinkEd lets you extract only the object segments you want, so you can exclude segments you don't need for a particular application. It lets you specify object segments by object-segment name or by load-segment name, whether those segments are in object files or load files.

The price you pay for this additional control and flexibility is that you must specify every file to be searched and every segment to be included. You must be familiar with the contents, not only of the source files you write, but also of any other object files and library files you wish to link. If you need the power provided by LinkEd, however, you will find the time spent in learning how to use it and in writing the command files well worth the effort.

Part III

Inside the Apple IIGS Programmer's Workshop

Chapter 6

Adding a Program to APW

This chapter describes how to add a utility program or compiler to the Apple IIGS Programmer's Workshop. None of the information in this chapter is essential for writing programs that are independent of APW.

Note that when you add a utility or language to APW, you should update the APW command table to include it. APW will execute a program that is not listed in the command table, but it does not automatically search the utility or language prefix for the program if it is not listed in the command table. The command table is described in the section "Command Types and the Command Table" in Chapter 3, and a list of language numbers currently assigned is given in Appendix B.

To get started as an Apple developer, write to

Developer Relations
Mail Stop 27 S
Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014

Types of APW Programs

ProDOS 16 supports two principal kinds of executable load files: ProDOS 16 file types $B3 and $B5. These two file types have the following characteristics:

- Programs of file type $B3 take over complete control of the computer; they do not operate under a shell program. APW itself is an example of such a program. When a program of file type $B3 is called, the calling program executes a ProDOS 16 `QUIT` call, shutting itself down. When the called program finishes and executes a `QUIT` call, ProDOS 16 reboots the calling program (assuming the calling program instructed ProDOS 16 to do so). The ProDOS 16 `QUIT` call is described in the *Apple IIGS ProDOS 16 Reference*.
- Programs of file type $B5 run under a shell program; they do not remove the shell from memory. The shell calls a program of file type $B5 in full native mode via a `JSL` instruction. When the program terminates, it returns control to the shell via an `RTL` instruction (or, if the shell supports it, through a ProDOS 16 `QUIT` call).

APW utility programs are programs of file type $B5 designed to be run under the APW Shell program. They perform operations too complex to be performed by the shell itself, but appear to the user to be shell commands. APW compilers and assemblers are also programs of file type $B5, but they make use of special APW Shell calls (described in Chapter 8) to pass parameters to and from the shell, and they are distinguished from

utilities in the command table (see the section "Command Types and the Command Table" in Chapter 3). Since the requirements for compilers and assemblers are different from those for utility programs, they are discussed separately in this chapter.

You can write a program of file type $B3 and use it with APW; APW launches any executable load file it finds on disk when you type in the program's pathname. Since APW quits and ProDOS 16 clears the desktop when a type $B3 program is called, however, there are no special requirements for the program (other than those required by the Apple IIGS system in general), and so these programs are not discussed in this chapter.

> **Note:** Any $B5 file that runs under the APW Shell can be made into a $B3 file, provided that it makes no calls to the APW Shell and that it terminates with a ProDOS 16 `QUIT` call. Use the Shell's `FILETYPE` command to change the file type of a $B5 file to $B3.

See the *Programmer's Guide to the Apple IIGS* for guidance in writing an event-driven program for the Apple IIGS computer.

> **Important:** Before writing any programs to run under the APW Shell, you should become familiar with the shell calls described in Chapter 8. These calls help you to implement a variety of APW features, such as wildcard expansion and early termination of the program in response to an Apple-Period (⌘-.) key press.

APW Utilities

APW utilities are applications designed to run under the APW Shell. They must be ProDOS 16 file type $B5. By following the guidelines described in this section, you can write a utility that can be executed from the APW Shell with APW remaining resident in memory.

> **Note:** Although many of the rules listed in this section for APW utilities apply to *any* utility written to run under *any* shell, the purpose of this section is to describe how to add a utility to APW only. To write a utility to run under another shell, you will have to know the specific requirements of that shell.
>
> APW Exec files can be installed as utility programs by placing the file in the utility subdirectory (prefix 6) and adding the name of the file to the command table. This section describes $B5 load files only, *not* Exec files. Exec files are discussed in the section "Exec Files" in Chapter 3.

When you enter an APW command, the APW Shell looks for the command name in the command table (see the section "Command Types and the Command Table" in Chapter 3). If the command is listed in the command table as a utility, the shell loads it from the utility prefix (prefix 6); if the command is not in the command table, then the shell looks for a file with that name in the current prefix. In either case, the shell strips any I/O redirection information from the command line and places the command line (together with the shell identifier string) in a buffer in memory. The shell then places the address of the command-line buffer in the X and Y registers. The shell requests a user ID for the program from the User ID Manager and places this ID in the accumulator.

If the utility program does not have a direct-page/stack segment, then when the APW Shell calls the program, it provides a 1024-byte memory block in bank 00 for the utility to use

for its direct page and stack. The shell places the address of the start of the memory block in the direct-page (D) register and sets the stack pointer (S register) to point to the last byte of the block. If it finds a direct-page/stack segment, the shell sets the D register to point to its first byte and the stack pointer to its last.

Requirements

Any utility must obey the following rules in order to execute successfully under the APW Shell.

Warning: If a program with ProDOS 16 file type $B5 does not obey the following rules, you must quit APW before calling it. Executing such a program from the APW Shell can cause the system to crash. In fact, such a program should not be given the file type $B5.

- The utility must be designed to be called in full native mode via a `JSL` instruction.
- As soon as the utility is called, it should check the X and Y registers for the address of the command-line buffer, which contains the following information:

    1. An 8-byte ASCII string containing the APW Shell identifier string `BYTEWRKS`. The utility should check this identifier to make sure that it has been launched by the APW Shell, so that the environment it needs is in place. If the shell identifier is not correct, the shell load file should write an error message to standard error output (normally the screen) and exit with an `RTL` instruction or a ProDOS 16 `QUIT` call.
    2. A null-terminated ASCII string containing the input line for the utility. The APW Shell strips any I/O redirection or pipeline commands from the input line, since those commands are intended for the shell itself, but passes on the command name and all input parameters intended for the utility.

- All input must come from standard input, which provides a sequential character stream. Standard input is discussed in the section "Redirecting Input and Output" in Chapter 3. You can use Apple IIGS Text Tool Set calls to read the next input character. Tool calls are described in the *Apple IIGS Toolbox Reference* manual.

Important: Your utility should not read the keyboard directly, because in that case the shell input redirection command would not work, contrary to the expectations of the user. For the same reason, your utility should not initialize or reset the Text Tool Set.

- All output must go to standard output, which appears to the program as a sequential, write-only ASCII output device. Standard output is discussed in the section "Redirecting Input and Output" in Chapter 3. You can use Apple IIGS Text Tool Set calls to send output to standard output.
- The utility must handle its own errors. You can use standard output or standard error output as you prefer. The utility should place an error-condition code in the accumulator before returning control to the shell. If no error has occurred, the error code should be $0000; otherwise, the code should be $FFFF. When the program returns control to an Exec file, the error code is placed in the `{Status}` variable. If `{Exit}` is non-null, the Exec file terminates. Exec files are discussed in the section "Exec Files" in Chapter 3.

- The utility must use the Memory Manager to request memory; since several programs can be resident in memory at one time, there is no way to predict what areas of memory will be free for the utility to use.
- The utility should use the APW Shell calls described in Chapter 8 whenever possible to perform a necessary operation. For example, use the `Execute` call to pass a command on to the shell command interpreter rather than duplicating the function in your program.
- If appropriate, the utility should use the APW Shell `STOP` call described in Chapter 8 to detect a request for an early termination of the program. Note that it should call `STOP` frequently in order for this function to be effective.

  Important: If your utility uses APW Shell calls, it will not run if called by ProDOS 16 or by another shell.
- If the utility launches another program, it must request a User ID from the User ID Manager. The utility is then responsible for intercepting ProDOS 16 `QUIT` calls and system resets, so that it can remove from memory all memory buffers with that user ID before passing control back to the APW Shell.
- A utility should use the following procedure to quit:
  1. If the utility has requested any User IDs, it must release all memory buffers with those User IDs.
  2. The utility must place an error code in the accumulator. If no error occurred, the error code should be $0000; otherwise, the code should be $FFFF.
  3. The utility should execute an `RTL` instruction or a ProDOS 16 `QUIT` call. If the utility is not restartable, the APW Shell releases all memory buffers associated with it.

  Important: Do not add any utility to APW that writes to or modifies directory files, as such a utility would interfere with any file servers added in the future and may be incompatible with new operating systems.

Conventions

The following features are not required for an APW utility to work, but they are recommended in order to provide a consistent appearance and manner of operation of all utilities.

- Utilities should take any input as command-line parameters, rather than prompting for input, although the utility should prompt for any required parameter that is omitted by the user. There are two kinds of parameters: pathnames and options. Options begin with a minus sign (–) to distinguish them from pathnames. Each option is a single letter or a single word, but some options may require additional parameters, which are separated from the option name with a space. If more than one parameter is required following the option name, the usual separators between them are commas and equal signs; for example

  ```
  COMMAND -DEFINE TURN='ON' -PAGE 84,110
  ```

  Options and pathnames may appear in any order. All of the options apply to the processing of all of the files, regardless of the order in which the options and pathnames appear on the command line.

Notice that a somewhat different convention is used for options for APW compilers and linkers. See the description of the ASML command in Chapter 3 for a discussion of compiler options.

- If your utility can generate more output than can fit on a single screen, the user will expect to be able to pause the output by pressing any character key on the keyboard. To implement this feature, call the following procedure frequently; for example, after every line of output. This procedure returns a 1 in the accumulator if Apple-Period was pressed; therefore, for languages that pass parameters in the accumulator, you can also use this procedure to replace the Shell's STOP call.

```
; STOP_PAUSE:  Handles key press pause and resume. Returns TRUE for
; open-Apple/period, FALSE otherwise.
; To use, assemble, and add object file name to linker command line.
;
; Example of use from C:  if(STOP_PAUSE()) exit(0);

;
PAUSE    START   main
         LDA     #$0000     preset default result in all 16 bits
;
         LONGA   OFF        change to 8 bit mode
         LONGI   OFF
         SEP     #$30
;
         PHB                save data bank on stack
         PHA                set data bank to 0
         PLB                so we can read the key-board strobe and data
;
         BIT     $C000      test strobe
         BPL     done       done if strobe not set
         BIT     $C025      test for open apple modifier
         BPL     key
         LDA     #$AE       test for period (high bit still set)
         CMP     $C000
         BEQ     stopset
;
         LDA     #$00       restore default result
key      BIT     $C010      reset strobe
loop     BIT     $C000      test strobe
         BPL     loop       until next key press
;
         BIT     $C025      test for open apple modifier
         BPL     out
         LDA     #$AE       test for period (high bit still set)
         CMP     $C000
         BEQ     stopset
;
         LDA     #$00       restore default result
         BRA     out
stopset  LDA     #$01       set return value for stop
out      BIT     $C010      reset strobe
done     PLB                restore data bank
;
         REP     #$30       back to 16 bit mode
         LONGI   ON
         LONGA   ON
;
```

```
RTL
END
```

- When you add a utility program to APW, you should provide a help file to go with it. Help files are ASCII text files (APW language type PRODOS) that have the same name as the command and that are kept in the /APW/UTILTIES/HELP/ subdirectory. To see an example of a help file for an APW utility, enter the following command:

  ```
  HELP MAKELIB
  ```

  Notice that the user cannot scroll through a help file; the text should fit on one screen.

- If you wish, you can make your utility program restartable, so that it does not have to be reloaded from disk each time it is run. For a program to be restartable, it must reinitialize all variables and arrays each time it starts. OMF Version 2.0 provides the following special segment types that support restartable programs: *initialization segments*, which are reloaded from disk and executed each time a program is restarted from memory; and *reload segments*, which are reloaded from disk each time a program is restarted.

  The APW Linker creates OMF Version 1.0 files. You can use the Compact utility to convert an OMF Version 1.0 load file to OMF Version 2.0. See the description of the COMPACT command in Chapter 3 and the description of the SEGMENT command in Chapter 5 for ways to create reload and initialization segments. Versions 1.0 and 2.0 of the OMF are defined in Chapter 7.

  To indicate to the APW Shell that the program is restartable, put an asterisk (*) in the command table in front of the command type (the U). If you precede the command type with an asterisk, the shell assumes that the program can be restarted and does not remove static segments from memory as long as that memory is not needed for other purposes.

Compilers and Assemblers

Compilers, assemblers, and interpreters are implemented in nearly identical ways in APW. In this section, the term *compiler* is used generically to include compilers, assemblers, and interpreters, unless an explicit distinction is made.

Source File Format

Your compiler must be capable of accepting files that conform to the Apple IIGS text-file format, as specified in Chapter 7. In this format, lines are separated by carriage return characters ($0D). The form-feed character ($0C) should be accepted, and used to generate a form feed in printed output. Your compiler should handle tabs as discussed in Chapter 7.

All lines in APW source files are assumed to be no more than 255 characters long.

Identifying the Language Type

Each language used by the Apple IIGS Programmer's Workshop has a unique language number. Language numbers are discussed in the section "Command Types and the Command Table" in Chapter 3, and a list of the language numbers currently assigned is given in Appendix B. If you are a certified Apple developer and you need a new language number for your compiler, write to

Developer Technical Support
Mail Stop 27 T
Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014

Each source file must have one of these language numbers as the first byte of the `aux_type` field in the file entry of the subdirectory. The APW Editor automatically includes this language number when it writes a file to disk; if the program is written with a different editor, the user must use the APW Shell's `CHANGE` command to assign the appropriate language type to the file. The format of directory entries is described in the *Apple IIGSProDOS 16 Reference* manual.

Your compiler should include a command that corresponds to the APW Assembler `APPEND` directive, which transfers control from the file being processed to a new file. When this command is used, your compiler must check the language type of the new file; if the language type does not match that of your compiler, the compiler must close the object file it is generating and transfer control back to the shell by executing a `SET_LINFO` call (described in Chapter 8).

Entry and Exit

Compilers and assemblers that operate under APW should have ProDOS 16 file type $B5. When a user enters the `COMPILE` command (or one of its aliases), the shell checks the language type of the source file and uses a `JSL` instruction to pass control to the appropriate compiler. The first thing the compiler should do is to execute a `GET_LINFO` call (described in Chapter 8) to read the input parameters. Upon completion, the compiler should execute a `SET_LINFO` call and return control to the APW Shell via an `RTL` or a ProDOS 16 `QUIT` call. The system is in full native mode when it calls the compiler, and it should be in full native mode when control is returned to the shell.

The compiler should use the APW Shell `STOP` call described in Chapter 8 to detect a request for an early termination of the program. If it receives such a request, the compiler should treat it like a fatal error (see the following discussion of the `SET_LINFO` call).

The compiler is responsible for reading and using the parameters passed to it via the `GET_LINFO` call, updating any values that have changes, and returning them via the `SET_LINFO` call when the compile is complete. These parameters are all described in Chapter 8. In order to make your compiler fully consistent with other APW compilers, you should keep the following points about these parameters in mind:

- If the compile completes with a nonfatal error, the compiler should return the error level in the `merrf` field of the `SET_LINFO` call. If `merrf` is greater than `merr`, the shell stops processing the program, even if `CMPL`, `CMPLG`, or an equivalent command was used. Use the following error levels for nonfatal errors:

Error Level	Meaning
$02	Warning. An anomaly has been found in the code. It may execute successfully.
$04	Error. The compiler may be able to correct this error. Examples include misspellings or omitted keywords.
$08	Error. The compiler cannot correct the error but knows how much space to leave. This error level is usually restricted to assemblers.
$10	Error. The compiler cannot correct the error, but only the segment containing the error is affected. An example would be an undeclared local variable.
$20	Syntax error. The entire result of the compile is suspect. This error would occur, for example, when a syntax checker had to skip symbols in an attempt to resynchronize with the code stream. In some languages, such as FORTRAN, the syntax checker can resynchronize with the beginning of the next line, in which case this type of syntax error should never occur. In free-format languages such as Pascal, on the other hand, an entire subroutine could be discarded before the compiler resynchronizes; in this case, a syntax error should be flagged.

- If the compile terminates prematurely due to a fatal error, the compiler should return an $FF in the `merrf` field of the `SET_LINFO` call.
- All memory buffers pointed to by parameters in the `SET_LINFO` call should be in static segments loaded when your program was launched. The APW Shell does not unload your program's static segments until after it has processed the `SET_LINFO` call.
- Your compiler can read any special parameters passed to it in the buffer pointed to by the `istring` field of the `GET_LINFO` call. There is no need to pass those parameters back to the shell when your compiler exits via a `SET_LINFO` call.
- If the compile terminates prematurely and the `+E` flag is set, the compiler should place the pathname of the source file in which the error occurred into a buffer and set the `sfile` parameter of the `SET_LINFO` call to point to that buffer.
- If the compile terminates prematurely and the `+E` flag is set, the compiler should place the text of the error message into a buffer and set the `parms` parameter of the `SET_LINFO` call to point to that buffer.
- If the compile terminates prematurely and the `+E` flag is set, the compiler should place in the `org` field of the `SET_LINFO` call the displacement into the source file of the last line processed. When the APW Shell receives control, it calls the APW Editor, which displays the source file indicated by the `sfile` parameter; the line containing the error as indicated by the displacement in the `org` field is placed at the fifth line on the screen and the error message pointed to by the `parms` parameter is displayed at the bottom of the screen.

- The least significant bit (bit 0) of the operations-flags (`lops`) field in the `GET_LINFO` call is always set (1) when a compiler is called; this bit indicates that a compile is to be performed. If the next bit (bit 1) is set, it indicates that a link should be performed after a successful compile; if bit 2 is also set, it indicates that the finished program is to be executed immediately after the link.
- If the compile completes normally, the compiler should clear the least significant bit of the `lops` field of the `SET_LINFO` call.
- If a compile completes with `merrf>merr`, or terminates prematurely with a fatal error, then no further processing is done regardless of the setting of the operations flags.
- If the compile stops because a file was appended that had a language type different from the language type of the compiler, the compiler should *not* clear the least significant bit of the `lops` field of the `SET_LINFO` call. This indicates to the shell that the compile is not complete so that it can then call the compiler appropriate to the new file.
- The `kflag` parameter of the `GET_LINFO` call is used by the compiler to determine the names and number of output files to generate. The `kflag` parameter is discussed in detail in the section "Ouput Files" in this chapter.
- If any segment names are listed in the buffer pointed to by the `parms` parameter of the `GET_LINFO` call, a partial compile is to be performed. Partial compiles are discussed in detail in the section "Partial Compiles" later in this chapter.
- There is a set of standard options that are passed by the `mflags` and `pflags` parameters of the `GET_LINFO` call. The purpose of each of these options is described in the section on the `ASML` command in Chapter 3 and in the section "Compiling (or Assembling) and Linking a Program" in Chapter 2. If your compiler does not support any of these options, or responds in a manner differently from that described in this manual, your manual should clearly state so.

If you wish, you can make your compiler restartable so that it does not have to be reloaded from disk each time it is run. For a program to be restartable, it must reinitialize all variables and arrays each time it starts. To indicate to the APW Shell that the program is restartable, put an asterisk (`*`) in the command table in front of the command type (the `L`). If you precede the command type with an asterisk, the shell assumes that the program can be restarted and does not remove static segments from memory as long as that memory is not needed for other purposes.

Command Precedence

If your compiler includes source-file commands that control functions that can also be controlled from the command line, the command-line input should take precedence. For example, if the source code includes a command that suppresses a listing of the source file, but the user requests a listing by specifying `+L` on the command line, then a listing should be generated.

Output Files

Every compiler under APW must be capable of producing one or more object files that conform to APW object module format (described in Chapter 7). These files are then processed by the APW Linker to produce an executable load file.

Both object files and load files are segmented, but a load segment can contain more than one object segment. In assembly language, the object-segment name is in the label field of a START or DATA directive, and the name of the load segment to which that object segment is to be assigned is specified in the operand field of the directive.

In order to make it easier for users to link together object files made with your compiler, you can assign one default load-segment name (such as a string of spaces) to all code segments and another (~globals) to all global variables. You might want to place all global variables that are called with short addresses in one segment (~globals) and all global variables called with long addresses in another segment (~arrays), as is done by APW C (notice that these segment names are all lowercase characters). In order to aid users in linking together routines written in different languages, your manual should state clearly what segment-naming conventions you have adopted and how to use these segment names to gain access to global variables.

The APW Linker normally assigns all object segments with the same load-segment name to the same load segment. The user has the option of using a LinkEd file to instruct the linker to place any object segment in any load segment.

See the section "Object Module Format" in Chapter 7 for a description of segments, segment types, and segment headers.

When the CMPL, CMPLG, or COMPILE command (or an alias) is executed, the user can specify the name of the output file with the KEEP parameter.

> **Important:** Normally in APW, parameters listed on the command line take precedence over those set in the source file. Therefore, your compiler should use the name given in the KEEP parameter in preference to any output filename given in the source file. If for some reason your compiler does not support the KEEP parameter, or an output filename in the source file takes precedence, your manual should clearly explain that this is the case.

The shell checks the directory for filenames that match the KEEP filename, excluding extensions, and sets the kflag parameter in the GET_LINFO call accordingly. The shell places the object filename in a buffer and puts the address of the buffer in the dfile parameter of the GET_LINFO call. The kflag parameter can be equal to 0, 1, 2, or 3, as follows.

> **Note:** An object filename assigned by the shell from a KeepName shell variable is passed to the compiler in exactly the same way as one specified with the KEEP parameter. There is no way for your compiler to tell whether the name was specified with a KEEP parameter or with a KeepName variable.

- If kflag = 0, no KEEP parameter was used in the command line. If a KEEP directive (or the equivalent) was used in the source code, the compiler must perform

its own check for filenames that match the KEEP filename. If no KEEP directive was used, do not save the output.

- If kflag = 1, no output files have been previously generated with this filename. The compiler should place the first segment to be executed in a file with the filename specified with the KEEP parameter and with the extension .ROOT. For example, if kflag=1 and if the COMPILE command included the parameter KEEP=MYFILE, then the compiler should place the first segment to be executed in a file named MYFILE.ROOT. If there are additional segments in the source file, they may be put in a file named MYFILE.A.
- If kflag = 2, a file with the KEEP filename and the extension .ROOT already exists. In this case, the compiler should start by creating a file with the extension .A. If the main program segment was written in assembly language and a subroutine was written in C, for example, then the assembler would create the .ROOT file, and the C compiler would create the .A file.
- If kflag = 3, at least two files with the KEEP filename already exist: one with the extension .ROOT and one with the extension .A. In this case, files with other alphabetic extensions might also exist; these files are created by partial compiles, as discussed in the following section. The compiler should start by searching the directory of the KEEP filename to determine the highest alphabetic suffix on the disk, and then use the next higher suffix. For example, if the files MYFILE.ROOT, MYFILE.A, and MYFILE.B all exist, the compiler should start with the filename MYFILE.C. Multiple output files can be created by a multilanguage compile (the first language creates the .ROOT and .A files, the second language the .B file, and so on) or by partial assemblies.

The paradigm followed by the APW Assembler is to first look for the .ROOT file, then the .A file, then the .B file, and so on. The search is terminated as soon as one file in the sequence is not found. Therefore, if the files MYFILE.A, MYFILE.B, and MYFILE.D were in the subdirectory, but MYFILE.C was not, the assembler would never find MYFILE.D. The next file created by the assembler, then, would be MYFILE.C.

Notice that in this example, the linker would start the link with the file MYFILE.D. Because MYFILE.C was the last file created, it is unlikely that this is what the user expected.

Your compiler must follow certain conventions when writing names to object files:

- If the source language is case-insensitive, always use uppercase letters in identifiers. If the source language is case-sensitive, retain the case of all characters. The linker retains the case of labels.
- For fixed-length names (as specified by the LABLEN field in the OMF segment header), use space characters ($20) to pad names to the required length.

Partial Compiles

The Apple IIGS object module format, the System Loader, and the Memory Manager are all designed to support program code that is organized in segments that can be loaded independently. If your compiler is going to work well in the Apple IIGS Programmer's Workshop environment, it should be capable of creating segments that can be linked to

segments output by other compilers and also of using segments created by other compilers. The use of segmented code provides two additional benefits: first, it facilitates the use of libraries, since the entire library file need not be linked to each program, and second, it allows for partial compiles.

In a partial compile, a list of segments to be compiled is passed to the compiler by the GET_LINFO call; the compiler searches through the source code for the named segments and compiles them. Other segments need not be compiled. Any segments compiled (other than the first segment to be executed when the program is run) are placed in a file with the next available alphabetic suffix, as discussed in the previous section, "Output Files." If one of the segments compiled is the first code segment that will be executed when the program is run, the compiler deletes the old .ROOT file and creates a new one.

When the linker links the program, it uses the following procedure:

1. It starts with the .ROOT file, and links that segment.
2. It looks for a .A file. If it finds one, the linker looks for a .B file, and so on.
3. It links the file with the highest alphabetic suffix it has found.
4. It works its way back through the alphabet to the .A file, ignoring any segments with names identical to those it has already found, and linking the rest.

For example, suppose you have compiled a program that has four segments, SEG1, SEG2, SEG3, and SEG4. SEG1 is the first segment that will be executed when the program is run. The compiler places SEG1 in the file MYPROG.ROOT, and the remaining three segments in the file MYPROG.A. Now suppose that, in testing the program, you have to make changes to segments SEG2 and SEG4, so you perform a partial compile. In this case, the compiler places segments SEG2 and SEG4 in the file MYPROG.B. Finally, to fix the one remaining bug in the program, you do another partial compile on SEG2. The compiler places the latest version of SEG2 in the file MYPROG.C. Now when you link the program, the linker operates as follows:

1. It finds MYPROG.ROOT and links it.
2. It finds MYPROG.A, then finds MYPROG.B, and then MYPROG.C. It does not find MYPROG.D, so it links MYPROG.C.
3. It searches MYPROG.B and finds that it has already linked SEG2, so it ignores the SEG2 in MYPROG.B and links SEG4.
4. It searches MYPROG.A and finds that it has already linked SEG2 and SEG4; it ignores those two segments and links SEG3.

Important: Keep in mind that for partial compiles to work, the order in which segments are linked must not be significant.

Note: You can use the CRUNCH command, described in Chapter 3, to combine all of the alphabetic-extension files for a program into a single .A file. The CRUNCH command scans the files for the latest version of each segment and restores the segments to their original order.

Help Files

When you add a new language to APW, you should provide a help file to go with it. Help files are ASCII text files (APW language-type PRODOS) that have the same name as the command, and that are kept in the APW/UTILTIES/HELP/ subdirectory. To see an example of a help file for an APW language, enter the following command:

```
HELP ASM65816
```

If your language includes language-specific parameters for the COMPILE, CMPL, and CMPLG commands, you should provide replacement help files for those commands (and their aliases) as well.

Notice that the user cannot scroll through a help file; the text should fit on one screen.

Interpreters

Installing an interpreter under APW is almost identical to installing a compiler, with the following exceptions:

- Interpreted code is not linked. An interpreter cannot make calls to code compiled by a compiler, because the linker cannot be used to combine interpreted and compiled code.
- An interpreter should clear all three operations flags of the lops parameter in the SET_LINFO call when returning control to the shell. Since the interpreter executes the program, linking and separate execution are not needed.

Chapter 7

File Formats

This chapter describes and defines two standard file formats used on the Apple IIGS: the text-file format, which is used for standard ASCII text files and program source files by all APW programs; and the object module format, which is used for all APW object files, library files, and load files. The Apple IIGS System Loader requires that a load file conform to object module format.

Text-File Format

Under ProDOS 8, each application defines its own format for text and data files. On the Apple IIGS, there is a standard format for text files, so that any program that conforms to the standard can read text files written by any other standard program. This format does not preclude the use of files in other formats by these programs; however, to be considered a standard application on the Apple IIGS, it is required that a program be capable of reading and writing files in the standard text-file format.

An Apple IIGS text file contains ASCII codes representing printable characters, plus a few specific control characters. When displayed on a screen or printed out, a text file can be read by humans; that is, there are no binary codes that specify printing formats, printer controls, graphics patterns, and so forth. Related file types, such as word processor files that contain representations of ASCII text but include formatting information, should be assigned unique file types.

Text-File Specifications

An Apple IIGS text file has the following attributes:

- It consists of zero or more lines.
- Each line consists of zero or more ASCII character codes in the range $00 to $FF.
- Each line ends with the ASCII code $0D (carriage return); every time the character code $0D appears, it indicates the end of a line. Even the last line of the file must end with $0D.
- There are no gaps in the file; that is, every character code is part of a line.
- The end of a text file is determined by the ProDOS 16 end-of-file (EOF) pointer. EOF is part of the file descriptor maintained by ProDOS 16, not part of the file itself.

A line with zero characters contains only the end-of-line code, $0D. A text file of length zero contains no lines, characters, carriage returns, or anything else.

This file format includes no provision for file compression or for including descriptive information about the file. Information about the file can be encoded in publicly available file descriptor fields or in another file associated with the given file. For example, a text editor might store the tab stop values for the file `TEXTFILE` in the associated file `TEXTFILE.TABS`. Such file associations must be defined by the individual application.

The following characters require special handling:

HT ($09): Horizontal Tab

A program reading the file should interpret HT as a **field delimiter**, where the definition of field delimiter is left to the individual application. A field delimiter usually denotes a definite separation between characters, whether or not there are space characters between the characters or white space when the line is printed out. A program writing out a line that contains an HT character should insert enough spaces to get to the next tab stop before writing out subsequent characters. The definition of *tab stop* is left to the individual application.

LF ($0A): Line Feed

A program writing out a line that contains a line-feed character should move the cursor to the next line without changing its horizontal position. A carriage-return/line-feed sequence should be handled on the screen like a carriage return: the cursor should be moved to the beginning of the next line.

CR ($0D): Carriage Return

The carriage-return character indicates the end of a line. A program writing out a line that contains a CR character should move the cursor to the beginning of the next line. When a CR character is sent to a printer, it may or may not also cause a line feed, depending on the printer and the settings of dip switches and printer options.

FF ($12): Form Feed

The form-feed character usually causes a printer to scroll to the beginning of the next page. When writing a line to the screen, your program can treat an FF like a carriage return, or it can add blank lines to fill out the page of text. If your program has a convention to indicate page breaks, the FF character should be interpreted as a page break.

SP ($20): Space

A character that prints as a blank space.

High ASCII ($80–$FF)

These codes are used by some programs on Apple IIGS for special characters, such as Greek letters and block graphics (depending on the character font in use). Your program

can display these characters on the screen in any way you choose. If you elect to strip the high bit, be sure to handle characters $80 through $9F and $FF carefully, because the low-ASCII equivalents of these codes ($00 through $1F and $7F) represent special codes to some programs and printers.

Other Characters

Other characters have no specific interpretation in this specification. It is recommended that you limit text files to printable characters ($21 through $7E and $80 through $FF) plus CR, LF, FF, HT, and SP.

Examples

Let the symbols [and] represent the beginning and end of the file, respectively. Then the following text files store the specified text:

Text consisting of no characters:

```
[]
```

Text consisting of one line with no characters:

```
[$0D]
```

Text consisting of two lines with no characters in either line:

```
[$0D $0D]
```

Text consisting of the line *Hi there!*:

```
[$48 $69 $20 $74 $68 $65 $72 $65 $21 $0D]
```

Text consisting of the two lines

Hi
there!

```
[$48 $69 $0D $20 $74 $68 $65 $72 $65 $21 $0D]
```

Object Module Format

Important: This section describes Version 2.0 of the Apple IIGS object module format (OMF). The System Loader supports files written in either Version 2.0 or Version 1.0 of the OMF. The APW Linker, however, creates load files that conform to Version 1.0 of the OMF. Notes in this section describe the differences between Version 1.0 and Version 2.0 of the OMF. The Compact utility program, described in Chapter 3, converts load files from Version 1.0 to Version 2.0.

Under ProDOS 8 on the Apple IIe and Apple IIc, there is only one loadable file format, called the **binary file format**, which consists of one absolute memory image along with its destination address. ProDOS 8 does not have a relocating loader, so that even if you write relocatable code, you must specify the memory location at which the file is to be loaded. The Apple IIGS uses a more general format that allows dynamic loading and unloading of file segments while a program is running and that supports the various needs of many languages and assemblers. The APW Linker and System Loader fully support relocatable code; in general, you do not specify a load address for an Apple IIGS program, but let the loader and Memory Manager determine where to load the program.

The Apple IIGS object module format (OMF) supports language, APW Linker, library, and System Loader requirements, and it is extremely flexible, easy to generate, and fast to load.

There are four kinds of files that use object module format: object files, library files, load files, and run-time library files.

- *Object files* are the output from an assembler or compiler and the input to a linker. Object files must be fast to process, easy to create, independent of the source language, and able to support libraries in a convenient way. In APW, object files also support segmentation of code and partial assemblies and compiles. They support both absolute and relocatable program segments.

  Apple IIGS object files contain both machine-language code and relocation information for use by the linker. Object files cannot be loaded directly into memory; they must first be processed by the linker to create load files.

- *Library files* contain general object segments that a linker can find and extract to resolve references unresolved in the object files. Only the code needed during the link process is extracted from the library file.

- *Load files*, which are the output of a linker, contain memory images that a loader loads into memory. Load files must be *very* fast to process. Apple IIGS load files contain load segments that can be relocatable, movable, dynamically loadable, or have any combination of these attributes. *Shell load files* are load files that can be run from a shell program without requiring the shell to shut down. *Startup load files* are load files that ProDOS 16 loads during its startup.

  Load files are created by the linker from object files and library files. Load files can be loaded into memory by the System Loader; they *cannot* be used as input to the linker.

- *Run-time library files* are load files containing general routines that can be shared between applications. The routines are contained in file segments that can be loaded as needed by the System Loader and then purged from memory when they are no longer needed. Run-time library files are not currently supported by the APW Linker but are defined in the OMF to allow for future enhancements to the system.

All four types of files consist of individual components called *segments*. Each file type uses a subset of the full object module format. Each compiler or assembler uses a subset of the format depending on the requirements and complexity of the language.

The ProDOS 16 file types used by APW are as follows:

File Type	Name	Mnemonic
$B0	source	SRC
$B1	object	OBJ
$B2	library	LIB
$B3	load	S16
$B4	run-time library	RTL
$B5	shell load	EXE
$B6	startup load	STR
$B7–$BE	other load file types	

An APW source file has an auxiliary type that represents the programming language for which it is to be used.

The rest of this chapter defines object module format. First, the general format specification for all OMF files is described. Then, the unique characteristics of each of the following file types are discussed:

- object files
- library files
- load files
- run-time library files
- shell load files

General Format for OMF Files

Each object-module-format (OMF) file contains one or more segments. Figure 7.1 represents the structure of an OMF file. Each segment in an object file is a separate entity that contains all the information needed to link it with other segments (and to relocate it if it is relocatable code). Each segment in a load file is a separate entity that contains all the information needed to load it into memory. Load file segments on the Apple IIGS are usually relocatable.

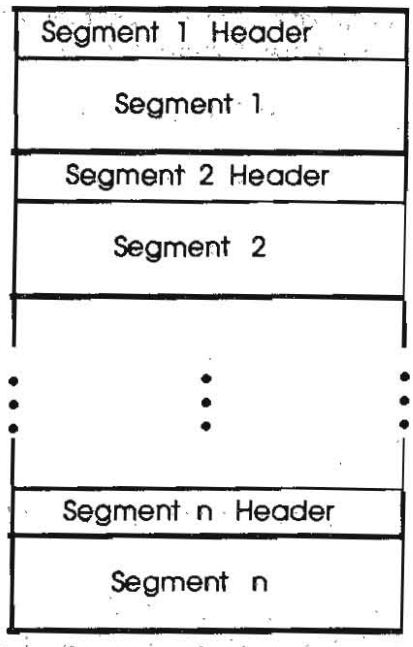


Figure 7.1. The Structure of an OMF File

Each segment in an OMF file contains a set of records that indicate relocation information or contain code or data. If the file is an object file, the linker processes each record and generates a load file containing load segments. Object code includes the information the linker needs to generate a relocatable load segment. Load files consist of a memory image followed by a relocation dictionary; the System Loader loads the memory image and then processes the information in the relocation dictionary. Relocation dictionaries are discussed in the section "Load Files" later in this chapter.

Segments in object files can be combined by the linker into one or more segments in the load file (see the discussion of the LOADNAME field in the section "Segment Header" later in this chapter). For instance, each subroutine in a program can be placed in a separate code segment and compiled independently; then the linker can be told to place all the code segments into one load segment.

Segment Types and Attributes

Each OMF segment has a segment type and can have several attributes. The following segment types are defined by the object module format:

- code
- data
- jump table segment
- pathname segment
- library dictionary segment
- initialization segment
- direct-page/stack segment

The following segment attributes are defined by the object module format:

- reloadable or not reloadable
- absolute-bank or not restricted to a particular bank
- loadable in special memory or not loadable in special memory
- position-independent or position-dependent
- private or nonprivate
- static or dynamic

Code and **data segments** are object segments provided to support languages that distinguish program code from data. A segment specified by using a START assembler directive is flagged as a code segment; if you use a DATA directive instead, the segment is a data segment.

Jump table segments and **pathname segments** are load segments that facilitate the dynamic loading of segments; they are described in the section "Load Files" later in this chapter.

Library dictionary segments allow the linker to quickly scan library files for needed segments; they are described in the section "Library Files" later in this chapter.

Initialization segments are optional parts of load files that are used to perform any initialization required by the application during an initial load. If used, they are loaded and executed immediately when they are found by the System Loader and are reloaded any time the program is restarted from memory. Initialization segments are described in the section "Load Files" later in this chapter.

Direct-page/stack segments are load segments used to preset the direct-page and stack registers for an application. See the section "Direct-Page/Stack Segments" later in this chapter for more information.

Reload segments are load segments that the loader must reload even if the program is restartable and is restarted from memory.

Version 1.0: Reload segments do not exist in Version 1.0 of the OMF.

Absolute-bank segments are load segments that are restricted to a specified bank but that can be relocated within that bank. The ORG field in the segment header specifies the bank to which the segment is restricted.

Loadable in special memory means that a segment can be loaded in banks $00, $01, $E0, and $E1. Because these are the banks used by programs running under ProDOS 8 in standard-Apple II emulation mode, you may wish to prevent your program from being loaded in these banks so that it can remain in memory while programs are run under ProDOS 8.

> **Version 1.0:** The loadable-in-special-memory attribute for segments does not exist in Version 1.0 of the OMF.

Position-independent segments can be moved during program execution.

A **private code segment** is a segment in an object file whose name is available only to other object-code segments within the same object file. (The labels within a code segment are local to that segment.)

A **private data segment** is a segment in an object file whose labels are available only to object-code segments in the same object file.

Static segments are load segments that are loaded at program execution time and are not unloaded during execution; **dynamic segments** are loaded and unloaded during program execution as needed. A segment can be designated as dynamic with the `/DYNAMIC` qualifier to the `SEGMENT` command in a LinkEd file. If you do not use a LinkEd file, all segments in your program are static

A segment can have only one segment type but can have any combination of attributes. The segment types and attributes are specified in the segment header by the `KIND` segment-header field, described in the next section.

Segment Header

Each segment in an OMF file has a header that contains general information about the segment, such as its name and length. Segment headers make it easy for the linker to scan an object file for the desired segments, and they allow the System Loader to load individual load segments. The format of the segment header is illustrated in Figure 7.2.

> **Version 1.0:** Figure 7.3 illustrates the format of the segment header in Version 1.0 of the OMF.

Following the figures is a detailed description of each of the fields in the segment header.

> **Important:** In future versions of the OMF, additional fields may be added to the segment header between the `DISPDATA` and `LOADNAME` fields. In order to assure that future expansion of the segment header does not affect your program, always use `DISPNAME` and `DISPDATA` instead of absolute offsets when referencing `LOADNAME`, `SEGNAME`, and the start of the segment body.

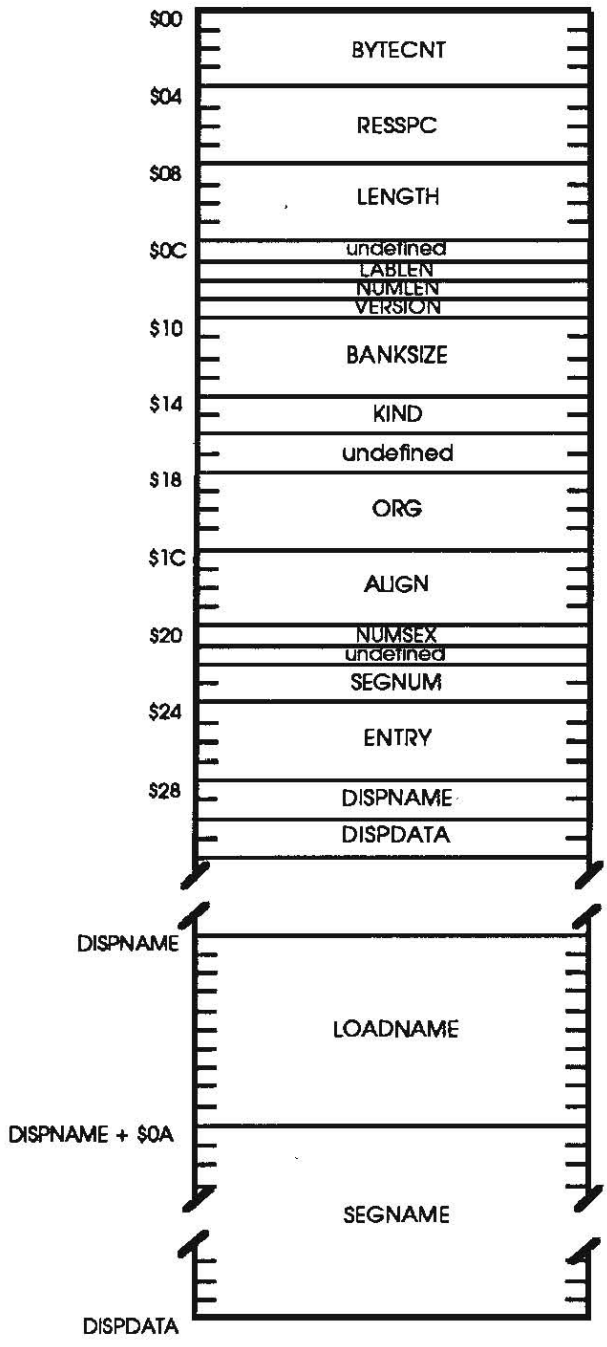


Figure 7.2. The Format of a Version 2.0 Segment Header

Version 1: In version 1 of the OMF, the segment header is as shown in Figure 7.3.

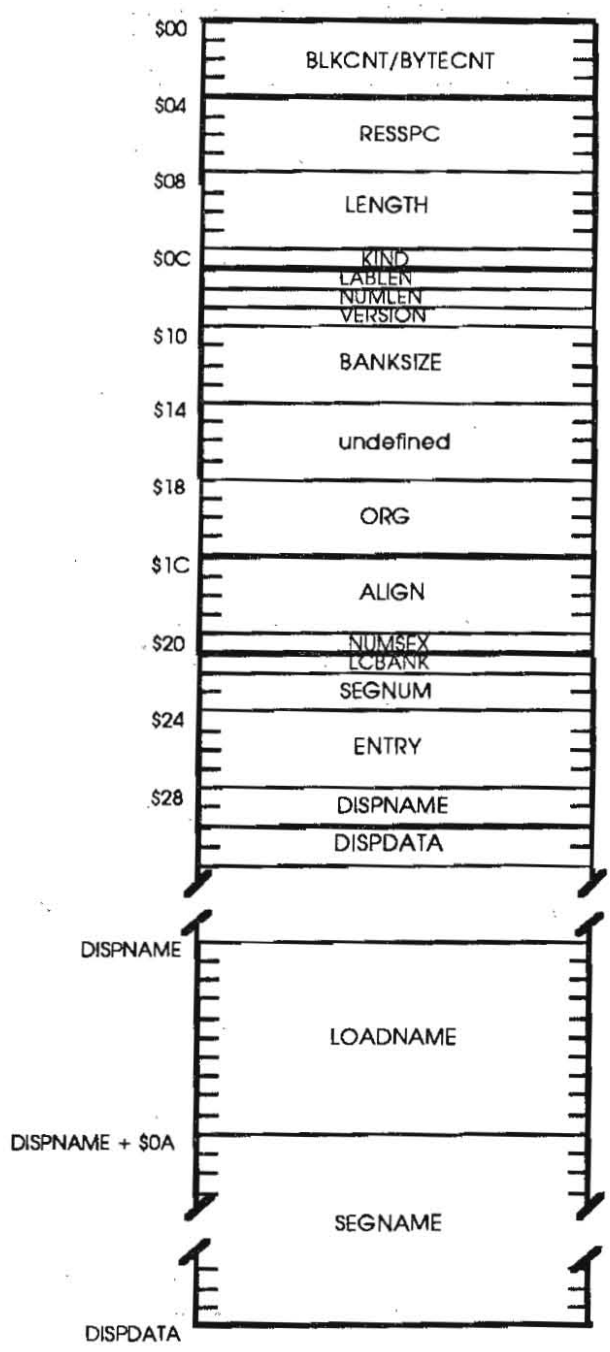


Figure 7.3. The Format of a Version 1.0 Segment Header

BYTECNT: A 4-byte field containing the number of bytes in the file that the segment requires. This number includes the segment header, so you can calculate the starting Mark of the next segment from the starting Mark of this segment plus BYTECNT. Segments need not be aligned to block boundaries.

Version 1.0 In Version 1.0, this field is described as follows. For object files and load files, BLKCNT is a 4-byte field containing the number of blocks in the file that the segment requires. Each block is 512 bytes. The segment header is part of the first block of the segment. Segments in an object file or load file start on block boundaries. For library files (ProDOS 16 file type $B2), this field is BYTECNT, indicating the number of bytes in the segment. Library-file segments are not aligned to block boundaries.

RESSPC: A 4-byte field containing the number of bytes of zeros to add to the end of the segment. This field can be used in an object segment instead of a large block of zeros at the end of the segment. Using this field can thus significantly reduce the block size of an object segment when the source code ends with a DS directive that reserves a large block of memory.

LENGTH: A 4-byte field containing the memory size that the segment will require when loaded. It includes the extra memory specified by RESSPC.

LENGTH is followed by one undefined byte, reserved for future changes to the segment header specification.

LABLEN: A 1-byte field indicating how long each name or label record in the segment body is in bytes. If LABLEN is 0, it indicates that the length of each name or label is specified in the first byte of the record (that is, the first byte of the record specifies how many bytes follow). LABLEN also specifies the length of the SEGNAME field of the segment header. (The LOADNAME field always has a length of 10 bytes.) Fixed-length labels are always left-justified and padded with spaces.

NUMLEN: A 1-byte field indicating how long each number field in the segment body is in bytes. This field is 4 for the Apple IIGS.

VERSION: A 1-byte field indicating the version number of the object module format with which the segment is compatible. This field is 2 for the current specification of the object module format.

BANKSIZE: A 4-byte binary number indicating the maximum memory-bank size for the segment. If the segment is in an object file, the linker assures that the segment is not larger than this value (the linker returns an error if the segment is too large). If the segment is in a load file, the loader ensures that the segment is loaded into a memory block that does not cross this boundary. For Apple IIGS code segments, this field must be $00010000, indicating a 64K bank size. A value of 0 in this field indicates that the segment can cross bank boundaries. Apple IIGS data segments can use any number from $00 to $00010000 for BANKSIZE.

KIND: A 2-byte field specif ying the type and attributes of the segment. The bits are defined as follows. The column labeled *Where Described* indicates the section in this chapter where the particular segment type or attribute is discussed:

Bit	Meaning		Where Described
0–4	Segment Type		
	$00	code	Segment Types and Attributes
	$01	data	Segment Types and Attributes
	$02	jump table segment	Load Files
	$04	pathname segment	Segment Types and Attributes
	$08	library dictionary segment	Library Files
	$10	initialization segment	Load Files
	$12	direct-page/stack segment	Direct-Page/Stack Segments
10–15	Segment Attribute		
10	1 = reload segment		Segment Types and Attributes
11	1 = absolute-bank segment		Segment Types and Attributes
12	0 = can be loaded in special memory		Segment Types and Attributes
13	1 = position-independent		Segment Types and Attributes
14	1 = private		Segment Types and Attributes
15	0 = static; 1 = dynamic		Segment Types and Attributes

A segment can have only one type but any combination of attributes. For example, a position-independent dynamic data segment has KIND = ($A001).

Important: If segment KINDs are specified in the source file and the KINDs of the object segments placed in a given load segment are not all the same, the segment KIND of the resulting load segment is unpredictable.

KIND is followed by two undefined bytes, reserved for future changes to the segment header specification.

Version 1.0 In Version 1.0 of the OMF, the KIND field is 1 byte long, defined as follows:

Bit	Meaning	
0–4	Segment Type	
	$00	code
	$01	data
	$02	jump table segment
	$04	pathname segment
	$08	library dictionary segment
	$10	initialization segment
	$11	absolute-bank segment
	$12	direct-page/stack segment
5–7	Segment Attribute	
5	1=position-independent	
6	1=private	
7	0=static; 1=dynamic	

ORG: A 4-byte field indicating the absolute address at which this segment is to be loaded in memory, or, for an absolute-bank segment, the bank number. A value of 0 indicates that this segment is relocatable and can be loaded anywhere in memory. A value of 0 is normal for the Apple IIGS.

ALIGN: A 4-byte binary number indicating the boundary on which this segment must be aligned. For example, if the segment is to be aligned on a page boundary, this field is $00000100; if the segment is to be aligned on a bank boundary, this field is $00010000. A value of 0 indicates that no alignment is needed. For the Apple IIGS, this field must be a power of 2, less than or equal to $00010000. Currently, the loader supports only values of 0, $00000100, and $00010000; for any other value, the loader uses the next higher supported value.

NUMSEX: A 1-byte field indicating the order of the bytes in a number field. If this field is 0, the *least* significant byte is first. If this field is 1, the *most* significant byte is first. This field is 0 for the Apple IIGS.

NUMSEX is followed by one undefined byte, reserved for future changes to the segment header specification.

> **Version 1.0:** In Version 1.0 of the OMF, the NUMSEX field is followed by the LCBANK field. The LCBANK field is described as follows. A 1-byte field indicating the bank of the language card into which the segment is to be loaded: if 0, bank 1; if 1, bank 2. LCBANK is meaningful only if the ORG field contains an address in the language card area ($D000 through $E000) of banks 0, 1, E0, or E1. The System Loader does not support the loading of segments into alternate banks of the language card.

SEGNUM: A 2-byte field specifying the segment number. The segment number corresponds to the relative position of the segment in the file (starting with 1). This field is used by the System Loader as a check while searching for a specific segment in a load file.

ENTRY: A 4-byte field indicating the offset into the segment that corresponds to the entry point of the segment.

DISPNAME: A 2-byte field indicating the displacement of the LOADNAME field within the segment header. Currently, DISPNAME = 44. DISPNAME is provided to allow for future additions to the segment header; any new fields will be added between DISPDATA and LOADNAME. DISPNAME allows you to reference LOADNAME and SEGNAME no matter what the actual size of the header.

DISPDATA: A 2-byte field indicating the displacement from the start of the segment header to the start of the segment body. Currently, DISPDATA = 54 + LABLEN. DISPDATA is provided to allow for future addtions to the segment header; any new fields will be added between DISPDATA and LOADNAME. DISPDATA allows you to reference the start of the segment body no matter what the actual size of the header.

LOADNAME: A 10-byte field specifying the name of the load segment that will contain the code generated by the linker for this segment. More than one segment in an object file can be merged by the linker into a single segment in the load file. This field is unused in a load segment. The position of LOADNAME may change in future revisions of the OMF; therefore, you should always use DISPNAME to reference LOADNAME.

SEGNAME: A field LABLEN bytes long, specifying the name of the segment. The position of SEGNAME may change in future revisions of the OMF; therefore, you should always use DISPNAME to reference SEGNAME.

Segment Body

The body of each segment is composed of sequential records, each of which starts with a 1-byte operation code. Each record contains either program code or information for the linker or System Loader. All names and labels included in these records are LABLEN bytes long, while all numbers and addresses are NUMLEN bytes long (unless otherwise specified in the following definitions). For the Apple IIGS, the least significant byte of each number field is first, as specified by NUMSEX.

Several of the OMF records contain expressions that have to be evaluated by the linker. The operation and syntax of expressions are described in the next section, "Expressions." If the description of the record type does not explicitly state that the opcode is followed by an expression, then an expression *cannot* be used. Expressions are never used in load segments.

The operation codes and segment records are described in this section, listed in order of the opcodes. Table 7.1 provides an alphabetical cross-reference between segment record types and opcodes. Library files consist of object segments, and so can use any record type that can be used in an object segment. Table 7.1 also lists the segment types in which each record type can be used.

Table 7.1. Segment-Body Record Types

Record Type	Opcode	Segment Types
ALIGN	$E0	object
BEXPR	$ED	object
cINTERSEG	$F6	load
CONST	$01–$DF	object
cRELOC	$F5	load
DS	$F1	all
END	$00	all
ENTRY	$F4	run-time library
EQU	$F0	object
EXPR	$EB	object
GEQU	$E7	object
GLOBAL	$E6	object
INTERSEG	$E3	load
LCONST	$F2	load
LEXPR	$F3	object
LOCAL	$EF	object
MEM	$E8	object
ORG	$E1	object
RELEXPR	$EE	object
RELOC	$E2	load
STRONG	$E5	object
SUPER	$F7	load
USING	$E4	object
ZEXPR	$EC	object

The rest of this section defines each of these record types. The record types are listed in order of their opcodes.

Record Type	Opcode	Description
END	$00	This record indicates the end of the segment.
CONST	$01–$DF	This record contains absolute data that needs no relocation. The operation code specifies how many bytes of data follow.
ALIGN	$E0	This record contains a number that indicates an alignment factor. The linker inserts as many zero bytes as necessary to move to the memory boundary indicated by this factor. The value of this factor is in the same format as the ALIGN field in the segment header and cannot have a value greater than that in the ALIGN field. ALIGN must equal a power of 2.
ORG	$E1	This record contains a number that is used to increment or decrement the location counter. If the location counter is incremented (ORG is positive), zeros are inserted to get to the new address. If the location counter is decremented (ORG is a twos complement negative number), the location counter is decremented and subsequent code overwrites the old code.
RELOC	$E2	This is a relocation record, which is used in the relocation dictionary of a load segment. It is used to patch an address in a load segment with a reference to another address in the same load segment. It contains two 1-byte counts followed by two offsets. The first count is the number of bytes to be relocated, and the second count is a bit-shift operator, telling how many times to shift the relocated address before inserting the result into memory. If the bit-shift operator is positive, the number is shifted to the left, filling vacated bit positions with zeros (logical shift left). If the bit-shift operator is (two's complement) negative, the number is shifted right (logical shift right). The first offset gives the location (relative to the start of the segment) of the (first byte of the) number that is to be patched (relocated). The second offset is the location of the reference relative to the start of the segment; that is, it is the value that the number would have if the segment it's in started at address $000000. For example, suppose the segment includes the following lines:

```
35    LABEL . . .
 .
 .
 .
400   LDA   LABEL+4
```

LABEL is a local reference to a location 53 ($35) bytes after the start of the segment. When this segment is loaded into memory, the value of LABEL+4 depends on the starting location of the segment, so the linker creates a RELOC record in the relocation dictionary for this value. LABEL+4 is two bytes long; that is, the number of bytes to be relocated is 2. No bit-shift operation is needed. The number to be calculated during relocation is 1025 ($401) bytes after the start of the segment (immediately after the LDA, which is one byte). The value of LABEL+4 would be $39 if the segment started at address $000000.

The RELOC record for the number to be loaded into the A register by this statement would therefore look like this (note that the values are stored low-byte first, as specified by NUMSEX):

```
E2020001 04000039 000000
```

This sequence corresponds to the following values:

$E2	operation code
$02	number of bytes to be relocated
$00	bit-shift operator
$00000401	offset of value from start of segment
$00000039	value if segment started at $000000

Note: Certain types of arithmetic expressions are illegal in a relocatable segment; specifically, any expression that cannot be evaluated (relative to the start of the segment) by the assembler cannot be used. The expression LAB | 4 can be evaluated, for example, since the RELOC record includes a bit-shift operator. The expression LAB | 4+4 cannot be used, however, because the assembler would have to know the absolute value of LAB in order to perform the bit-shift operation *before* adding 4 to it. Similarly, the value of LAB*4 depends on the absolute value of LAB, and cannot be evaluated relative to the start of the segment, so multiplication is illegal in expressions in relocatable segments.

INTERSEG $E3

This record is used in the relocation dictionary of a load segment and contains a patch to a long call to an external reference. The INTERSEG record is used to patch an address in a load segment with a reference to another address in a different load segment. It contains two 1-byte counts followed by an offset, a 2-byte file number, a 2-byte segment number, and a second offset. The first count is the number of bytes to be relocated, and the second count is a bit-shift operator, telling how many times to shift the relocated address before inserting the result into memory. If the bit-shift operator is positive, the number is shifted to the left, filling vacated bit positions with zeros (logical shift left). If the bit-shift operator is (two's complement) negative, the number is shifted right (logical shift right).

The first offset is the location (relative to the start of the segment) of the (first byte of the) number that is to be relocated. If the reference is to a static segment, the **file number**, **segment number**, and second offset correspond to the subroutine referenced. (The linker assigns a file number to each load file in a program. This feature is provided primarily to support run-time libraries. In the normal case of a program having one load file, the file number is 1. The load segments in a load file are numbered by their relative location in the load file, where the first load segment is number 1.) If the reference is to a dynamic segment, the file and segment numbers correspond to the jump table segment, and the second offset corresponds to the call to the System Loader for that reference.

For example, suppose the segment includes an instruction like

```
JSL EXT
```

The label EXT is an external reference to a location in a *static* segment.

If this instruction is at relative address $720 within its segment and EXT is at relative address $345 in segment $000A in file $0001, the linker creates an INTERSEG record in the relocation dictionary that looks like this (note that the values are stored low-byte first, as specified by NUMSEX):

```
E3030021 07000001 000A0045 030000
```

This sequence corresponds to the following values:

$E3	operation code
$03	number of bytes to be relocated
$00	bit-shift operator
$00000721	offset of instruction's operand
$0001	file number
$000A	segment number
$00000345	offset of subroutine referenced

When the loader processes the relocation dictionary, it uses the first offset to find the JSL and patches in the address corresponding to the file number, segment number, and offset of the referenced subroutine.

If the JSL is to an external reference in a *dynamic* segment, the INTERSEG records refer to the file number, segment number, and offset of the call to the System Loader in the jump table segment.

If the jump table segment is in segment 6 of file 1, and the call to the System Loader is at relative location $2A45 in the jump table segment, then the INTERSEG record looks like this (note that the values are stored low-byte first, as specified by NUMSEX):

```
E3030021 07000001 00060045 2A0000
```

This sequence corresponds to the following values:

$E3	operation code
$03	number of bytes to be relocated
$00	bit-shift operator
$00000721	offset of instruction's operand
$0001	file number of jump table segment
$0006	segment number of jump table segment
$00002A45	offset of call to System Loader

The jump table segment entry that corresponds to the external reference EXT contains the following values:

User ID	
$0001	file number
$0005	segment number
$00000200	offset of instruction
call to System Loader	

INTERSEG records are used for any long-address reference to a static segment.

See the section "Jump Table Segment" in this chapter for a discussion of the function of the jump table segment.

USING $E4 This record contains the name of a data segment. After this record is encountered, local labels from that data segment can be used in the current segment.

STRONG $E5 This record contains the name of a segment that must be included during linking even if no external references have been made to it.

GLOBAL $E6

This record contains the name of a global label followed by three attribute fields. The label is assigned the current value of the location counter. The first attribute field is 2 bytes long and gives the number of bytes generated by the line that defined the label. If this field is $FFFF, it indicates that the actual length is unknown but that it is greater than or equal to $FFFF. The second attribute field is 1 byte long and specifies the type of operation in the line that defined the label. The following type attributes are defined:

A	address-type DC statement
B	Boolean-type DC statement
C	character-type DC statement
D	double-precision floating-point-type DC statement
F	floating-point-type DC statement
G	EQU or GEQU statement
H	hexadecimal-type DC statement
I	integer-type DC statement
K	reference-address-type DC statement
L	soft-reference-type DC statement
M	instruction
N	assembler directive
O	ORG statement
P	ALIGN statement
S	DS statement
X	arithmetic symbolic parameter
Y	Boolean symbolic parameter
Z	character symbolic parameter

The third attribute field is 1 byte long and is the `private` flag (1 = private). This flag is used to designate a code or data segment as private (see the section "Segment Types and Attributes" in this chapter for a definition of private segments).

GEQU $E7

This record contains the name of a global label followed by three attribute fields and an expression. The label is given the value of the expression. The first attribute field is 2 bytes long and gives the number of bytes generated by the line that defined the label. The second attribute field is 1 byte long and specifies the type of operation in the line that defined the label, as listed in the discussion of the GLOBAL record. The third attribute field is 1 byte long and is the `private` flag (1 = private). This flag is used to designate a code or data segment as private (see the section "Segment Types and Attributes" earlier in this chapter for a definition of private segments).

MEM $E8

This record contains two numbers that represent the starting and ending addresses of a range of memory that must be reserved.

EXPR	$EB	This record contains a 1-byte count followed by an expression. The expression is evaluated, and its value is truncated to the number of bytes specified in the count. The order of the truncation is from most significant to least significant.
ZEXPR	$EC	This record contains a 1-byte count followed by an expression. ZEXPR is identical to EXPR, except that any bytes truncated must be all zeros. If the bytes are not zeros, the record is flagged as an error.
BEXPR	$ED	This record contains a 1-byte count followed by an expression. BEXPR is identical to EXPR, except that any bytes truncated must match the corresponding bytes of the location counter. If the bytes don't match, the record is flagged as an error. This record allows the linker to make sure that an expression evaluates to an address in the current memory bank.
RELEXPR	$EE	This record contains a 1-byte length followed by an offset and an expression. The offset is NUMLEN bytes long. RELEXPR is used to generate a relative branch value that involves an external location. The length indicates how many bytes to generate for the instruction, the offset indicates where the origin of the branch is relative to the current location counter, and the expression is evaluated to yield the destination of the branch. For example, a BNE LOC instruction, where LOC is external, generates this record. For the 6502 and 65816 microprocessors, the offset is 1.
LOCAL	$EF	This record contains the name of a local label followed by three 1-byte attribute fields. The label is assigned the value of the current location counter. The first attribute byte gives the number of bytes generated by the line that defined the label. The second attribute byte specifies the type of operation in the line that defined the label, as listed in the discussion of the GLOBAL record. The third attribute byte is the private flag (1 = private). This flag is used to designate a code or data segment as private (see the section "Segment Types and Attributes" earlier in this chapter for a definition of private segments). Note that the linker ignores local labels from code segments and that it recognizes local labels from other data segments only if a USING record was processed (see the discussion of the USING statement).

EQU	$F0	This record contains the name of a local label followed by three 1-byte attribute fields and an expression. The label is given the value of the expression. The first attribute byte gives the number of bytes generated by the line that defined the label. The second attribute byte specifies the type of operation in the line that defined the label, as listed in the discussion of the GLOBAL record. The third attribute byte is the `private` flag (1 = private). This flag is used to designate a code or data segment as private (see the section "Segment Types and Attributes" earlier in this chapter for a definition of private segments).
DS	$F1	This record contains a number indicating how many bytes of zeros to insert at the current location counter.
LCONST	$F2	This record contains a 4-byte count followed by absolute code or data. The count indicates the number of bytes of data. The LCONST record is similar to CONST except that it allows for a much greater number of data bytes. Each relocatable load segment consists of LCONST records, DS records, and a relocation dictionary. See the discussions on INTERSEG records, RELOC records, and the relocation dictionary for more information.
LEXPR	$F3	This record contains a 1-byte count followed by an expression. The expression is evaluated, and its value is truncated to the number of bytes specified in the count. The order of the truncation is from most significant to least significant. If the expression evaluates to a single label with a fixed, constant offset, and if the label is in another segment and that segment is a dynamic code segment, then the linker is allowed to create an entry for that label in the jump table segment. (The jump table segment provides a mechanism to allow dynamic loading of segments as they are needed—see the section "Load Files" later in this chapter.) Only a JSL instruction should generate an LEXPR record.
ENTRY	$F4	This record is used in the run-time-library entry dictionary; it contains a 2-byte number and an offset followed by a label. The number is the segment number. The label is a code-segment name or entry and the offset is the relative location within the load segment of the label. Run-time library entry dictionaries are described in the section "Run-Time Library Files" in this chapter.

cRELOC $F5

This record is the compressed version of the RELOC record. It is identical to the RELOC record, except that the offsets are 2 bytes long rather than 4 bytes. The cRELOC record can be used only if both offsets are less than $10000 (65536). The following example compares a RELOC record and a cRELOC record for the same reference (for an explanation of each line of these records, see the discussion of the RELOC record):

RELOC	**cRELOC**
$E2	$F5
$02	$02
$00	$00
$00000401	$0401
$00000039	$0039
(11 bytes)	(7 bytes)

cINTERSEG $F6

This record is the compressed version of the INTERSEG record. It is identical to the INTERSEG record, except that the offsets are 2 bytes long rather than 4 bytes, the segment number is 1 byte rather than 2 bytes, and it does not include the 2-byte file number. The cINTERSEG record can be used only if both offsets are less than $10000 (65536), the segment number is less than 256, and the file number associated with the reference is 1 (that is, the initial load file). References to segments in run-time library files must use INTERSEG records rather than cINTERSEG records.

The following example compares an INTERSEG record and a cINTERSEG record for the same reference (for an explanation of each line of these records, see the discussion of the INTERSEG record):

INTERSEG	**cINTERSEG**
$E3	$F6
$03	$03
$00	$00
$00000720	$0720
$0001	
$000A	$0A
$00000345	$0345
(15 bytes)	(8 bytes)

SUPER $F7

This is a supercompressed relocation-dictionary record. Each SUPER record is the equivalent of many cRELOC, cINTERSEG, and INTERSEG records. It contains a 4-byte length, a 1-byte record type, and one or more subrecords of variable size, as follows:

opcode	$F7
length	number of bytes in the rest of the record (4 bytes)
type	0–37 (1 byte)
subrecords	(variable size)

Version 1.0: SUPER records do not exist in Version 1.0 of the OMF.

When SUPER records are used, some of the relocation information is stored in the LCONST record at the address to be patched.

The length field indicates the number of bytes in the rest of the SUPER record (that is, the number of bytes exclusive of the opcode and the length field).

The type byte indicates the type of SUPER record. There are 38 types of SUPER record, as follows:

Type	SUPER record
0	RELOC2
1	RELOC3
2–37	INTERSEG1–INTERSEG36

SUPER RELOC2: This record can be used instead of cRELOC records that have a bit-shift count of 0 and that relocate 2 bytes.

SUPER RELOC3: This record can be used instead of cRELOC records that have a bit-shift count of 0 and that relocate 3 bytes.

SUPER INTERSEG1: This record can be used instead of cINTERSEG records that have a bit-shift count of 0 and that relocate 3 bytes.

SUPER INTERSEG2 through SUPER INTERSEG12: The number in the name of the record refers to the file number of the file in which the record is used. For example, to relocate an address in file number 6, use a SUPER INTERSEG6 record. These records can be used instead of INTERSEG records that meet the following criteria:

- Both offsets are less than $10000.
- The segment number is less than 256.
- The bit-shift count is 0.
- The record relocates 3 bytes.
- The file number is from 2 through 12.

`SUPER INTERSEG13` through `SUPER INTERSEG24`: These records can be used instead of `cINTERSEG` records that have a bit-shift count of 0, that relocate 2 bytes, and that have a segment number of *n*–12, where *n* can be from 13 to 24. For example, to replace a `cINTERSEG` record in segment number 6, use a `SUPER INTERSEG18` record.

`SUPER INTERSEG25` through `SUPER INTERSEG36`: These records can be used instead of `cINTERSEG` records that have a bit-shift count of $F0 (–16), that relocate 2 bytes, and that have a segment number of *n* – 24, where *n* can be from 25 to 36. For example, to replace a `cINTERSEG` record in segment number 6, use a `SUPER INTERSEG30` record.

Each subrecord consists either of either a 1-byte *offset count* followed by a list of 1-byte offsets, or a 1-byte *skip count*.

Each offset count indicates how many offsets are listed in this subrecord. The offsets are 1 byte each. Each offset corresponds to the low byte of the first (2-byte) offset in the equivalent `INTERSEG`, `cRELOC` or `cINTERSEG` record. The high byte of the offset is indicated by the location of this offset count in the `SUPER` record: each subsequent offset count indicates the next 256 bytes of the load segment. Each skip count indicates the number of 256-byte *pages* to skip; that is, a skip count indicates that there are no offsets within a certain number of 256-byte pages of the load segment.

For example, if patches must be made at offsets 0020, 0030, 0140, and 0550 in the load segment, the subrecords would include the following fields:

2 20 30	the first 256-byte page of the load segment has two patches: one at offset 20 and one at offset 30
1 40	the second 256-byte page has one patch at offset 40
skip-3	skip the next three 256-byte pages
1 50	the sixth 256-byte page has one patch at offset 50

In the actual `SUPER` record, the patch count byte is the number of offsets –1 and the skip count byte has the high bit set. A `SUPER INTERSEG1` record with the offsets in the above example would look like this:

$F7	opcode
$00000009	number of bytes in the rest of the record
$02	INTERSEG1-type SUPER record
$01	the first 256-byte page has two patches
$20	patch the load segment at offset $0020
$30	patch the segment at $0030
$00	the second page has one patch
$40	patch the segment at $0140
$83	skip the next three 256-byte pages
$00	the sixth page has one patch
$50	patch the segment at $0550

A comparison with the RELOC record shows that a SUPER RELOC record is missing the offset of the reference. Similarly, the SUPER INTERSEG1 through SUPER INTERSEG12 records are missing the segment number and offset of the subroutine referenced. The offsets (which are 2 bytes long) are stored in the LCONST record at the "to be patched" location. For the SUPER INTERSEG1 through 12 records, the segment number is stored in the third byte of the "to be patched" location.

For example, if the example given in the discussion of the INTERSEG record were instead referenced through a SUPER INTERSEG1 record, the value $0345 (the offset of the subroutine referenced) would be stored at offset $0721 in the load segment (the offset of the instruction's operand) and the segment number ($0A) would be stored at offset $0723, as follows:

```
4503 0A
```

Experimental $FB–$FF These record types are reserved for use in system development by Apple.

Expressions

Several of the OMF records contain expressions. Expressions form an extremely flexible reverse-Polish stack language that can be evaluated by the linker to yield numeric values such as addresses and labels. Each expression consists of a series of *operators* and *operands* together with the values on which they act.

An **operator** takes one or two values from the evaluation stack, performs some mathematical or logical operation on them, and places a new value onto the evaluation stack. The final value on the evaluation stack is used as if it were a single value in the record. Note that this evaluation stack is purely a programming concept and does not relate to any hardware stack in the computer. Each operation is stored in the object module file in postfix form; that is, the value or values come first, followed by the operator. For example, since a binary operation is stored as *Value1 Value2 Operator*, the operation Num1 - Num2 is stored as

```
Num1Num2-
```

The operators are as follows:

Binary Math Operators: These operators take two numbers as two's-complement signed integers from the top of the evaluation stack, perform the specified operation, and place the single-integer result back on the evaluation stack. The binary math operators include

$01	addition	(+)
$02	subtraction	(-)
$03	multiplication	(*)
$04	division	(/)
$05	integer remainder	(MOD)
$07	bit shift	(|)

The subtraction operator subtracts the second number from the first number. The division operator divides the first number by the second number. The integer-remainder operator divides the first number by the second number and returns the unsigned integer remainder to the stack. The bit-shift operator shifts the first number by the number of bit positions specified by the second number. If the second number is positive, the first number is shifted to the left, filling vacated bit positions with zeros (logical shift left). If the second number is negative, the first number is shifted right, preserving the sign bit (arithmetic shift right).

Unary Math Operator: A unary math operator takes a number as a two's-complement signed integer from the top of the evaluation stack, performs the operation on it, and places the integer result back on the evaluation stack. The only unary math operator currently available is

$06	negation	(-)

Comparison Operators: These operators take two numbers as two's-complement signed integers from the top of the evaluation stack, perform the comparison, and place the single-integer result back on the evaluation stack. Each operator compares the second number in the stack (TOS – 1) with the number at the top of the stack (TOS). If the comparison is true, a 1 is placed on the stack; if false, a 0 is placed on the stack. The comparison operators include

$0C	less than or equal to	(<=)
$0D	greater than or equal to	(>=)
$0E	not equal	(<> or !=)
$0F	less than	(<)
$10	greater than	(>)
$11	equal to	(= or ==)

Binary Logical Operators: These operators take two numbers as Boolean values from the top of the evaluation stack, perform the operation, and place the single Boolean result back on the stack. Boolean values are defined as being FALSE for the number 0 and TRUE

for any other number. Logical operators always return a 1 for true. The binary logical operators include

$08	AND	(logical AND)
$09	OR	(inclusive OR)
$0A	EOR	(exclusive OR)

Unary Logical Operator: A unary logical operator takes a number as a Boolean value from the top of the evaluation stack, performs the operation on it, and places the Boolean result back on the stack. The only unary logical operator currently available is

$0B	NOT	(complement)

Binary Bit Operators: These operators take two numbers as binary values from the top of the evaluation stack, perform the operation, and place the single binary result back on the stack. The operations are performed on a bit-by-bit basis. The binary bit operators include

$12	Bit AND	(logical AND)
$13	Bit OR	(inclusive OR)
$14	Bit EOR	(exclusive OR)

Unary Bit Operator: This operator takes a number as a binary value from the top of the evaluation stack, performs the operation on it, and places the binary result back on the stack. The unary bit operator is

$15	Bit NOT	(complement)

Termination Operator: All expressions end with the termination operator $00.

An **operand** causes some value, such as a constant or a label, to be loaded onto the evaluation stack. The operands are as follows:

Location Counter Operand ($80): This operand loads the value of the current location counter onto the top of the stack. Because the location counter is loaded before the bytes from the expression are placed into the code stream, the value loaded is the value of the location counter before the expression is evaluated.

Constant Operand ($81): This operand is followed by a number that is loaded on the top of the stack.

Label Reference Operands ($82–$86): Each of these operand codes is followed by the name of a label, and is acted on as follows:

$82 Weak reference (see the note below).

$83 The value assigned to the label is placed on the top of the stack.

$84 The length attribute of the label is placed on the top of the stack.

$85 The type attribute of the label is placed on the top of the stack. (Type attributes are listed in the discussion of the GLOBAL record in the section "Segment Body" earlier in this chapter).

$86 The count attribute is placed on the top of the stack. The count attribute is 1 if the label is defined and 0 if it is not.

Note: The operand code $82 is referred to as the *weak reference*. The weak reference is an instruction to the linker that asks for the value of a label *if it exists*. It is not an error if the linker cannot find the label. However, the linker does not load a segment from a library if only weak references to it exist. If a label does not exist, a 0 is loaded onto the top of the stack. This operand is generally used for creating jump tables to library routines that may or may not be needed in a particular program.

Relative Offset Operand ($87): This operand is followed by a number that is treated as a displacement from the start of the segment. Its value is added to the value that the location counter had when the segment started, and the result is loaded on the top of the stack.

Example

Assume your assembly-language program contains the following line where MSG4 and MSG3 are global labels:

```
LDX     #MSG4-MSG3
```

This line would be assembled into two OMF records:

```
CONST ($01)     A2
EXPR ($EB)      02 : MSG4MSG3-
```

In hexadecimal format, these records appear as follows:

```
01 A2                                                    ."
EB 02 83 04 4D 53 47 34 83 04 4D 53 47 33 02 00      k...MSG4..MSG3..
```

The initial $01 is the OMF opcode for a 1-byte constant. The $A2 is the 65816 opcode for the LDX instruction. The $EB is the OMF opcode for an EXPR record, which is followed by a 1-byte count indicating the number of bytes to which the expression is to be truncated ($02 in this case). The next number, $83, is a label-reference operand for the first label in the expression, indicating that the value assigned to the label (MSG4) is to be placed on top of the evaluation stack. Next is a length byte ($04), followed by MSG4 spelled out in ASCII codes.

The next sequence of codes, starting with $83, places the value of MSG3 on the evaluation stack. Finally, the expression-operator code $02 indicates that a subtraction is to be performed, and the termination operator ($00) indicates the end of the expression.

Note: You can use the DumpOBJ utility program to examine the contents of any OMF file. DumpOBJ can list the header contents of each segment, and can list the body of each segment in OMF format, 65816 disassembly format, or as hexademical codes. DumpOBJ is described in the section "Command Descriptions" in Chapter 3.

Object Files

Object files (ProDOS 16 file type $B1) are created from source files by a compiler or assembler. Object files can contain any of the OMF record types except `INTERSEG`, `cINTERSEG`, `RELOC`, `cRELOC`, `SUPER`, and `ENTRY`. Object files can contain unresolved references, because all references are resolved by the linker. If you are writing a compiler for the Apple IIGS, you can use the `DUMPOBJ` utility to examine the contents of a variety of object files in order to get an idea of their content and structure.

Library Files

Library files (ProDOS 16 file type $B2) contain object segments that the linker can search for external references. Usually, these files contain general routines that can be used by more than one application. Any object segment that contains a global definition that was referenced during the link process is extracted from the library file; this segment is then added to the load segment that the linker is currently creating.

Library files differ from object files in that each library file includes a segment called the *library dictionary segment* (segment-type `KIND` = $08). The library dictionary segment contains the names and locations of all segments in the library file. This information allows the linker to scan the file quickly for needed segments. Library files are created from object files by the MakeLib utility program (described in Chapter 3). The format of the library dictionary segment is illustrated in Figure 7.4.

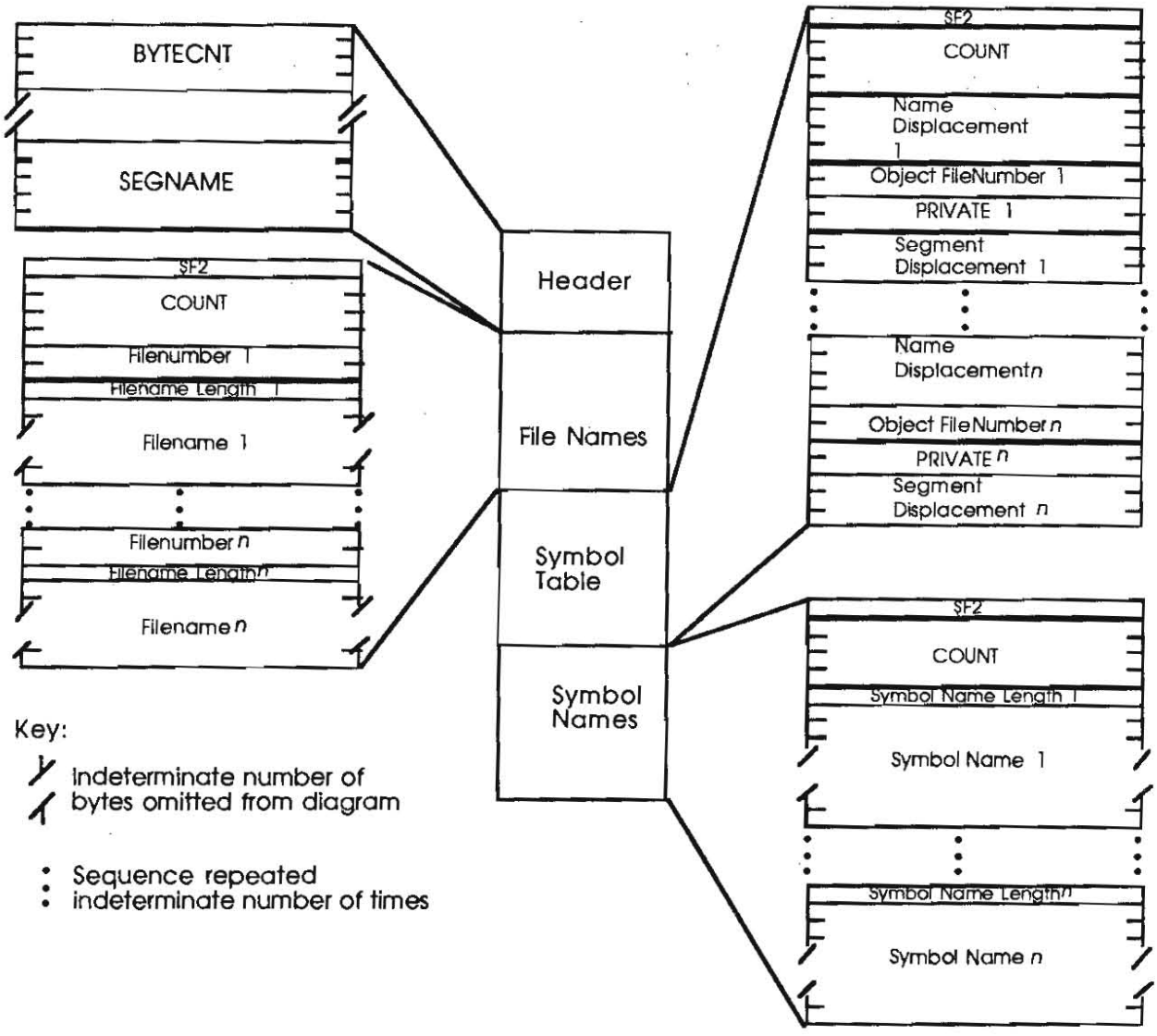


Figure 7.4. The Format of a Library Dictionary Segment

The library dictionary segment begins with a segment header, which is identical in form to other segment headers. The BYTECNT field indicates the number of bytes in the library dictionary segment, including the header. The body of the library dictionary segment consists of three LCONST records, as follows:

1. Filenames
2. Symbol Table
3. Symbol Names

The Filenames record consists of one or more subrecords, each consisting of a 2-byte file number followed by a filename. The filename is in Pascal-string format: that is, a length byte indicating the number of characters, followed by an ASCII string. The filenames are the full pathnames of the object files from which the segments in this library file were extracted. The file numbers are assigned by the MAKELIB program and used only within

the library file. These file numbers are not related to the load-file numbers in the pathname table.

The Symbol Table record consists of a cross-reference between the symbol names in the symbol-names record and the object segments in which the symbol names occur. For each global symbol in the library file, the Symbol Table record contains the following:

1. A 4-byte displacement into the Symbol Names record indicating the start of the symbol name.
2. The 2-byte file number of the file that the name occurred in. This is the file number assigned by the MakeLib utility and used in the Filenames record of this library dictionary segment.
3. A 2-byte flag, the `private` flag. If this flag equals 1, the symbol name is valid only in the object file in which it occurred (that is, it was in a private segment). If this flag equals 0, the symbol name is not private.
4. A 4-byte displacement into the library file indicating the beginning of the object segment in which the symbol occurs. The displacement is to the beginning of the segment even if the symbol occurs inside the segment; the location within the segment is resolved by the linker.

The Symbol Names record consists of a series of symbol names; each symbol name consists of a length byte followed by up to 255 ASCII characters. All global symbols that appear in an object segment, including entry points and global equates, are placed in the library dictionary segment. Duplicate symbols are not allowed.

Library dictionary segments are created by the MakeLib utility program,which also changes the file type of the file from $B1 to $B2 (see Chapter 3 for a discussion of the MakeLib utility).

Load Files

Load files (ProDOS 16 file types $B3 through $BE) contain the load segments that are moved into memory by the System Loader. They are created by the APW Linker from object files and library files. Load files conform to the object module format but are restricted to a small subset of that format. Because the segments must be quickly relocated and loaded, they cannot contain any unresolved symbolic information. This section discusses the following components of load files:

- The format of each load segment is a loadable binary memory image that is followed by a *relocation dictionary*. The memory image consists of long-constant (`LCONST`) records and define-storage (`DS`) records that can be located anywhere in memory. The relocation dictionary contains relocation (`RELOC`, `cRELOC`, or `SUPER RELOC`) records and intersegment (`INTERSEG`, `cINTERSEG`, or `SUPER INTERSEG`) records only. These records provide the information needed to modify the memory image according to its location in memory.
- The *jump table segment*, when used, is the segment of a load file that contains the calls to the System Loader to load dynamic segments. Each time the linker comes across a statement that references a label in a dynamic segment, it generates an entry in the jump table segment for that label (it also creates an entry in the relocation dictionary). The entry in the jump table segment contains the file number, segment number, and offset of the reference in the dynamic segment, plus a call to the System

Loader to load the segment. The relocation dictionary entry provides the information the loader needs to patch a call to the jump table segment into the memory image.

- The *pathname segment*, when used, is the segment of a load file that contains a cross-reference between file numbers and pathnames that the System Loader needs in order to reference load segments.
- An *initialization segment*, when used, is executed by the System Loader to perform any initialization required by the application.

The load segments in a load file are numbered by their relative location in the load file, where the first load segment is number 1. The segment number is used by the System Loader to find a specific segment in a load file.

Memory Image and Relocation Dictionary

Each load segment consists of two parts:

1. A memory image consisting of `LCONST` records and `DS` records containing all of the code and data that do not change with load address (with space reserved for location-dependent addresses). The `DS` records are inserted by the linker (in response to `DS` records in the object file) to reserve large blocks of space, rather than putting large blocks of zeros in the load file.
2. A relocation dictionary that provides the information necessary to patch the `LCONST` records at load time.

When the segment is loaded into memory, each `LCONST` record or `DS` record is loaded in one piece, and then the relocation dictionary is processed. The relocation dictionary includes `RELOC` (or `cRELOC` or `SUPER RELOC`) and `INTERSEG` (or `cINTERSEG` or `SUPER INTERSEG`) records only: the `RELOC` records provide the information necessary to recalculate the values of location-dependent local references, and the `INTERSEG` records provide the information necessary to transfer control to external references. See the discussions of the `RELOC` and `INTERSEG` records in the section "Segment Body" earlier in this chapter for more information. The sequence of events that occurs when a `JSL` to an external dynamic segment is executed is described in detail in the "System Loader" chapter of the *Apple IIGS ProDOS 16 Reference* manual.

Jump Table Segment

The jump table segment is a segment in a load file that is created by the linker to allow dynamic loading of code segments as they are needed during program execution. The segment type of the jump table segment is `KIND` = $02. There is one jump table segment per load file; it is a static segment, and it is loaded into memory at program boot time at a location determined by the Memory Manager at that time. The System Loader maintains a list, called the *jump table list* (or just the *jump table*), of the jump table segments in memory.

Each entry in the jump table segment corresponds to a call to an external (intersegment) routine in a dynamic segment. The jump table segment initially contains entries in the *unloaded* state. When the external call is encountered during program execution, a jump to the jump table segment occurs. The code in the jump table segment entry, in turn, jumps to the System Loader. The System Loader figures out which segment is referenced and loads

it. Next, the System Loader changes the entry in the jump table segment to the *loaded* state. The entry stays in the loaded state as long as the corresponding segment is in memory. If the application tells the System Loader to unload a segment, all jump table segment entries that reference that segment are changed to their unloaded states.

Unloaded State

The unloaded state of a jump table segment entry contains the code that calls the System Loader to load the needed segment. An entry contains the following fields:

User ID (2 bytes)
load-file number (2 bytes)
load-segment number (2 bytes)
load-segment offset (4 bytes)
JSL to jump-table load function (4 bytes)

The User ID field is reserved for the identification number assigned to the program by the UserID Manager; until initial load time, this field is 0. The load-file number, load-segment number, and load-segment offset refer to the location of the external reference. The rest of the entry is a call to the System Loader jump-table load function. The User ID and the address of the load function are patched by the System Loader during initial load. See the *Apple IIGS ProDOS 16 Reference* manual for information on the jump-table load function. A load-file number of 0 indicates that there are no more entries in this jump table segment (there may be other jump table segments for this program, however—each load file that is part of a program has its own jump table segment).

Loaded State

The loaded state of a jump-table segment entry is identical to the unloaded state except that the JSL to the System Loader jump-table load function is replaced by a JML to the external reference. A loaded entry contains the following fields:

User ID (2 bytes)
load-file number (2 bytes)
load-segment number (2 bytes)
load-segment offset (4 bytes)
JML to external reference (4 bytes)

Note: In Versions 1.0 and 2.0 of the OMF, the jump table segment starts with eight bytes of zeros. In future versions of the OMF, these zeros may be eliminated.

Pathname Segment

The pathname segment is a segment in a load file that is created by the linker to help the System Loader find the load segments of run-time library files that must be loaded dynamically. It provides a cross-reference between file numbers and file pathnames. The segment type of the pathname segment is KIND = $04. When the loader processes the load file, it adds the information in the pathname segment to the pathname table that it maintains in memory. Pathname tables are described in the *Apple IIGS ProDOS 16 Reference* manual.

The pathname segment contains one entry for each load file and run-time library file referenced in the load file. The format of each entry is as follows:

file number (2 bytes)
file date (2 bytes)
file time (2 bytes)
file pathname (length byte and ASCII string)

File number: A number assigned by the linker to a specific load file. File number 1 is reserved for the load file in which the pathname segment resides (usually the load file of the application program). A file number of 0 indicates that there are no more entries in this pathname segment.

File date and **file time:** ProDOS 16 directory items retrieved by the linker during the link process. The System Loader compares these values with the ProDOS 16 directory of the run-time library file at run time. If they are not the same, the System Loader does not load the requested load segment, thus ensuring that the run-time library file used at link time is the same as the one loaded at execution time.

File pathname: The pathname of the load file. The pathname is listed as a Pascal-type string: that is, a length byte followed by an ASCII string. A pathname segment created by the linker may contain partial pathnames. A partial pathname begins with one of the eight prefixes supported by ProDOS 16; these prefixes have the form *n*/, where *n* is a number from 0 to 7. The first three prefixes have fixed definitions, as follows:

0/ system prefix (initially the volume from which ProDOS 16 was booted)

1/ application subdirectory (the subdirectory out of which the application is running)

2/ system library subdirectory (initially */boot_volume*/SYSTEM/LIBS/)

ProDOS 16 prefixes are described in the *Apple IIGS ProDOS 16 Reference* manual.

Important: Currently, run-time library files and multiple load files are not supported by the linker. The pathname table is created, but it contains only one pathname—that of the single load file.

Initialization Segment

The initialization segment is an optional segment in a load file. When the System Loader encounters an initialization segment during the initial loading of segments, it transfers control to the initialization segment. After the initialization segment returns control to the System Loader, the loader continues the normal initial load of the remaining segments in the load file. The segment type of the initialization segment is KIND = $10.

One way in which the initialization segment might be used is to initialize the graphics environment of an application and to display a "splash screen" (such as a copyright message and company logo) for the duration of the program load.

The initialization segment must obey the following rules:

- It must not reference any segments not yet loaded.

- It must exit with an RTL instruction.
- It must not change the stack pointer.
- It must not use the current direct page. To avoid writing over a portion of the direct page being used by the loader, the initialization segment must allocate its own direct page if it needs direct-page space.

Note: Initialization segments are reexecuted during a restart of an application from memory.

Direct-Page/Stack Segments

The Apple IIGS stack can be located anywhere in the lower 48K of bank $00 and can be any size up to 48K. The direct page is the Apple IIGS equivalent of the zero page of 8-bit Apple II's; the direct page can also be located anywhere in the lower 48K of bank $00. Like the zero page, the direct page occupies 256 bytes of memory; on the Apple IIGS, however, a program can move its direct page while it is running. Consequently, a given program can use more than 256 bytes of memory for direct-page functions.

Each program running on the Apple IIGS reserves a portion of bank $00 as a combined direct-page/stack space. Since more than one application can be loaded in memory at one time on the Apple IIGS, there may be more than one stack and one direct page in bank $00 at a given time. Furthermore, some applications may place some of their code in bank $00. A given program should therefore probably not use more than about 4K for its direct-page/stack space.

When an instruction uses one of the direct-page addressing modes, the effective address is calculated by adding the value of the operand of the instruction to the value in the direct-page register. The stack pointer, on the other hand, is decremented each time a stack-push instruction is executed. The convention used on the Apple IIGS, therefore, is for the direct page to occupy the lower part of the direct-page/stack space, while the stack grows downward from the top of the space.

Important: ProDOS 16 provides no mechanism for detecting stack overflow or underflow, or collision of the stack with the direct page. Your program must be carefully designed to make sure those conditions cannot occur.

If you do not define a direct-page/stack segment in your program, ProDOS 16 assigns a 1024-byte direct page/stack when the System Loader INITIAL LOAD or RESTART call is executed. To specify the size and contents of the direct-page/stack space, use the following procedure:

1. Create a data segment in your source file with the size and contents you want for your initial direct page and stack. Start the segment with a DATA directive, use DS and DC directives to define the contents of the segment, and end it with an END statement.
2. Assemble the program.
3. Use a LinkEd file to link the program. Place the direct-page/stack segment in a load segment by itself, and specify the segment-type KIND=$12 for the segment. For example, suppose you have created the data segment DEFPAGE, and assembled it so that it is now in the object file MYOBJ.A. To make that segment a direct page/stack

segment with the load-segment name DIRSTACK in the load file MYPROG, use the following LinkEd commands:

```
KEEP MYPROG
SEGMENT/$12 DIRSTACK
  SELECT MYOBJ.A (DEFPAGE)
    •
    •
    •
```

LinkEd is described in Chapter 5.

Run-Time Library Files

Run-time library files (ProDOS 16 file type $B4) contain dynamic load segments that the System Loader can load when these segments are referenced through the jump table. Usually, run-time library files contain general routines that can be used by more than one application.

Run-time library files are scanned by the linker during the link process. When the linker finds a referenced segment in the run-time library file, it generates an INTERSEG reference to the segment in the relocation dictionary and adds an entry to the jump table segment for that file. It does *not* extract the segment from the file and place it in the file that referenced it, as it does for ordinary library files. In other words, references to segments in run-time library files are treated by the linker like references to any other dynamic segments.

The last load segment of the run-time library file contains all the information the linker needs in order to find referenced segments; it is not necessary for the linker to scan through every subroutine in every segment each time a subroutine is referenced. The last segment contains a table of ENTRY records, each one corresponding to a segment name or global reference in the run-time library file.

Run-time library files are created from corresponding object files. When you create a run-time library file, you specify the location of the source file and the pathname at which the run-time library file will be located at load time. The location of the run-time library file is stored in the pathname segment in the load file of the application program. At load time, the run-time library file must reside in the specified subdirectory.

Currently, run-time library files are not supported by the linker. This specification is provided to allow for future enhancements to the system.

Shell Load Files

Shell load files (ProDOS 16 file type $B5) are executable load files that are run under a shell program, such as the APW Shell. The shell calls the System Loader's Initial Load function and transfers control to the shell load file by means of a JSL instruction, rather than launching the program through the ProDOS 16 QUIT function. Therefore, the shell does not shut down, and the program can use shell facilities during execution. The program returns control to the shell with an RTL, or with a ProDOS 16 QUIT call if the shell intercepts and acts on ProDOS 16 calls. (The APW Shell is an example of a shell that

intercepts ProDOS QUIT calls.) Shell load files should use standard Text Tool Set calls for all nongraphics I/O. The shell program is responsible for initializing the Text Tool Set routines.

> **Note:** A load file of file type $B5 can be launched by ProDOS 16 via the QUIT call if it requires no support other than standard input from the keyboard and output to the screen. ProDOS 16 initializes the Text Tool Set to use the Pascal I/O drivers (see the *Apple IIGS Toolbox Reference*) for the keyboard and 80-column screen. Only $B5 files that end in a ProDOS 16 QUIT call can be run in this way.

As soon as a shell load file is launched, it should check the X and Y registers for a pointer to the shell-identifier string and input line. The X register holds the high word and the Y register holds the low word of this pointer. The shell program is responsible for loading this pointer into the index registers and for placing the following information in the area pointed to:

1. An 8-byte ASCII string containing an identifier for the shell (the identifier for the APW Shell, for example, is BYTEWRKS). The shell load file should check this identifier to make sure that it has been launched by the correct shell, so that the environment it needs is in place. If the shell identifier is not correct, the shell load file should write an error message to standard error output (normally the screen) and then exit with an RTL instruction (or a ProDOS QUIT call if the shell intercepts ProDOS calls).
2. A null-terminated ASCII string containing the input line for the shell load file. The shell program can strip any I/O redirection or pipeline commands from the input line, since those commands are intended for the shell itself, but must pass on all input parameters intended for the shell load file.

The shell program must request a User ID for the shell load file; the User ID is passed in the accumulator. The shell must set up a direct-page and stack area for the shell load file. The shell places the address of the start of the direct-page/stack space in the direct-page (D) register and sets the stack pointer (S register) to point to the last byte of the block. If the shell application does not have a direct-page/stack segment, the shell should follow the same conventions used by ProDOS 16 for default direct-page/stack allocation. See the section "Direct-Page/Stack Segments" in this chapter and the *Apple IIGS ProDOS 16 Reference* manual for more information on direct-page and stack allocation.

> **Note:** ProDOS 16 does not support the identifier string or input line. If the shell load file is launched by ProDOS 16, the X and Y registers contain zeros.

Some shell load files may launch other programs; for example, a shell nested within another shell would be a shell load file. When a shell load file requests a User ID for a program, the calling program is responsible for intercepting ProDOS QUIT calls and system resets, so that it can remove from memory all memory buffers with that User ID before passing control to the shell.

A shell load file should use the following procedure to quit:

1. If the shell load file has launched any programs, it must call the System Loader's User Shutdown function to shut down those programs.
2. The shell load file should release any memory buffers that it has requested and dispose of their handles.

3. The shell load file must place an error code in the accumulator. If no error occurred, the error code should be $0000. The error code $FFFF is used as a general (nonspecific) error code. You can define any other error codes you want to use for a shell program you write and can handle them in any way you wish.
4. The shell load file should execute an RTL or a ProDOS 16 QUIT call. If the program ends in a QUIT call, the shell program that launched the shell load file is responsible for intercepting the QUIT call, releasing all memory buffers associated with that shell load file, and performing any other system tasks normally done by ProDOS 16 in response to a QUIT.

Important: When a shell launches a shell load file, the address of the shell program is not pushed onto the ProDOS 16 QUIT stack; therefore the shell must handle the shell load file's QUIT call itself, or control is not returned to the shell. In order to do this, the shell program must intercept *all* ProDOS 16 calls. The shell may pass any other ProDOS 16 calls on to ProDOS, but it must handle QUIT calls itself. If the shell you are using does not handle ProDOS 16 QUIT calls in this fashion, the shell load file must end in an RTL.

Chapter 8

Shell Calls

The Apple IIGS Progammer's Workshop Shell acts as an interface and extension to ProDOS 16. The shell provides several functions not provided by ProDOS 16; these functions are called exactly like ProDOS 16 functions. Every time a program running under the APW Shell issues a ProDOS-16-like call, the shell intercepts the call; if it is a shell call, the shell interprets it and acts on it. If it is a ProDOS 16 call, the shell passes it on to ProDOS 16. This chapter describes all of the shell's ProDOS-16-like calls, here referred to as *shell calls*.

The shell calls that are provided are listed in Table 8.1 in the order of their call numbers. The calls are described in alphabetical order in the section "Call Descriptions" later in this chapter.

Table 8.1. Summary of Shell Calls

Call Name	Call Number	Use
GET_LINFO	($0101)	Passes parameters from the shell to a program
SET_LINFO	($0102)	Passes parameters from a program to the shell
GET_LANG	($0103)	Reads the current language number
SET_LANG	($0104)	Sets the current language number
ERROR	($0105)	Prints error message for an Apple IIGS tool call
SET_VAR	($0106)	Sets the value of a shell variable
VERSION	($0107)	Returns the version number of the APW Shell
READ_INDEXED	($0108)	Reads variable table
INIT_WILDCARD	($0109)	Provides a filename that includes a wildcard character to the shell
NEXT_WILDCARD	($010A)	Causes the shell to find the next filename that matches the wildcard filename
GET_VAR	($010B)	Reads the value of a shell variable
EXECUTE	($010D)	Sends a command or list of commands to the shell command interpreter
DIRECTION	($010F)	Indicates whether I/O redirection has occurred
REDIRECT	($0110)	Sets device and file for I/O redirection
STOP	($0113)	Detects a request for an early termination of the program
WRITE_CONSOLE	($011A)	Sends output to the console

Warning: Call numbers $0100 through $01FF are reserved. Be careful to use only the call numbers documented here. Making calls to other, undocumented call numbers may have unpredicatable results.

Making a Shell Call

An assembly-language calling program makes a shell call by executing a set of instructions and directives referred to as a **shell-call block**. The shell-call block contains a pointer to a **parameter block**. The parameter block is used for passing information between the calling program and the shell. Each APW language provides an easy way to execute shell calls; in APW Assembly Language, for example, the shell-call block is normally executed by an assembler macro. The following sections discuss these aspects of shell calls.

Note: Although shell calls are made exactly like ProDOS 16 calls, this section does not provide all of the information relevant to making ProDOS 16 calls. ProDOS 16 calls are described in the *Apple IIGS ProDOS 16 Reference* manual.

This chapter assumes that you are using the APW Assembler to make shell calls. See the *Apple IIGS Programmer's Workshop Assembler Reference* for more information on the APW Assembler. To access shell calls from a program written in another language, see the manual that came with the language.

The Call Block

A shell-call block consists of a JSL to the ProDOS 16 entry point, followed by a 2-byte system call number and a 4-byte parameter block pointer. The APW Shell intercepts the call and determines whether it is an APW Shell call or ProDOS 16 call. If a shell call, it performs the requested function, if possible, and returns execution to the instruction immediately following the call block. If a ProDOS 16 call, the shell passes it on to ProDOS 16.

When making the call, the the processor should be in full native mode. The call block looks like this:

```
            JSL   PRODOS            ; Dispatch call to ProDOS 16 entry
            DC    I2'CALLNUM'       ; 2-byte call number
            DC    I4'PARMBLOCK'     ; 4-byte parameter block pointer
            BCS   ERROR             ; If carry set, go to error handler
                                    ; otherwise, continue. . .
            .
            .
            .
ERROR                               ; error handler
            .
            .
            .
PARMBLOCK                           ; parameter block
```

The call block itself consists of only the JSL instruction and the DC assembler directives. The BCS instruction in this example is a conditional branch to an error handler called ERROR.

Shell-Call Macros

For each call listed in Table 9-1, there is an APW Assembler macro that you can use to make the call. The macro call consists of the name of the call (as shown in Table 9-1), with the address of the parameter block in the operand field. For example, to call the GET_LINFO function, use the following sequence:

```
            MCOPY     2/AINCLUDE/M16.SHELL  ; Make the macro file available
            .
            .
            .
            GET_LINFO PARMBLOCK             ; The macro call
            BCS       ERROR                 ; If carry set, go to error handler
                                            ; otherwise, continue. . .
            .
            .
            .
ERROR                                       ; error handler
            .
            .
            .
PARMBLOCK                                   ; parameter block
```

The Parameter Block

A parameter block is a specifically formatted table that occupies a set of contiguous bytes in memory. It consists of a number of fields that hold information that the calling program supplies to the shell, as well as information returned by the shell to the caller.

Every shell call requires a valid parameter block (PARMBLOCK in the above examples), referenced by a 4-byte pointer in the call block or by the operand of the macro call. You are responsible for constructing the parameter block for each call you make; the block may be anywhere in memory. Formats for individual parameter blocks accompany the detailed system call descriptions in this chapter.

Types of Parameters

Each field in a parameter block contains a single parameter. There are three types of parameters used by the shell: values, results, and pointers. Each is either an input to the shell from the caller or an output from the shell to the caller.

- A **value** is a numeric quantity, one or more bytes long, that the caller passes to the shell through the parameter block. It is an input parameter.
- A **result** is a numeric quantity, one or more bytes long, that the shell places into the parameter block for the caller to use. It is an output parameter.
- A **pointer** is the 4-byte address of a location containing data, code, an address, or buffer space in which the shell can receive or place data. The pointer itself is an input; that is, you always provide the pointer and reserve space for the data. The data pointed to may be either input by your program, returned by the shell, or both.

A given parameter may be both a value and a result.

Important: Unless noted otherwise, each string in a parameter block or pointed to by a parameter block consists of a length byte, which is a binary number indicating the number of characters in the string, followed by ASCII characters.

Setting Up a Parameter Block in Memory

Each APW Shell call references a parameter block, which may be anywhere in memory. Because all applications must obtain needed memory from the Memory Manager, an application cannot know in advance where the memory segment holding such a parameter block will be.

There are two ways to set up a parameter block in memory. Either

1. Code the block directly into the program, referencing it with a label. The parameter block will always have the same relative location in the program code.

or

2. Use Memory Manager and System Loader calls to place the block in memory.

The first method is by far the simplest and most typical way to do it. For instructions on using the second method, see the *Apple IIGS ProDOS 16 Reference* manual.

Register Values

There are no register requirements on entry to a shell call. The APW Shell saves and restores all registers except the accumulator (A) and the processor status register (P); those two registers store information on the success or failure of the call. On exit, the registers have the following values:

A	zero if the call is successful; if nonzero, the number is the error code
X	unchanged
Y	unchanged
S	unchanged
D	unchanged
P	(see below)
DB	unchanged
PB	unchanged
PC	address of location following the parameter block pointer

Unchanged means that APW initially saves, and then restores when finished, the value the register had just before the shell call.

On exit, the processor status register (P) bits are

n	undefined
v	undefined
m	unchanged
x	unchanged
d	zero
i	unchanged
z	undefined
c	zero if the call is successfull, 1 if not
e	zero

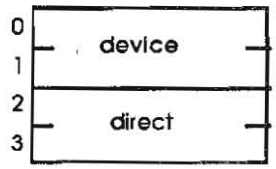
Call Descriptions

This section describes each call, including its use and the contents of its parameter block. The possible errors returned by a call are listed at the end of each call description. The calls are listed here in alphabetical order. Table 8.1 lists all the calls in order of their call numbers.

DIRECTION ($010F)

A program can use this function to find out whether command-line I/O redirection has occurred. This information can be used by a program, for example, to determine whether to send form feeds to standard output.

Parameter Block:



Offset	Label	Description
$00–$01	`device`	**parameter name:** device number **size and type:** 2-byte value **range of values:** $0000–$0002 This parameter indicates which type of input or output you are inquiring about, as follows: $0000 standard input $0001 standard output $0002 error output
$02–$03	`direct`	**parameter name:** direction **size and type:** 2-byte result **range of values:** $0000–$0002

This parameter indicates the type of redirection that has occurred, as follows:

$0000	console (default)
$0001	printer
$0002	disk file

Possible Errors

$53	Parameter out of range

ERROR ($0105)

When an Apple IIGS tool call returns an error, your program can use this function to print out the name of the tool and the appropriate error message. This function makes it unneccessary for your program to store a complete table of error messages for tool calls. The error number is placed in the accumulator by the tool; you need only store the accumulator value in the parameter block and execute this call to print the error message to standard error output.

Parameter Block:

Offset	Label	Description
$00–$01	`error`	**parameter name:** error number **size and type:** 2-byte value **range of values:** $0000–$FFFF This parameter specifies the error number returned by the tool call.

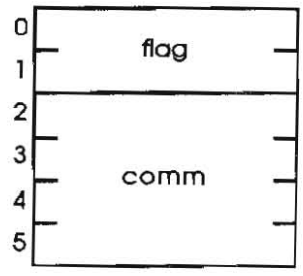
Possible Errors

None

EXECUTE ($010D)

This function sends a command or list of commands to the APW Shell command interpreter.

Parameter Block:



Offset	Label	Description
$00–$01	`flag`	**parameter name:** echo command flag **size and type:** 2-byte value **range of values:** $0000 or $8000 If you set the most significant bit of this flag to 1 (binary), a new variable table is not defined when the commands are executed. Setting this flag is similar to executing an Exec file with an `EXECUTE` command: if no new variable table is defined, the variables defined by the list of commands modify the current variable table. If this flag is set to $0000, a new variable table is defined for the list of commands being executed; the current variable table is not modified. Exec files, variables, and the `EXECUTE` command are described in the section "Exec Files" in Chapter 3.
$02–$05	`comm`	**parameter name:** address of command string **size and type:** 4-byte pointer **range of values:** $0000 0000–$00FF FFFF The address of the buffer in which you place the commands. If you include more than one command, separate the commands with semicolons (`;`) or carriage return characters ($0D). The last ASCII character in the command string must be a carriage return. The command string has no length byte; terminate the command string with a null character ($00). Any output is sent to standard output. If the shell variable `{Exit}` is not null and any command returns a nonzero error code, any remaining commands are ignored. Error codes and variables are described in the section "Exec Files" in Chapter 3.

Possible Errors

Any error returned from the last command or program executed by the list of commands executed.

GET_LANG ($0103)

This function reads the current language number. Language numbers are described in the section "Command Types and the Command Table" in Chapter 3 and are listed in Appendix B.

Parameter Block:

Offset	Label	Description
$00-$01	`lang`	**parameter name:** language number **size and type:** 2-byte result **range of values:** $0000–$7FFF This parameter specifies the current APW language number. The current language number is set by the APW Editor when it opens an existing file or by the user with an APW Shell command.

Possible Errors

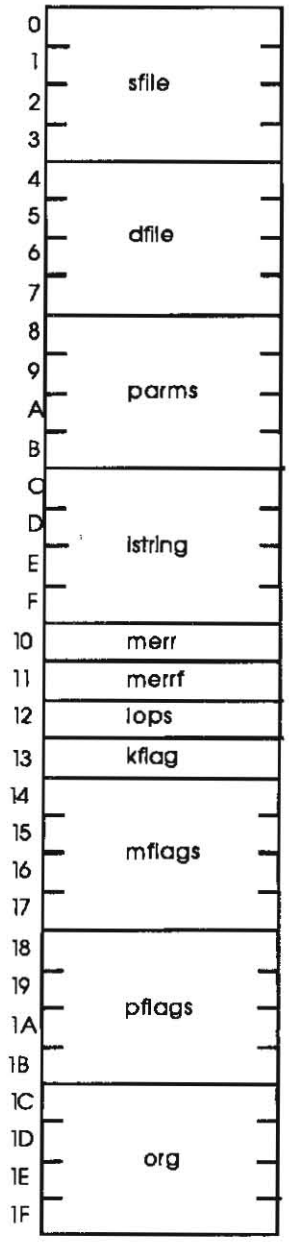
None

GET_LINFO ($0101)

This function is used by an assembler, compiler, linker, or editor to read the parameters that are passed to it. When you make this call, you reserve the specified amount of space for each parameter in the parameter block; then when the APW Shell returns control to your program, you can read the parameter block to obtain the information you need.

Use the `SET_LINFO` call when your program finishes before executing an `RTL` to return control to the shell.

Parameter Block:



Offset	Label	Description
$00–$03	`sfile`	**parameter name:** address of source filename **size and type:** 4-byte pointer **range of values:** $0000 0000–$00FF FFFF The address of a 65-byte buffer into which the shell will put the filename of the source file: that is, the file that the compiler or assembler is to process. The filename can be any valid ProDOS 16 filename and can be either a partial or full pathname. If the `+E` flag is set and the compiler exits due to an error, the compiler places the pathname of the source file in which the error occurred into a buffer and sets the `sfile` parameter in the `SET_LINFO` call to point to that buffer. An editor can then use this pathname to open the source file and display it on the screen.
$04–$07	`dfile`	**parameter name:** address of output filename **size and type:** 4-byte pointer **range of values:** $0000 0000–$00FF FFFF The address of a 65-byte buffer into which the shell puts the filename of the output file (if any): that is, the file that the compiler or assembler writes to. The filename can be any valid ProDOS 16 filename and can be either a partial or full pathname.
$08–0B	`parms`	**parameter name:** address of parameter list **size and type:** 4-byte pointer **range of values:** $0000 0000–$00FF FFFF The address of a 256-byte buffer into which the shell puts the list of names from the `NAMES` parameter list in the APW Shell command that called the assembler or compiler. If there was no `NAMES` parameter list, the buffer pointed to by `parms` begins with the length byte $00. If the `+E` flag is set and the compiler exits due to an error, the compiler places the text of the error message into a buffer and set the `parms` parameter in the `SET_LINFO` call to point to that buffer. An editor can then display the error message at the bottom of the screen.
$0C–$0F	`istring`	**parameter name:** address of input strings **size and type:** 4-byte result **range of values:** $0000 0000–$00FF FFFF The address of a 256-byte buffer into which the shell puts the string of commands to be passed on to a specific language compiler. For example, if the `COMPILE` command includes the parameter `CC=(-I/CINCLUDES/)`, the string enclosed in parentheses is found in that buffer when the C compiler is called.

\$10 `merr` **parameter name:** maximum error level allowed
size and type: 1-byte result
range of values: \$00–\$10

If the maximum error level found by the assembler, compiler, or linker (`merrf`) is greater than `merr`, the APW Shell does not call the next program in the processing sequence. For example, if you use the `ASML` command to assemble and link a program, but the assembler finds an error level of 8 when `merr` equals 2, then the linker is not called when the assembly is complete.

\$11 `merrf` **parameter name:** maximum error level found
size and type: 1-byte result
range of values: \$00–\$FF

This field is used by the `SET_LINFO` call to return the maximum error level found. In the case of a multilanguage compile, this field contains the error level returned by the last compiler. The shell sets this field to \$00 before the first compile.

\$12 `lops` **parameter name:** operations flags
size and type: 1-byte result
range of values: \$00–\$10

This field is used to keep track of the operations that are to be performed by the system. The format of this byte is

Bit:	7	6	5	4	3	2	1	0
Value:	0	0	0	0	0	E	L	C

where C = Compile
L = Link
E = Execute

When a bit is set (to 1), the indicated operation is to be done. For example, the `COMPILE` command sets bit 0, while the `CMPLG` command sets bits 0, 1, and 2. When a compiler finishes its operation and returns control to the APW Shell, it clears bit 0 unless a file with another language is appended to the source.

\$13 `kflag` **parameter name:** keep flag
size and type: 1-byte result
range of values: \$00–\$03

This flag indicates what should be done with the output of a compiler, assembler, or linker, as follows:

Kflag Value	Meaning
$00	Do not save output.
$01	Save to an object file with the root filename pointed to by `dfile`. For example, if the output filename pointed to by `dfile` is `PROG`, the first segment to be executed should be put in `PROG` or `PROG.ROOT` and the remaining segments should be put in `PROG.A`. For linkers, save to a load file with the name pointed to by `dfile` (for example, `PROG`).
$02	The `.ROOT` file has already been created (by another language compiler, for example). In this case, the first file created by the compiler or assembler should end in the `.A` extension.
$03	At least one alphabetic suffix has already been used. In this case, the compiler or assembler must search the directory for the highest alphabetic suffix that has been used, and then use the next one. For example, if `PROG.ROOT`, `PROG.A`, and `PROG.B` already exist, the compiler should put its output in `PROG.C`.

See the section "Compilers and Assemblers" in Chapter 6 for more information on object-file naming conventions.

$14–$17 `mflags`

parameter name: flags with a minus sign
size and type: 4-byte result
range of values: binary string

This parameter passes command-line-option flags such as `-L` or `-C`. The first 26 bits of these four bytes represent the letters A through Z, arranged with A as the most significant bit of the most significant byte. The bytes are ordered least significant byte first. The bit map is as follows:

```
11000000 11111111 11111111 11111111
YZ       QRSTUVWX IJKLMNOP ABCDEFGH
```

For each flag set with a minus sign in the command, the corresponding bit in this parameter is set to 1. See the discussions of the `ALINK` and `ASML` commands in Chapter 3 for descriptions of these option flags.

$18–$1B `pflags`

parameter name: flags with a plus sign
size and type: 4-byte result
range of values: binary string

This parameter passes command-line-option flags such as +L or +C. The first 26 bits of these four bytes represent the letters A through Z; the bit map for this parameter is the same as for the `mflags` parameter. See the discussions of the `ALINK` and `ASML` commands in Chapter 3 for descriptions of these option flags.

$1C–$1F `org`

parameter name: origin
size and type: 4-byte result
range of values: $0000 0000–$FFFF FFFF

This parameter specifies the absolute start address of a nonrelocatable load file, if one has been specified. The origin is used only by a linker. If the +E flag is set and the compiler exits due to an error, the compiler puts the offset of the line containing the error into the `org` field of the `SET_LINFO` call. An editor can then place that line on the fifth line of the screen.

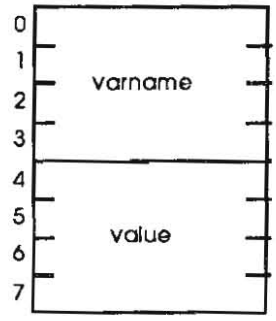
Possible Errors

None

GET_VAR ($010B)

This function reads the string associated with a variable (that is, the value of the variable). The value returned is the one valid for the currently executing Exec file, or for the interactive command interpreter (if that is the command level in use). Variables and Exec files are described in the section "Exec Files" in Chapter 3. Use the SET_VAR call to set the value of a variable.

Parameter Block:



Offset	Label	Description
$00–$03	varname	**parameter name:** pointer to name of variable **size and type:** 4-byte pointer **range of values:** $0000 0000–$00FF FFFF This is a pointer to a buffer that contains the name of the variable whose value you wish to read. The variable name consists of a length byte and a string of up to 255 ASCII characters.
$04–$07	value	**parameter name:** pointer to value of variable **size and type:** 4-byte pointer **range of values:** $0000 0000–$00FF FFFF This is a pointer to a 256-byte buffer into which the shell places the value of the variable. The value consists of a length byte and a string of ASCII characters. For an undefined variable, the value consists of a null string (that is, the length byte is $00).

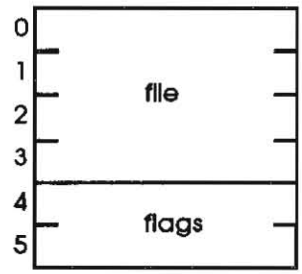
Possible Errors

None

INIT_WILDCARD ($0109)

This function provides to the APW Shell a filename that can include a wildcard character. The shell can then search for filenames matching the filename you specified when it receives a `NEXT_WILDCARD` command. This function accepts any filename, whether it includes a wildcard or not, and expands device names (such as .D1), prefix numbers, and the double-period (..) before the filename is passed on to ProDOS 16. Therefore, you should call this function every time you want to search for a filename. Doing so will ensure that your routine supports all of the conventions for partial pathnames that the user expects from APW.

Parameter Block:



Offset	Label	Description

$00–$03	`file`	**parameter name:** address of pathname
		size and type: 4-byte pointer
		range of values: $0000 0000–$00FF FFFF
		
		This parameter specifies the address of a buffer containing a pathname or partial pathname that can include a wildcard character. Examples of such pathnames are
		
		`A=`
		`/APW/MYPROGS/?.ROOT`
		`.D2/HELLO`
		
		When you execute a `NEXT_WILDCARD` call, the shell finds the next filename that matches the filename pointed to by `file`. If the wildcard character you specified was a question mark (?), the filename is written to standard output and you are prompted for confirmation before the file is acted on or the next filename is found. The use of wildcard characters is described in the section "Using Wildcard Characters" in Chapter 2.

$04–$05	`flags`	**parameter name:** prompting flags **size and type:** 2-byte value **range of values:** $0000, $4000, $8000 or $C000 If the most significant bit is set, prompting is not allowed; that is a question mark (?) is treated as if it were an equal sign (=). If the next-most significant bit is set and prompting is being used, only the first choice accepted by the user (that is, the first choice

for which the user types a `Y` in response to the prompt) is acted on. The second flag is for use with commands that can act on only one file, such as `RENAME` or `EDIT`.

Possible Errors

Errors for the following ProDOS 16 and Memory Manager calls. See the *Apple IIGS ProDOS 16 Reference* manual and the *Apple IIGS Toolbox Reference* manual for descriptions of these errors.

Open
Read
Close
Dispose
Get info
Get end of file
Lock
Allocate new memory

NEXT_WILDCARD ($010A)

Once a filename that includes a wildcard has been suppled to the shell with an `INIT_WILDCARD` call, the `NEXT_WILDCARD` call causes the shell to find the next filename in the directory that matches the wildcard pathname. For example, if the wildcard pathname specified in `INIT_WILDCARD` were `/APW/LIBRARIES/AINCLUDE/M16.?`, then the first pathname returned by the shell in response to a `NEXT_WILDCARD` call might be `/APW/LIBRARIES/AINCLUDE/M16.UTIL`.

Parameter Block:

Offset	Label	Description
$00–$03	`nextfile`	**parameter name:** address of next filename **size and type:** 4-byte pointer **range of values:** $0000 0000–$00FF FFFF This parameter specifies the address of the buffer to which the shell has returned the next filename that matches a wildcard filename. The wildcard filename is the last one specified with an `INIT_WILDCARD` call. If there are no more matching filenames, or if `INIT_WILDCARD` has not been called, then the shell returns a null string (that is, a string with a length of zero). See also the description of `INIT_WILDCARD`.

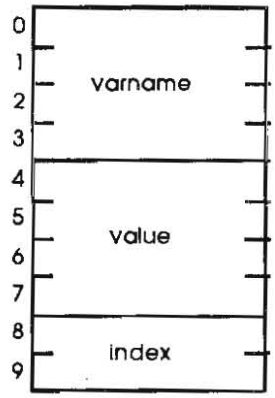
Possible Errors

None

READ_INDEXED ($0108)

You can use this function to read the contents of the variable table for the command level at which the call is made. To read the entire contents of the variable table, you must repeat this call, incrementing the index number by 1 each time, until the entire contents have been returned.

Parameter Block:



Offset	Label	Description
$00–$03	`varname`	**parameter name:** pointer to name of variable **size and type:** 4-byte pointer **range of values:** $0000 0000–$00FF FFFF This is a pointer to a 256-byte buffer into which the shell is to place the name of the next variable in the variable table. The variable name consists of a length byte and a string of ASCII characters. A null string is returned when the index number exceeds the number of variables in the variable table.
$04–$07	`value`	**parameter name:** pointer to value of variable **size and type:** 4-byte pointer **range of values:** $0000 0000–$00FF FFFF This is a pointer to a 256-byte buffer into which the shell is to place the value of the variable. The value consists of a length byte and a string of ASCII characters. For an undefined variable, the value consists of a null string (that is, the length byte is $00).
$08–$09	`index`	**parameter name:** index number **size and type:** 2-byte value **range of values:** $0000–$FFFF This is an index number that you provide. Start with $01 and increment the number by 1 with each successive `READ_INDEXED` call until there are no more values in the variable table.

Possible Errors

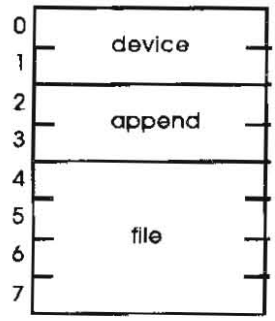
Errors for the following Memory Manager calls. See the *Apple IIGS Toolbox Reference* manual for descriptions of these errors.

Lock
Unlock

REDIRECT ($0110)

This function instructs the shell to redirect input or output to the printer, console, or a disk file.

Parameter Block:



Offset	Label	Description
$00–$01	device	**parameter name:** device number **size and type:** 2-byte value **range of values:** $0000–$0002 This parameter indicates which type of input or output you wish to redirect, as follows: $0000 standard input $0001 standard output $0002 error output
$02–$03	append	**parameter name:** append flag **size and type:** 2-byte value **range of values:** $0000–$FFFF This flag indicates whether redirected output should be appended to an existing file with the same filename, or the existing file should be deleted first. If append is 0, the file is deleted; if it is any other value, the output is appended to the file.
$04–$07	file	**parameter name:** address of filename **size and type:** 4-byte pointer **range of values:** $0000 0000–$00FF FFFF This parameter specifies the address of a 65-byte-long buffer containing the filename of the file to or from which output is to be redirected. The filename can be any valid ProDOS 16 filename, a partial or full pathname, or the device names .PRINTER or .CONSOLE.

Possible Errors

$53	Parameter out of range

Errors for the following ProDOS 16 calls. See the *Apple IIGS ProDOS 16 Reference* manual and the *Apple IIGS Toolbox Reference* manual for descriptions of these errors.

Open
Close
Read
Write
Get end of file

SET_LANG ($0104)

This function sets the current language number. Language numbers are described in the section "Command Types and the Command Table" in Chapter 3 and are listed in Appendix B.

Parameter Block:

Offset	Label	Description
$00-$01	`lang`	**parameter name:** language number **size and type:** 2-byte value **range of values:** $0000–$7FFF This parameter specifies the APW language number to which the current APW language should be set. If the language specified is not installed (that is, not listed in the command table), the "Language not available" error is returned.

Possible Errors

$80	Language not available

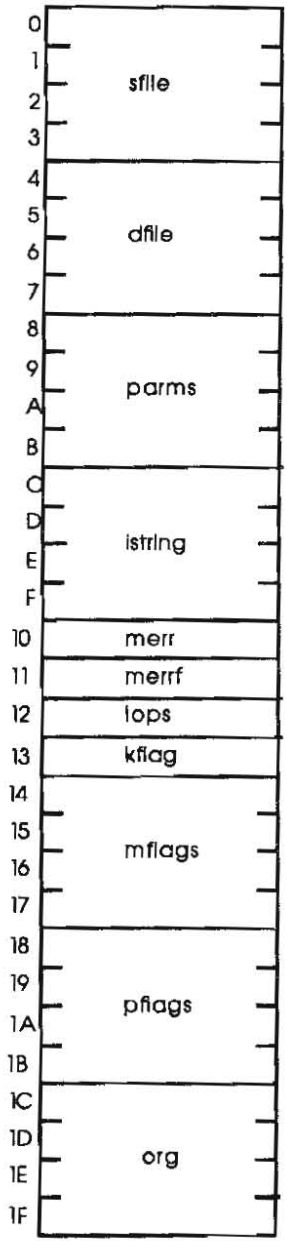
SET_LINFO ($0102)

This function is used by an assembler, compiler, linker, or editor to pass parameters to the APW Shell before returning control to the shell. It can also be used by a shell program under which you are running APW to pass parameters to the APW Shell.

Use the GET_LINFO call to read parameters passed to your assembler, compiler, linker, or editor.

> **Important:** Memory buffers pointed to by parameters in the SET_LINFO parameter block must be in static segments that are loaded when your program is launched. The APW Shell does not unload your program's static segments until after it has processed the SET_LINFO call.

Parameter Block:



Offset	Label	Description
$00–$03	`sfile`	**parameter name:** address of source filename **size and type:** 4-byte pointer **range of values:** $0000 0000–$00FF FFFF This parameter specifies the address of a buffer into which the compiler has placed the pathname of the next source file, if any: that is, the next file that a compiler or assembler is to process. Your compiler may have obtained this pathname from an APPEND directive, for example. The filename can be any valid ProDOS 16 filename and either a partial or full pathname.

If the +E flag is set and the compiler exits due to an error, the compiler should place the pathname of the source file in which the error occurred into a buffer and set the `sfile` parameter to point to that buffer. The editor uses this pathname to open the source file and display it on the screen.

$04–$07 `dfile`

parameter name: address of output filename
size and type: 4-byte pointer
range of values: $0000 0000–$00FF FFFF

This parameter specifies the address of a buffer into which your program has placed the pathname of the output file (if any): that is, the file that the compiler or assembler writes to. The filename can be any valid ProDOS 16 filename and either a partial or full pathname.

$08–0B `parms`

parameter name: address of parameter list
size and type: 4-byte pointer
range of values: $0000 0000–$00FF FFFF

This parameter specifies the address of a buffer containing the list of names from the NAMES= parameter list in the APW Shell command that called the assembler or compiler. Because the compiler can remove or modify these names as it processes them, this list can be different from the one received through the `GET_LINFO` call.

If the +E flag is set and the compiler exits due to an error, the compiler should place the text of the error message into a buffer and set the `parms` parameter to point to that buffer. The editor can then display the error message at the bottom of the screen.

$0C–$0F `istring`

parameter name: address of input strings
size and type: 4-byte pointer
range of values: $0000 0000–$00FF FFFF

This parameter is a placeholder for the address of a buffer containing the string of commands passed to the compiler. Because this command string is not reused by the shell, it is not necessary to pass it back to the shell with the `SET_LINFO` call.

$10 `merr`

parameter name: maximum error level allowed
size and type: 1-byte value
range of values: $00–$10

If the maximum error level found by the assembler, compiler, or linker (`merrf`) is greater than `merr`, the shell does not call the next program in the processing sequence. For example, if you use the ASML command to assemble and link a program, but the assembler finds an error level of 8 when `merr` equals 2, then the linker is not called when the assembly is complete.

\$11 `merrf`

parameter name: maximum error level found
size and type: 1-byte value
range of values: \$00–\$FF

This field is used by the `SET_LINFO` call to return the maximum error level found. If `merrf` is greater than `merr`, no further processing is done by the shell. If the high bit of `merrf` is set, `merrf` is considered to be negative; a negative value of `merrf` indicates a fatal error (normally, all fatal errors are flagged as `merrf` = \$FF). In this case, processing terminates immediately. See also the discussion of the `org` parameter.

\$12 `lops`

parameter name: operations flags
size and type: 1-byte value
range of values: \$00–\$10

This field is used to keep track of the operations that have been performed by the system. The format of this byte is

Bit:	7	6	5	4	3	2	1	0
Value:	0	0	0	0	0	E	L	C

where C = Compile
L = Link
E = Execute

When a bit is set (to 1), the indicated operation is to be done. When a compiler finishes its operation and returns control to the shell, it clears bit 0 unless a file with another language is appended to the source. When a linker returns control to the shell, it clears bit 1. When you execute the APW Linker by compiling a LinkEd file, the linker clears bits 0 and 1.

\$13 `kflag`

parameter name: keep flag
size and type: 1-byte value
range of values: \$00–\$03

This flag indicates what should be done with the output of a compiler, assembler, or linker, as follows:

kflag Value	Meaning
$00	Do not save output.
$01	Save to an object file with the root filename pointed to by `dfile`. For example, if the output filename pointed to by `dfile` is `PROG`, the first segment to be executed should be put in `PROG` or `PROG.ROOT` and the remaining segments should be put in `PROG.A`. For linkers, save to a load file with the name pointed to by `dfile` (for example, `PROG`). A compiler or assembler will never set `kflag` to $01, but a shell program calling APW might use this value.
$02	The `.ROOT` file has already been created. In this case, the first file created by the next compiler or assembler should end in the `.A` extension.
$03	At least one alphabetic suffix has been used. In this case, the compiler or assembler must search the directory for the highest alphabetic suffix that has been used, and then use the next one. For example, if `PROG.ROOT`, `PROG.A`, and `PROG.B` already exist, the compiler should put its output in `PROG.C`.

When the compiler or assembler passes control back to the shell, it should reset `kflag` to indicate which object files it has written; for example, if it found only one segment and created a `.ROOT` file but no `.A` file, then `kflag` should be $02 in the `SET_LINFO` call. See the section "Compilers and Assemblers" in Chapter 6 for more information on object-file naming conventions.

$14–$17 `mflags`

parameter name: flags with a minus sign
size and type: 4-byte value
range of values: binary string

This parameter passes command-line-option flags such as `-L` or `-C`. The first 26 bits of these four bytes represent the letters A through Z, arranged with A as the most significant bit of the most significant byte. The bytes are ordered least significant byte first. The bit map is as follows:

```
11000000 11111111 11111111 11111111
YZ       QRSTUVWX IJKLMNOP ABCDEFGH
```

For each flag set with a minus sign in the command, the corresponding bit in this parameter is set to 1. See the discussions of the `ALINK` and `ASML` commands in Chapter 3 for descriptions of these option flags.

$18–$1B `pflags`

parameter name: flags with a plus sign
size and type: 4-byte value
range of values: binary string

This parameter passes command-line-option flags such as `+L` or `+C`. The first 26 bits of these four bytes represent the letters A through Z; the bit map for this parameter is the same as for the `mflags` parameter. See the discussions of the `ALINK` and `ASML` commands in Chapter 3 for descriptions of these option flags.

$1C–$1F `org`

parameter name: origin
size and type: 4-byte value
range of values: $0000 0000–$FFFF FFFF

This parameter specifies the absolute start address of a nonrelocatable load file, if one has been specified. The origin is used only by the linker. If the `+E` flag is set and the compiler exits due to an error, the compiler should put the offset of the line containing the error into the `org` field. The editor can then place that line on the fifth line of the screen.

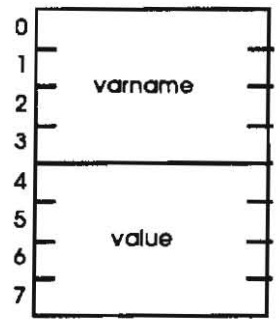
Possible Errors

None

SET_VAR ($0106)

This function sets the value of a variable. If the variable has not been previously defined, this function defines it. Variables are described in the section "Exec Files" in Chapter 3. Use the GET_VAR call to read the current value of a variable and the READ_INDEXED call to read a variable table.

Parameter Block:



Offset	Label	Description
$00–$03	varname	**parameter name:** pointer to name of variable **size and type:** 4-byte pointer **range of values:** $0000 0000–$00FF FFFF This is a pointer to a buffer in which you place the name of the variable whose value you wish to change. The name is an ASCII string.
$04–$07	value	**parameter name:** pointer to value of variable **size and type:** 4-byte pointer **range of values:** $0000 0000–$00FF FFFF This is a pointer to a buffer in which you place the value to which the variable is to be set. The value is an ASCII string.

Possible Errors

Errors for the following Memory Manager calls. See the *Apple IIGS Toolbox Reference* manual for descriptions of these errors.

Lock
Unlock
Grow
New

STOP ($0113)

This function lets your application detect a request for an early termination of the program. The stop flag is set when the keyboard buffer is read after the user presses Apple–Period (⌘-.).

Parameter Block:

Offset	Label	Description
$00–$01	stop	**parameter name:** stop flag **size and type:** 2-byte result **range of values:** $0000–$0001 This flag is set ($0001) by the shell when it finds an Apple–Period in the keyboard buffer. When an APW utility reads from the keyboard as standard input, the shell reads the keyboard buffer and passes the keys on to the utility. When standard input is not from the keyboard, the shell still checks the keyboard buffer for Apple–Period whenever a STOP call is executed. The flag is cleared ($0000) when the STOP call is executed, when the utility program is terminated, or when windows are switched so that the utility program is no longer active. See the section "Conventions" in Chapter 6 for a routine that both checks for Apple-Period and pauses output to the screen when a key is pressed.

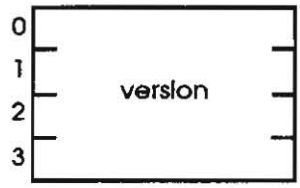
Possible Errors

None

VERSION ($0107)

This function returns the version of the APW Shell that you are using.

Parameter Block:



Offset	Label	Description
$00–$03	`version`	**parameter name:** version number **size and type:** 4-byte result **range of values:** $0000 0000–$3939 3939 A four-byte ASCII string specifying the version number of the APW Shell that you are using. The initial release returns 10 followed by two space characters ($3130 2020), to indicate Version number 1.0.

Possible Errors

None

WRITE_CONSOLE ($011A)

This function writes a character to the Pascal console driver. The resulting output is not redirectable, so you can use this function to echo keyboard input and to send messages that must appear on the screen.

Parameter Block:

Offset	Label	Description
$00-$01	ochar	**parameter name:** output character **size and type:** 2-byte value **range of values:** $0000–$00FF A two-byte value specifying a character to write on the screen. The low byte of the value is sent to the Pascal console driver.

Possible Errors

None

Appendixes

Appendix A

Contents of the APW Disks

The following files should all be present on your APW system disks.

/APW Disk

Directory or File	Description
/APW/	APW directory.
PRODOS	ProDOS system startup.
SYSTEM/	Operating system subdirectory.
P8	ProDOS 8 operating system
P16	ProDOS 16 operating system and System Loader.
START	The Program Launcher.
SYSTEM.SETUP/	A subdirectory containing system programs to be executed at system startup time.
TOOLS/	A subdirectory containing all the RAM-based Apple IIGS tool sets.
DESK.ACCS/	A subdirectory containing Apple IIGS desk accessories.
DRIVERS/	A subdirectory containing device drivers.
FONTS/	A subdirectory containing fonts.
APW/	A subdirectory containing APW files.
SYSTEM/	A subdirectory containing APW system files.
LOGIN	APW command file executed on startup.
SYSHELP	Help screen for the editor.
EDITOR	APW Editor.
SYSCMND	List of APW command names and command numbers. You can edit this file to add or delete commands.
SYSTABS	Editor defaults file. You can edit this file to set editor defaults for any APW language.
SYSEMAC	Editor macro file.
LANGUAGES/	APW languages subdirectory. All compilers must be installed in this subdirectory.
LINKED	APW Linker.
ASM65816	APW Assembler.
WORK/	Subdirectory for APW temporary work files.
LIBRARIES/	Subdirectory for library files. Linker libraries made with the MakeLib program should go in here.
AINCLUDE/	Subdirectory of assembler macro files and global equates files.
APW.SYS16	The APW Shell program.
UTILITIES/	APW utilities subdirectory.
INIT	Formats disks.
INSTALL	Installs APW on a hard disk.
CRUNCH	Combines object files into a single file.
DO	Used during APW installation.
MACGEN	Makes custom macro files.
XDO	Used during APW installation.
INSTALL2	Installation routine.
INSTALLHD	Installation routine.

/APWU Disk

/APWU	APW utilities volume.
/UTILITIES	APW utilities subdirectory.
HELP/	Help-file subdirectory. This directory contains one help file for each APW command.
DO	Installation routine.
XDO	Installation routine.
MAKEBIN	Creates BIN files from load files.
INIT	Formats disks.
DUMPOBJ	APW object-module-format file dump routine.
MAKELIB	Creates library files.
CRUNCH	Combines object files into a single file.
MACGEN	Makes custom macro files.
COMPACT	Makes load files more compact.
CANON	Canonical spelling checker.
EQUAL	Compares files and directories.
FILES	Lists directories.
SEARCH	Searches for specified character string.
CANON.DICT	Dictionary for Canon utility.
INSTALL	Installation routine.
DEBUG	Message about debugger.
VERSION	Displays version number of APW.
INSTALLHD	Installation routine.
INSTALL2	Installation routine.

Appendix B

Command Summary

This appendix lists the currently defined APW language types and summarizes the commands used in the APW Shell, Exec files, APW Editor, and LinkEd files.

The following notation is used to describe commands:

The following notation is used to describe commands:

UPPERCASE	Uppercase letters indicate a command name or an option that must be spelled exactly as shown. The shell is not case sensitive; that is, you can enter commands in any combination of uppercase and lowercase letters. Segment names *are* case-sensitive. In case-sensitive languages, segment names must be entered exactly as they appear in the source code. Segment names in case-insensitive languages must be entered in uppercase.
italics	Italics indicate a variable, such as a filename or address.
directory	This parameter indicates any valid directory pathname or partial pathname. It does *not* include a filename. If the volume name is included, *directory* must start with a slash (/); if *directory* does not start with a slash, the current prefix is assumed. The device names `.D1`, `.D2`, . . . ,`.D`*n* can be used for volume names. ProDOS 16 prefix numbers can be used for directory prefixes. If you use a device name or prefix number, do not precede it with a slash.
filename	This parameter indicates a filename, *not* including the prefix. The unit names `.CONSOLE` and `.PRINTER` can be used as filenames.
pathname	This parameter indicates a full pathname, including the prefix and filename, or a partial pathname, in which the current prefix is assumed. A full pathname (including the volume name) must begin with a slash (/); do *not* precede *pathname* with a slash if you are using a partial pathname. The device names `.CONSOLE` and `.PRINTER` can be used as filenames, the device names `.D1`, `.D2`, . . . `.D`*n* can be used for volume names, and ProDOS 16 prefix numbers can be used for prefixes.
|	A vertical bar indicates a choice. For example, `+L|-L` indicates that the command can be entered as either `+L` or as `-L`.
A | **B**	An underlined choice is the default value.
[ ]	Parameters enclosed in square brackets are optional.
. . .	Ellipses indicate that a parameter or sequence of parameters can be repeated as many times as you wish.

•
•
•

Vertical ellipses indicate that any number of commands can be inserted between the two commands shown.

Language Types

The following language types are currently assigned. The inclusion of a language on this list does not necessarily imply that the language compiler exists or ever will exist for APW. For a complete list of currently-assigned language types, see Apple IIGS Technical Note #20.

Language	Number	Use
ASM6502	2	6502 Assembler
ASM65816	3	65816 Assembler
BASIC	4	APW BASIC
BWBASIC	9	Byte Works BASIC
BWC	8	Byte Works C
BWPASCAL	5	Byte Works Pascal
CC	10	APW C
COMMAND	12	APW command-processor window
EXEC	6	command file
LINKED	9	APW Linker command language
PASCAL	11	APW Pascal
PRODOS	0	ProDOS 16 text file (ProDOS 16 file type $04)
SMALLC	7	Byte Works small C
TEXT	1	APW text file
TMLPASCAL	30	TML Pascal

If you are a certified Apple developer and you need a new language number for your compiler, write to

Developer Technical Support
Mail Stop 27 T
Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014

Shell

*

Null command, used to add comments to Exec files.

ALIAS [*alias* [*command*]]
Create an alias for a command.

ALINK [+E|-E] [+L|-L] [+S|-S] [+T|-T] [+W|-W] *file1* [*file2* ...]
[KEEP=*outfile*]
Compile a linker command file.

ASM65816
Change default language to 65816 assembly language.

ASML [+E|-E] [+L|-L] [+S|-S] [+T|-T] [+W|-W]
file1 [*file2*] [...] [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2*] [...])] [*language1*=(*option* ...)
[*language2*=(*option* ...)] [...]]
Assemble and link the program.

ASMLG [+E|-E] [+L|-L] [+S|-S] [+T|-T] [+W|-W]
file1 [*file2*] [...] [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2*] [...])] [*language1*=(*option* ...)
[*language2*=(*option* ...)] [...]]
Assemble, link, and go (run the program).

ASSEMBLE [+E|-E] [+L|-L] [+S|-S] [+T|-T] [+W|-W]
file1 [*file2*] [...] [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2*] [...])] [*language1*=(*option* ...)
[*language2*=(*option* ...)] [...]]
Assemble the program.

BREAK
Terminate the innermost FOR, LOOP, or IF statement currently executing.

CANON [+A|-A] [+C *n*] [+S|-S] *dictionary* [*inputfile*]
Compare the spelling of words in the input file with words in the dictionary file and replace with canonical spelling in the dictionary file.

CAT [*pathname*]
List the specified directory.

CATALOG [*pathname*]
List the specified directory.

CC
Change default language to APW C.

CHANGE *pathname language*
Change the language type of an existing source file.

CMPL [+E|-E] [+L|-L] [+S|-S] [+T|-T] [+W|-W]
file1 [*file2*] [...] [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2*] [...])] [*language1*=(*option* ...)
[*language2*=(*option* ...)] [...]]
Compile and link the program.

CMPLG [+E|-E] [+L|-L] [+S|-S] [+T|-T] [+W|-W]
file1 [*file2*] [...] [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2*] [...])] [*language1*=(*option* ...)
[*language2*=(*option* ...)] [...]]
Compile, link, and go (run the program).

COMMANDS *pathname*
Read the command table.

COMMENT
Null command, used to add comments to Exec files.

COMPACT *infile* [-O *outfile*] [-P] [-R] [-S]
Convert a load file to the most compact form provided for by the object module format.

COMPILE [+E|-E] [+L|-L] [+S|-S] [+T|-T] [+W|-W]
file1 [*file2*] [...] [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2*] [...])] [*language1*=(*option* ...)
[*language2*=(*option* ...)] [...]]
Compile the program.

CONTINUE
Cause control to skip over following statements to the next END statement.

COPY [-C] *pathname1* [*pathname2*]
COPY [-D] *volume1 volume2*
Copy a file to a file, a file to a directory, a directory to a directory, or do a block copy of a disk.

CREATE *directory1* [*directory2* ...]
Create one or more new subdirectories.

CRUNCH *rootname*
Combine object modules formed by partial compiles or assemblies into a single file.

DEBUG
Execute the Apple IIGS Debugger program, if available.

DELETE [-C] *pathname1* [*pathname2* ...]
Delete a file or files.

DISABLE D|N|W|R *pathname1* [*pathname2* ...]
Disable file attributes.

DUMPOBJ [+X] [+D] [-H] [-O] [-F] [-M] [-I] [-A] [-L] [-S]
pathame [NAMES=(*seg1 seg2* ...)]
List the contents of an OMF file to standard output.

ECHO *string*
Write a message to the screen.

EDIT *pathname*
Edit an existing file or open a new file.

ELSE
Part of an IF–END command sequence.

ELSE IF
Part of an IF–END command sequence.

ENABLE D|N|B|W|R *pathname1* [*pathname2* ...]
Enable file attributes.

END
Terminate a FOR, IF, or LOOP command sequence.

EQUAL [+D|-D] [+M|-M] [+N *n*] [+P|-P] [+T|-T] *pathame1 pathname2*
Compare two files or directories for data equality and show differences in file dates or types.

EXEC
Change default language to EXEC command language.

EXECUTE *pathname* [*paramlist*]
Execute an Exec file at present command level.

EXIT [*number*]
Terminate execution of an Exec file.

EXPORT [*variable*]
Make the specified variable available to Exec files called by the current Exec file.

FILES [+C *n*] [+F *string*] [+L|-L] [+P|-P] [+R|-R] *directory*
List the contents of a directory.

FILETYPE *pathname filetype*
Change file type to the type specified.

FOR *variable* [IN *value1 value2* ...]
Together with END, create a loop that is executed once for each parameter value listed.

HELP [*commandname*]
Provide on-screen help for commands or list all available commands.

HISTORY
List the last 20 APW commands entered on the command line.

`IF` *expression*
Provide conditional branching in an Exec file.

`INSTALL` *volume*
`INSTALL /APW` *directory*
Install an APW distribution disk or install the `/APW` disk.

`INIT [-C]` *device* `[`*name*`]`
Initialize a disk.

`LINK [+L|-L] [+S|-S] [+W|-W]` *file1* `[`*file2*`] [...] [KEEP=`*outfile*`]`
Link an object module.

`LINKED`
Change default language to the LinkEd command language, LINKED.

`LOOP`
Together with `END`, define a loop that repeats continuously until a `BREAK` or `EXIT` command is encountered.

`MACGEN [+C|-C]` *infile outfile macrofile1* `[`*macrofile2* ...`]`
Generate a macro library for a specific program.

`MAKEBIN` *loadfile* `[`*binfile*`] [ORG=`*val*`]`
Convert a ProDOS 16 load file (file type $B5 only) to a ProDOS 8 binary load file (file type $06).

`MAKELIB [-F] [-D]` *libfile* `[+`*objectfile* ...`] [-`*objectfile* ...`] [^`*objectfile* ...`]`
Generate or edit a library file.

`MOVE [-C]` *pathname1* `[`*pathname2*`]`
Move a file and rename it.

`MOVE [-C]` *pathname* `[`*directory*`]`
Move a file without renaming it.

`MU`
An alias for `PREFIX 6 /APWU/UTILITIES`; defined in the `LOGIN` file for running APW on floppy disks.

`PREFIX [`*n*`]` *directory*`[/]`
Change the default prefixes.

`PRODOS`
Change default language to ProDOS 16 text (file type $04).

`QUIT`
Quit APW.

RENAME *pathname1 pathname2*
Change a filename.

RUN [+E|-E] [+L|-L] [+S|-S] [+T|-T] [+W|-W]
file1 [*file2*] [...] [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2*] [...])] [*language1*=(*option* ...)
[*language2*=(*option* ...)] [...]]
Compile, link, and run a program; same as ASMLG or CMPLG.

SEARCH [+C|-C] [+L|-L] [+P|-P] *string pathname*
Search a file or files for the string you specify.

SET [*variable* [*value*]]
Assign a value to a variable name.

SHOW [LANGUAGE] [LANGUAGES] [PREFIX] [TIME] [UNITS]
Show languages, system default language, prefixes, time, volumes on line.

TEXT
Change default language to TEXT.

TYPE [+N|-N] *pathname1* [*startline1* [*endline1*]]
[*pathname2* [*startline2* [*endline2*]] [...]]
Type a file to standard output.

UMU
An alias for PREFIX 6 4/../UTILITIES, defined in the LOGIN file for running APW on floppy disks.

UNALIAS *alias1* [*alias2* ...]
Delete aliases for commands.

UNSET *variable1* [*variable2* ...]
Delete the definition of a variable.

VERSION
Display the version number of the copy of APW that you are using.

Exec Files

BREAK
Terminate the innermost FOR or LOOP statement currently executing.

CONTINUE
Cause control to skip over following statements to the next END statement.

ECHO *string*
Write messages to the screen.

EXECUTE *pathname* [*paramlist*]
Execute an Exec file at present command level.

EXIT [*number*]
Terminate execution of the Exec file.

EXPORT [*variable*]
Make the specified variable available to Exec files called by the current Exec file.

FOR *variable* [IN *value1 value2* ...]
    •
    •
    •
END
Create a loop that is executed once for each parameter value listed.

IF *expression*
    •
    •
    •
[ELSE IF *expression*]
    •
    •
    •
[ELSE]
    •
    •
    •
END
Provide conditional branching in Exec files.

LOOP
    •
    •
    •
END
Define a loop that repeats continuously until a BREAK or EXIT command is encountered. The loop is also terminated if any command in the loop returns a nonzero error status while the variable {EXIT} has a non-null value.

SET [*variable* [*value*]]
Assign a value to a variable name.

UNSET *variable1* [*variable2* ...]
Delete the definition of a variable.

Editor

Beep the Speaker	Control-G
Beginning of Line	⌘-, ⌘-<
Bottom of Screen / Page Down	Control-⌘-J ⌘-↓
Change	See Search and Replace.
Clear	⌘-Delete
Copy	Control-C ⌘-C
Cursor Down	Control-J ↓
Cursor Left	Control-H ←
Cursor Right	Control-U →
Cursor Up	Control-K ↑
Cut	Control-X ⌘-X
Define Macros	⌘-Esc
Delete Block	See Clear
Delete Character	Control-F ⌘-F
Delete Character Left	Delete Control-D
Delete Line	Control-T ⌘-T
Delete to EOL	Control-Y ⌘-Y

Delete Word	Control-W Ó-W
End of Line	Ó-. Ó->
End Macro Definition	Option-Esc
Enter Escape Mode	See Turn On Escape Mode
Execute Macro	Option-*letter*
Find	See Search.
Help	Ó-/ Ó-?
Insert Line	Control-B Ó-B
Insert Space	Ó-Space bar
Paste	Control-V Ó-V
Quit	Control-Q Ó-Q
Quit Macro Definitions	Option
Remove Blanks	Control-R Ó-R
Repeat Count	1 to 32767
Return	Return Control-M
Screen Moves	Ó-1 to Ó-9
Scroll Down One Line	Control-P Ó-P
Scroll Down One Page	See Bottom of Screen/Page Down
Scroll Up One Line	Control-O Ó-O
Scroll Up One Page	See Top of Screen/Page Up
Search Down	Ó-L

Search Up	♁-K
Search and Replace Down	♁-J
Search and Replace Up	♁-H
Set and Clear Tabs	♁-Tab Control-♁-I
Start of Line	♁-, ♁-<
Tab	Tab Control-I
Tab Left	Control-A ♁-A
Toggle Auto Indent Mode	♁-Return ♁-Enter Control-♁-M
Toggle Escape Mode	Esc
Toggle Insert Mode	Control-E ♁-E
Toggle Select Mode	Control-♁-X
Toggle Wrap Mode	Control-♁-W
Top of Screen / Page Up	Control-♁-K ♁-↑
Turn Off Escape Mode	Control-♁-_
Turn On Escape Mode	Control-_
Undo Delete	Control-Z ♁-Z
Word Left	♁-← Control-♁-H
Word Right	♁-→ Control-♁-U

Defining Macros

♁-Esc	Begin macro definitions.
→	Display the next screen of macro definitions.
←	Display the previous screen of macro definitions.
letter	Begin defining the macro corresponding to the letter-key *letter*. Note that *letter* must be displayed on the screen before you begin to define it.
Option-Delete	Delete the character to the left of the cursor.
Option-Esc	Terminate the macro definition.
Option	Stop defining macros and return to editing the file. If you are currently defining a macro, press Option-Esc first to terminate the macro definition, and *then* press Option to return to the file.

Keystroke Summary

Key	Control	♂	Control-♂
A	tab left	tab left	
B	insert line	insert line	
C	copy	copy	
D	del char left		
E	toggle insert	toggle insert	
F	del char	del char	
G	beep speaker		
H	cursor left	replace up	word left
I	tab		set tabs
J	cursor down	replace down	bot scrn/pg down
K	cursor up	search up	top scrn/page up
L		search down	
M	Return		toggle auto indent
N			
O	up one line	up one line	
P	down one line	down one line	
Q	quit	quit	
R	remove blanks	remove blanks	
S			
T	del line	del line	
U	cursor right		word right
V	paste	paste	
W	del word	del word	toggle wrap
X	cut	cut	toggle select
Y	del to eol	del to eol	
Z	undo delete	undo delete	
?		help	
Delete		clear	
Esc		define macros	
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
<		start of line	
>		end of line	
_	turn on escape		turn off escape
↓		bot scrn/pg down	
↑		top scrn/page up	
←		word left	
→		word right	
Tab		set tabs	
Return		toggle auto indent	
Enter		toggle auto indent	
Space bar		insert a space	

LinkEd

APPEND *linkedname*
Append a LinkEd source file.

COPY *linkedname*
Copy a LinkEd source file.

EJECT
Skip to a new page if printer is on.

KEEP *loadname*
Open a file for output.

KEEPTYPE *filetype*
Set the file type of the load file produced by the linker.

LIBRARY *libname*
Search a library by object-segment names.

LIBRARY/LOADSELECT *libname lseg*
Search a library by load-segment names.

LINK[/ALL] *objname*
Link an object file.

LIST ON|OFF
Control link-map listing.

LOADSELECT[/SCAN] *objname lseg*
Include object segments with a specific load-segment name in the object file.

OBJ *val*
Set phantom program counter.

OBJEND
Turn off previous OBJ.

ORG *val*
Set program counter.

PRINTER ON|OFF
Control printed output.

SEGMENT[/DYNAMIC][/*kind*] *segname*
Start load segment.

SELECT[/SCAN] *objname* (*seg1*[,*seg2*[,...]])
Choose specific object segments.

`SOURCE ON|OFF`
Control LinkEd source program listing.

`SYMBOL ON|OFF`
Control symbol table output.

Appendix C

Error Messages

Shell Errors

When you are using the APW Shell, you can receive two types of errors: errors generated by the shell itself, and errors returned to the shell by another program. In the latter case, the error is preceded by the name of the program that returned it. For example, if the shell calls ProDOS 16 to open a file and ProDOS cannot find the file, the following error is printed on the screen:

```
ProDOS:  File not found
```

Since the APW Shell interacts with both the user and with a variety of other programs, both outside of APW (such as ProDOS and the Memory Manager) and within APW (such as the editor), the variety and possible causes of errors are too great to allow all possibilities to be listed here. If the message itself does not provide you with sufficient information to solve the problem, read the section of this manual that describes the operation you were trying to perform. A few hints are given here, however, for specific errors for which the cause may not at first be clear.

File Not Found

When you type a command and press Return, APW first checks the command table to see if it is a standard command. If the command is not in the command table, APW assumes it is the name of an executable file and asks ProDOS 16 to open a file by that name in the current prefix. If ProDOS 16 does not find a file by that name, the message `ProDOS: File not found` is printed on the screen. This message indicates that ProDOS 16 could not find a file with the name of the command you typed. Check the prefix and spelling of your command and try again.

The `File not found` error can be confusing when you have also typed a pathname as a parameter for the command. For example, suppose you want to edit the file `MYFILE`, so you enter the following command:

```
ED MYFILE
```

Unfortunately, `ED` is not a valid APW command (unless you have added it to the command table yourself). APW looks in the command table for `ED`, doesn't find it, and calls ProDOS 16 to try to open a file named `ED`. ProDOS can't find the file, and the message `File not found` is printed on the screen. When you see this message, it is important to realize that the file that ProDOS 16 couldn't find is `ED`, *not* `MYFILE`.

The `File not found` message is also printed when you attempt to execute the Paste command in the editor without first executing a Copy or Cut command. When you execute the Paste command, the Editor looks for the file `SYSTEMP` in the work prefix; this file does not exist unless a Copy or Paste command has been executed first.

The `ALIAS` command can disable any command in the command table, resulting in a `File not found` message when you try to execute the command. See the discussion of the `ALIAS` command in Chapter 3 for details.

Volume Not Found

A similar problem can occur if you remove your APW disk from the disk drive or change a APW prefix (such as the utility prefix, prefix 6) and then try to execute an external command (such as `INIT`) or to read a help file. In this case, ProDOS 16 cannot find the directory containing the utility program or help file, and the message `Volume not found` or `Path not found` is printed to the screen. Again, it is important to realize that the volume or path that could not be found is the one containing the utility or help file, *not* one used in a parameter to the command.

For example, if you remove the APW disk from the disk drive and then enter the command `EDIT MYFILE`, ProDOS 16 cannot find the volume `/APW` in order to load the editor (`/APW/SYSTEM/EDITOR`), so it prints the message `Volume not found`.

Unable to Open File

ProDOS may be unable to open a file for a variety of reasons: the disk may be write-protected, you may have specified a name for the file that exceeds the maximum allowed length (15 characters), the disk may be full, or the directory may be full (too many filenames in the directory), for example.

When you name an output file using the `KEEP` parameter on a command line or a `KEEP` directive in the source file, you must restrict the filename to ten characters so that APW can append the extension `.ROOT` to the filename. Using more than ten characters in such a filename will result in a fatal assembler or compiler error (`Unable to open output file`).

Linker Errors

In producing object modules, compilers and assemblers are incapable of detecting certain programming errors, particularly those involving conflicts among global labels, missing global labels, and incorrect memory allocation. It is the responsibility of the linker to find and report those errors.

This section lists and describes the error messages returned by the APW Linker. They are divided into two groups: *nonfatal* (the linker continues processing), and *fatal* (the linker stops). For nonfatal errors, the linker also returns an error-level number as an indication of the severity of the problem that caused the error.

When the linker finds an error in a LinkEd source file, it continues to check the entire LinkEd source file for errors, reports the errors, and then stops. In this case, none of the LinkEd commands are executed.

Nonfatal Errors

When the linker detects a nonfatal error, it prints

1. the number of bytes from the beginning of the segment to the error
2. the name of the segment that contained the error
3. the value of the program counter where the error was detected
4. an error message

At the end of the link an error summary is printed, listing the number of nonfatal errors and the highest error level found.

The following error levels are recognized. Refer to individual error message listings for further illustration of the significance of error levels.

Level	Meaning
2	General warning. There may be a problem, but no corrective action has been taken.
4	Corrected error. The linker detected an error and has corrected it according to its own interpretation (*Check the results of this correction carefully!*)
8	Uncorrected error. The linker detected an error that it could not correct, but it understood enough about it to leave the proper space for correction.
16	Uncorrected error. The linker detected an error and could not even tell how much space to leave. Relinking will be required when the problem is corrected.

The following errors are nonfatal. The error message as it appears on the screen is printed in boldface, followed by the error level; an explanation and advice for correcting the error follow in normal text. The listing is in alphabetical order by the first word of the message.

Addressing error [16]:
A label could not be placed at the same location on pass two as it was on pass one.

This error is almost always accompanied by another error, which caused this one to occur; correcting the other error will correct this one. If there is no accompanying error, check for disk errors by doing a full assembly and link. If the error still occurs, report the problem as a bug.

Address is not in current bank [8]
The (most-significant-truncated) bytes of an expression did not evaluate to the value of the current location counter.

For short-address forms (6502-compatible), the truncated address bytes must match the current location counter. This restriction does not apply to long-form addresses (65816 native-mode addressing).

This error occurs when you use a JSR or JMP instruction to jump to a label that is not in the current load segment. Because in general the linker cannot know in which bank the target load segment will be loaded, it assumes that it will be loaded in a different bank from the current segment and that therefore a long address is needed. If you know that the two segments will be loaded into the same bank, you can prevent the linker from flagging this error by using the CODECHK OFF directive.

Similarly, references to labels in a data segment named in a USING directive cause this error unless: a) the data segment is linked into the same load segment as the code segment containing the reference, b) a long address is used to reference the label, or c) you use a DATACHK OFF directive.

The CODECHK, USING, and DATACHK directives are described in the *Apple IIGS Assembler Reference*.

Address is not zero page [8]
The most significant bytes of the evaluated expression were not zero, but they were required to be zero by the particular statement in which the expression was used.

This error occurs only when the statement requires a zero-page address operand (range = 0 to 255).

Alignment factor must be a power of two [8]
An alignment factor that was not a power of 2 was used in the source code. In APW Assembly Language, the ALIGN directive is used to set an alignment factor.

Alignment factor must not exceed segment align factor [8]
An alignment factor specified inside the body of an object segment is greater than the alignment factor specified before the start of the segment. For example, if the segment is aligned to a page boundary (ALIGN = 256), you cannot align a portion of the segment to a larger boundary (such as ALIGN = 1024).

Code exceeds code bank size [4]
The load segment is larger than one memory bank (64K). You have to divide your program into smaller load segments.

Data area not found [2]
A USING directive was issued in a segment, and the linker could not find a DATA segment with the given name.

Ensure that the proper libraries are included, or change the USING directive.

Duplicate label [8]
A label was defined twice in the program.

Remove one of the definitions.

Expression operand is not in same segment [8]
An expression in the operand of an instruction or directive includes labels that are defined in two different relocatable segments. The linker cannot resolve the value of such an expression.

Evaluation stack overflow [8]
(a) There may be a syntax error in the expression being evaluated.

Check to see if a syntax error has also occurred; if so, correct the problem that caused that error.

(b) The expression may be too complex for the linker to evaluate.

Simplify the expression. An expression would have to be extremely complex to overflow the linker's evaluation stack, particularly if the expression passed the assembler without error.

Expression syntax error [16]
The format of an expression in the object module being linked was incorrect.

This error should occur only in company with another error; correct that error and this one should be fixed automatically. If there are no accompanying errors, check for disk errors by doing a full assembly and link. If the error still occurs, report the problem as a bug.

Invalid operation on relocatable expression [8]
The APW Linker can resolve only certain expressions that contain labels that refer to relocatable segments. The following types of expressions *cannot* be used in an assembly-language operand involving one or more relocatable labels:

- a bit-by-bit NOT
- a bit-by-bit OR
- a bit-by-bit EOR
- a bit-by-bit AND
- a logical NOT, OR, EOR, or AND
- any comparison (<, >, <>, <=, >=, ==)
- mulitplication
- division
- integer remainder (MOD)

The following types of expressions involving a bit-shift operation *cannot* be used:

- The number of bytes by which to shift a value is a relocatable label.
- A relocatable label is shifted more than once.
- A relocatable label is shifted and then added to another value.

- You cannot use addition where both values being added are relocatable (you *can* add a constant to a relocatable value).

- You cannot subtract a relocatable value from a constant (you *can* subtract a constant from a relocatable value).

- You cannot subtract one relocatable value from another defined in a different segment (you *can* subtract two relocatable values defined in the *same* segment).

Only JSL can reference dynamic segment [8]
You referenced a dynamic segment in an instruction other than a `JSL`. Only a `JSL` can be used to reference a dynamic segment.

You can suppress this error by using the `DYNCHK OFF` directive, as described in the *Apple IIGS Assembler Reference*.

ORG Location has been passed [16]
The linker encountered an `ORG` directive for a location it had already passed.

Move the segment to an earlier position in the program. This error applies only to absolute code, and should therefore be rarely encountered when writing for the Apple IIGS.

Relative address out of range [8]
The given destination address is too far from the current location.

Change the addressing mode or move the destination code closer.

Segment header MEM directive not allowed [16]
The `MEM` directive cannot be used in a relocatable segment.

Segment header ORG not allowed [16]
If there is no `ORG` specified in the LinkEd file or at the beginning of the source code, you cannot include an `ORG` within the program. The linker generates relocatable code unless it finds an `ORG` before the start of the first segment. Once some relocatable code has been generated, the linker cannot accept an `ORG`.

Shift operator is not allowed on JSL to dynamic segment [8]
The operand to a `JSL` includes the label of a dynamic segment that is acted on by a bit-shift operator. You probably typed the wrong character, or used the wrong label by mistake.

Undefined opcode [16]
The linker encountered an instruction that it does not understand. There are four possible reasons:

1. The linker is an older version than that required by the assembler or compiler; in this case, a `Linker Version Mismatch` error should have occurred also. Update the linker.

2. An assembly or compilation error caused the generation of a bad object module. Check and remove all assembly/compilation errrors.

3. The object module file has been physically damaged. Recompile to a fresh disk.

4. There is a bug in the assembler, compiler, or linker. Please report the problem for correction.

Unresolved reference [8]
The linker could not find a segment referenced by a label in the program.

If the label is listed in the global symbol table after the link, make sure the segment that references the label has issued a `USING` directive for the segment that contains the label. Otherwise, correct the problem by (1) removing the label reference, (2) defining it as a global label, or (3) defining it in a data segment.

Fatal Errors

There are two kinds of fatal errors: for many fatal errors, the linker continues processing. It prints the error message, waits for a keypress, and then quits. For some others, the linker prints the error message, continues to process the file to search for other errors, and then quits without writing a load file.

The following errors are fatal. The error message as it appears on the screen is printed in boldface; an explanation follows in normal text. The listing is in alphabetical order by the first word of the message.

Cannot change languages.
An `APPEND` or `COPY` command in a LinkEd file has called a file that is not a LinkEd file.

LinkEd has to be the last language processed in an assembly or compile; you cannot append a source file to a LinkEd file.

Could not open file *filename*.
ProDOS 16 could not open the file *filename*, which you specified in the command line or LinkEd command.

Check the prefix and the spelling of the filename you specified. Make sure the file is present on the disk and that the disk is not write-protected.

Could not overwrite existing file *filename*.
The linker is only allowed to replace an existing output file if the file type of the output file is one of the executable types. It is not allowed to overwrite a `TXT`, `SRC`, or `OBJ` file.

Could not write the Keep file *filename*.
A ProDOS error occurred while the linker was trying to write the output file *filename*.

This error is usually caused by a full disk. Otherwise, there may be a bad disk or disk drive.

Dictionary file could not be opened.
The dictionary file is a temporary file on the work prefix that holds information destined for the load file's relocation dictionary. For some reason, this file could not be opened.

Use the `SHOW PREFIX` command to find out what the work prefix is. Perhaps you have assigned the work prefix to a RAM disk, but do not have a RAM disk on-line. Have you removed the volume containing the work prefix from the disk drive? Is the disk write-protected?

Expected '('.
The left parenthesis is missing from the list of segments in the LinkEd `SELECT` command.

Expression recursion level exceeded.
It is possible for an expression to be an expression itself; therefore, the expression evaluator in the linker is recursive. Generally, this error occurs when the recursion nest level exceeds ten. This should not happen very frequently. If it does, check for expressions with circular definitions, or reduce the nesting of expressions.

File name expected.
A filename is missing from a parameter or command that requires one, such as the `KEEP` parameter in the `ASML` command or the `LINK` command in a LinkEd file.

File read error.
An I/O error occurred when the linker tried to read a file that was already open.

This error should never occur. There may be a problem with the disk drive or with the file. You might have removed the disk before the link was complete.

File not found *filename*.
The file *filename* could not be found.

Check the prefix and spelling of the filename in both the `KEEP` directive and the `LINK` command. Make sure the `.ROOT` or `.A` file has the same prefix as the file specified in those commands.

Illegal command.
The linker does not recognize a command in your LinkEd file.

The offending LinkEd source line is printed out with an arrow pointing to the command in question. Check the spelling and syntax of the commands in your LinkEd file.

Illegal header value.
The linker checks the segment headers in object files to make sure they make sense. This error means that the linker has found a problem with a segment header.

This error should not occur. Your file may have been corrupted, or the assembler or compiler may have made an error.

Illegal segment structure.
There is something wrong with an object segment.

This error should not occur. Your file may have been corrupted, or the assembler or compiler may have made an error.

Invalid file name *filename*.
There is an illegal character in a filename, or you have used a filename that is longer than 15 characters.

Check the shell command or LinkEd file you used to call the linker and any KEEP directive in the source file to find the bad filename.

Invalid file type *filename*.
The file *filename* is not an object file or library file.

Check the shell command line or LinkEd file to make sure you didn't list any files that are not object files or library files. Check your disk directory to make sure there isn't a nonobject file with the same root filename as a file you are linking. For example, if you are linking object files named MYFILE.ROOT and MYFILE.A, make sure there is no (unrelated) file on the disk with the name MYFILE.B.

Invalid keep type.
The linker can generate several kinds of output files. The type of the output file must be one of the executable types. Since it is possible to set the keep type with a shell variable, this error can occur from a command-line call as well as from a LinkEd command.

Invalid segment name.
A segment name in your source file or named in a LinkEd SELECT command is not valid.

Object-segment names can be up to 255 characters long and must start with a letter, an underscore (_), or a tilde (~). The remaining characters must be alphanumeric or an underscore or tilde. Load-segment names follow the same rules as object-segment names, but they cannot be longer than ten characters.

Linker version mismatch.
The object-module-format version of the object segment is more recent than the version of the linker you are using.

Get the latest version of APW from the A.P.D.A.

Must be an object file *filename*.
The file *filename* is not an object file or a library file.

Multiple KEEP's not allowed.
Only one KEEP directive or parameter is allowed per link.

Make sure there is only one KEEP in your LinkEd file, and that there is not a KEEP both in your shell command line and the LinkEd file.

Must be an object file.
A file you specified for linking is not an object file.

Check the LINK commands in your LinkEd file to make sure that every file named is an object file (ProDOS 16 type $B1).

Number expected.
A number was missing for a parameter in your shell command or LinkEd file.

Object module read error.
A ProDOS error occurred while the linker was trying to read from the currently opened object module.

This error may occur after a nonfatal error; correcting the nonfatal errors may correct this one. Otherwise, it may be caused by a bad disk or disk drive.

OBJ not currently active.
You used an OBJEND command without first using an OBJ command.

'ON' or 'OFF' expected.
A command you used takes an ON or OFF as a parameter. This paramter is missing.

ORG location has been passed.
You specified a location in an ORG command that is before the beginning of the file.

Out of memory.
All free memory has been used; the memory needed by the linker is not available.

Output error.
A ProDOS error occurred while the linker was trying to write to the (output) load file.

This error is usually caused by a full disk. Otherwise, there may be a bad disk or disk drive.

Output file could not be opened.
A ProDOS error occurred while the linker was trying to open the (output) load file.

This error may be caused by trying to write to a full disk, a write-protected disk, or an unformatted disk. Otherwise, there may be a bad disk or disk drive.

Segment is not in module.
A segment you named in a SELECT command is not in the file you are linking.

Segment name expected.
A command you used takes a segment name as a parameter. This parameter is missing.

Segment name is too long.
A segment name you used is too long. Object-segment names must be 255 characters or less in length. Load-segment names must be ten characters or less in length.

Selected segment is already defined.
The segment you named in a SELECT command has already been linked. You cannot insert the same segment twice in the same load file.

Symbol table overflow
The symbol table could not hold all of the symbols needed by the program.

This error should occur only very rarely. If it does occur, decrease the number of global labels in the program. The START, DATA, ENTRY, and GEQU directives all create and pass global symbols to the linker. Labels inside data areas are also passed to the linker.

Value is out of range.
A number specified as a parameter is bigger than is permitted. For example, you may have specified a segment KIND larger than 255.

Glossary

absolute-bank segment: A load segment that is restricted to a specified bank but that can be relocated within that bank..

absolute code: Program code that must be loaded at a specific address in memory and never moved.

absolute segment: A segment that can be loaded only at one specific location in memory. Compare with **relocatable segment.**

access byte: An attribute of a ProDOS 16 file that determines what types of operations, such as reading or writing, may be performed on the file.

accumulator: The register in the 65C816 microprocessor of the Apple IIGS used for most computations.

address: A number that specifies the location of a single **byte** of memory. Addresses can be given as decimal or hexadecimal integers. The Apple IIGS has addresses ranging from 0 to 16,777,215 (in decimal) or from $00 00 00 to $FF FF FF (in hexadecimal). A complete address consists of a 4-bit **bank** number ($00 to $FF) followed by a 16-bit address within that bank ($00 00 to $FF FF).

Apple key: A modifier key on the Apple IIGS keyboard, marked with an Apple icon. It performs the same functions as the *Open Apple* key on standard Apple II machines.

Apple II: A family of computers, including the original Apple II, the Apple II Plus, the Apple IIe, the Apple IIc, and the Apple IIGS.

application prefix: The prefix of the last application launched.

APW Linker: The linker supplied with APW.

APW Shell: The shell program of APW. The APW Shell provides the interface between APW programs and ProDOS and between the user and APW.

assembler: A program that produces object files from source files written in assembly language.

bank: A 64K (65,536-byte) portion of the Apple IIGS internal memory. An individual bank is specified by the value of one of the 65C816 microprocessor's bank registers.

BIN file: A file in binary file format.

binary file format: The ProDOS 8 loadable file format, consisting of one absolute memory image along with its destination address. A file in binary file format has ProDOS file type $06 and is referred to as a BIN file. The System Loader cannot load BIN files.

block: (1) A unit of data storage or transfer, typically 512 bytes; (2) a contiguous, page-aligned region of computer memory of arbitrary size, allocated by the Memory Manager. Also called a *memory block.*

boot prefix: The volume name of the disk from which the computer was started up.

catalog: See **directory**.

character: Any symbol that has a widely understood meaning and thus can convey information. Some characters—such as letters, numbers, and punctuation—can be displayed on the monitor screen and printed on a printer. Most characters are represented in the computer as 1-byte values.

code segment: An object segment that contains program code. Code segments are provided for programs that differentiate between code and data segments.

command line: See **shell command line.**

command table: A text file containing a list of command names, command types (internal or *command*, external or *utility*, and *language*), and command or language numbers. The APW Shell checks the command table each time you execute a command. If it finds the command in the command table, it executes that command; if it doesn't find the command in the command table, the shell looks for a program with that name and attempts to run that program.

compiler: A program that produces object files from source files written in a high-level language such as C.

conditional assembly: A feature of an assembler that allows the programmer to define macros or other pieces of code such that the assembler assembles them differently under different conditions.

conditional compile: In a high-level language such as C, the use of preprocessor commands to vary the output depending on compile-time conditions.

controlling program: A program that loads and runs other programs, without itself relinquishing control. A controlling program is responsible for shutting down its subprograms and freeing their memory space when they are finished. A shell, for example, is a controlling program.

Control Panel: A **desk accessory** that lets you change certain system parameters, such as speaker volume, display colors, and configuration of slots and ports.

current application: The application program currently loaded and running. Every application program is identified by a **User ID** number; the current application is defined as that application whose User ID is the present value of the `USERID` global variable.

current language: The APW language type that is assigned to a file opened by the APW Editor. If an existing file is opened, the current language changes to match that of the file.

current prefix: The prefix that is used by the APW Shell if a partial pathname is used.

data segment: An object segment that consists primarily of data. Data segments are provided for programs that differentiate between code and data segments.

default prefix: See **current prefix.**

desk accessory: A small, special-purpose program that is available to the user regardless of which application is running. The Control Panel is an example of a desk accessory.

dispose: To permanently deallocate a memory block. The Memory Manager disposes of a memory block by removing its master pointer. Any handle to that pointer will then be invalid. Compare **purge**

directory: A file that contains a list of the names and locations of other files stored on a disk. Directories are either **volume directories** or **subdirectories.** A directory is sometimes called a *catalog.*

direct page: A page (256 bytes) of bank $00 of Apple IIGS memory, any part of which can be addressed with a short (1-byte) address because its high-address byte is always $00 and its middle-address byte is the value of the 65C816 processor's direct register. Coresident programs or routines can have their own direct pages at different locations. The direct page corresponds to the 6502 processor's **zero page.** The term *direct page* is often used informally to refer to the lower portion of the **direct-page/stack space.**

direct-page/stack segment: A load segment used to preset the direct-page and stack registers and to set the initial contents of the direct-page/stack space for an application.

direct-page/stack space: A portion of bank $00 of Apple IIGS memory reserved for a program's **direct page** and **stack.** Initially, the 65C816 processor's **direct register** contains the base address of the space, and its **stack register** contains the highest address. In use, the stack grows downward from the top of the direct-page/stack space, and the lower part of the space contains direct-page data.

direct register: A hardware register in the 65C816 processor that specifies the start of the direct page.

dormant: Said of a program that is not being executed, but whose essential parts are all in the computer's memory. A dormant program may be quickly **restarted** because it need not be reloaded from disk.

dynamic segment: A segment that can be loaded and unloaded during execution as needed. Compare with **static segment.**

emulation mode: For the Apple IIGS's 65C816 processor, the state in which it functions like a 6502 processor in all respects except clock speed. For the Apple IIGS computer, the state in which the computer functions like an 8-bit Apple II.

Exec file: A file of APW Shell commands that when executed, executes each command in turn as if it had been entered from the keyboard. You can pass parameters into Exec files and can include them in the command table as utilities.

external command: An APW utility program that functions like an APW Shell command.

external reference: A reference to a symbol that is defined in another segment. External references must be to global symbols.

fatal error: an error serious enough that the computer must halt execution.

field: A string of ASCII characters or a value that has a specific meaning to some program. Fields may be of fixed length, or they may be separated from other fields by field delimiters. For example, each parameter in a segment header constitutes a field.

field delimiter: A character or value that designates the start or end of a field. For example, in a BASIC file each field begins and ends with a Return character.

filename: The string of characters that identifies a particular file within a disk **directory**. ProDOS 16 filenames can be up to 15 characters long and can specify directory files, subdirectory files, text files, source files, object files, load files, or any other ProDOS 16 file type. Compare with **pathname.**

file number: A reference number assigned to a specific file. The loader assigns a file number to each load file in a program; the MakeLib utility program assigns a file number to each object file incorporated into a library file.

file number cross-reference: The part of the pathname table that contains load-file numbers and pointers to their corresponding pathnames.

file type: An attribute in a ProDOS 16 file's directory entry that characterizes the contents of the file and indicates how the file may be used. On disk, file types are stored as numbers; in a directory listing, they are often displayed as three-character mnemonic codes.

finder: A program that performs file and disk utilities (formatting, copying, renaming, and so on) and also starts applications at the request of the user.

full pathname: The complete name by which a file is specified. A full pathname always begins with a slash (/), because a volume directory name always begins with a slash. See **pathname.**

global symbol: A label in a code segment that is either the name of the segment or an entry point to it. Global symbols may be referenced by other segments. Compare with **local symbol.**

handle: See **memory handle.**

hexadecimal: The base-16 system of numbers, using the ten digits 0 through 9 and the six letters A through F. Hexadecimal numbers can be converted easily and directly to binary form, because each hexadecimal digit corresponds to a sequence of 4 bits. In Apple manuals hexadecimal numbers are usually preceded by a dollar sign ($).

high-level language: A programming language that is relatively easy for people to understand. A single statement in a high-level language typically corresponds to several instructions of machine language. Compare **low-level language.**

image: A representation of the contents of memory. A code image consists of machine-language instructions or data that may be loaded unchanged into memory.

index register: A register in a computer processor that holds an index for use in indexed addressing. The 6502 and 65C816 microprocessors used in the Apple II family of computers have two index registers, called the **X register** and the **Y register.**

initial load file: The first file of a program to be loaded into memory. It contains the program's main segment and the load file tables (jump table segment and pathname segment) needed to load dynamic segments and run-time libraries.

initialization segment: A segment in an initial load file that performs any initialization that the program may require

internal command: An APW Shell command that is executed by the shell program itself, rather than by a utility program.

INTERSEG record: A part of a relocation dictionary. It contains relocation information for external (intersegment) references.

jump table: A table contructed in memory by the System Loader from all Jump Table segments encountered during a load. The jump table contains all references to dynamic segments that may be called during execution of the program.

jump table directory: A master list in memory, containing pointers to all segments that make up the jump table.

jump table segment: A segment in a load file, created by the linker, that provides the information the loader needs to locate dynamic segments as they are needed during program execution. The loader creates a linked list in memory, called the **jump table**, that indicates the location of all jump table segments in memory.

K: 1024 bytes

kind: See **segment kind.**

language card: Memory with addresses between $D000 and $FFFF on any Apple II-family computer. It includes two RAM banks in the $D*xxx* space, called *bank-switched memory*. The language card was originally a peripheral card for the 48K Apple II or Apple II Plus that expanded its memory capacity to 64K and provided space for an additional dialect of BASIC.

language command: A command that changes the APW current language.

launch: To cause a program to be loaded into memory and to begin execution.

library dictionary segment: The first segment of a library file. It contains a list of all the symbols in the file together with their locations in the file. The linker uses the library dictionary segment to find the segments it needs.

library file: An object file containing object segments, each of which can be used in any number of programs. The linker can search through the library file for segments that have been referenced in the program source file. A library contains a **library dictionary segment**.

LinkEd: A command language that can be used to control the APW Linker.

linker: A program that combines files generated by compilers and assemblers, resolves all symbolic references, and generates a file that can be loaded into memory and executed.

link map: A listing, produced by the linker, that gives the name, length, and starting location of each segment in a load file.

load file: The output of the linker. Load files contain memory images that the system loader can load into memory, together with **relocation dictionaries** that the loader uses to relocate references.

load segment: A segment in a load file. Any number of object segments can go into the same load segment.

local symbol: A label defined only within an individual segment. Other segments cannot access the label. Compare with **global symbol.**

loop: A section of a program that is executed repeatedly until a limit or condition is met, such as an index variable's reaching a specified ending value.

low-level language: A programming language, such as assembly language, that is relatively close to the form the computer's processor can execute directly. One statement in a low-level language corresponds to a single machine-language instruction. Compare **high-level language.**

main segment: The first segment in the initial load file of a program. It is loaded first and never removed from memory until the program terminates.

macro: A single keystroke or command that a program replaces with several keystrokes or command. For example, the APW Editor allows you to define macros that execute several editor keystroke commands; the APW Assembler allows you to define macros that execute instructions and directives. APW also provides a library of predefined assembler macros.

macro assembler: A type of assembler that allows the programmer to define sequences of several assembly–language instructions as single pseudo–instructions called **macros.**

main segment: The first static segment (other than initialization segments) in the initial load file of a program. It is loaded first and never removed from memory until the program terminates.

MakeLib utility: A program that creates library files from object files.

Mark: The current position in an open file. It is the point in the file at which the next read or write operation will occur.

memory block: See **block.**

memory handle: The identifying number of a particular block of memory. A memory handle is a pointer to a master pointer to the memory block.

memory image: A portion of a disk file or segment that can be read directly into memory.

Memory Manager: A program in the Apple IIGS Toolbox that manages memory use. The Memory Manager keeps track of how much memory is available and allocates memory **blocks** to hold program segments or data.

memory-resident: (1) Stored permanently in memory as firmware (ROM). (2) Held continually in memory even while not in use. For example, ProDOS is a memory-resident program.

memory segment table: A linked list in memory, created by the loader, that allows the loader to keep track of the segments that have been loaded into memory.

Monitor: A program built into the firmware of Apple II computers, used for directly inspecting or changing the contents of main memory and for operating the computer at the machine-language level.

movable: A memory block attribute, indicating that the Memory Manager is free to move the block. Opposite of *fixed*. Only **position-independent** program segments may be in movable memory blocks. A block is made movable or fixed through Memory Manager calls.

native mode: The 16-bit operating state of the 65C816 processor.

object file: The output from an assembler or compiler, and the input to the linker. In APW an object file contains both machine-language instructions and instructions for the linker. Compare with **load file.**

object module format (OMF): The general format used in object files, library files, and load files.

object segment: A segment in an object file or in a library file.

OMF: Object module format.

OMF file: Any file in object module format.

op code: See **operation code.**

open: To allow access to a file. A file may not be read from or written to until it is open.

operand: In assembly language, a value used by an instruction or directive as an address or to calculate an address. In object module format, an operation code that is followed by a single value that constitutes part of an expression. The value following the operand opcode is acted on by an **operator.**

operation code: The part of an instruction or command that specifies the operation to be performed. Often called *op code* . In machine language, the operation code precedes the value to be acted on by the processor. In OMF, operation codes are used to identify types of records and types of operations in instructions.

operator: In object module format, an operation code that specifies an arithmetic or logical operation in an expression to be performed on one or two variables that precede it. The variables acted on by an operator are identified by **operand** opcodes that precede them.

page: (1)A portion of Apple IIGS memory that is 256 bytes long and that begins at an address that is an even multiple of 256. A memory block whose starting address is an even multiple of 256 is said to be *page aligned.* (2) An area of main memory containing text or graphical information being displayed on the screen.

parameter: A value passed to or from a command, function, or other routine.

parameter block: A set of contiguous memory locations, set up by a calling program to pass parameters to and receive results from an operating system or shell function that the program calls. Every ProDOS 16 and APW Shell call must include a pointer to a properly constructed parameter block.

partial assembly: A procedure whereby only specific segments of a program are assembled. If you have performed one full assembly followed by one or more partial assemblies on a program, the linker extracts only the latest version of each object segment to be included in the load file.

partial compile: A procedure whereby only specific segments of a program are compiled. If you have performed one full assembly followed by one or more partial compiles on a program, the linker extracts only the latest version of each object segment to be included in the load file.

partial pathname: A **pathname** that includes the **filename** of the desired file but excludes the volume directory name (and possibly one or more of the subdirectories in the path). It is the part of a pathname following a **prefix**; a prefix and a partial pathname together constitute a **full pathname.** A partial pathname does not begin with a slash because it has no volume directory name.

patch: To replace one or more bytes in memory or in a file with other values. The address to which the program must jump to execute a subroutine is *patched* into memory at load time when a file is **relocated.**

pathname: The complete name by which a file is specified. A pathname is a sequence of filenames separated by slashes, starting with the filename of the volume directory file and including every subdirectory file that the operating system must search to locate the file, in descending sequence of the subdirectory hierarchy. . A full pathname always begins with a slash (/) to indicate that the first name is a volume directory. See also **full pathname, partial pathname, prefix.**

pathname list: The part of the pathname table that contains the file pathnames.

pathname segment: A segment in a load file that contains the cross-references between load files referenced by number (in the jump table segment) and their pathnames (listed in the file directory). The pathname segment is created by the linker.

pathname table: A table constructed in memory by the loader from all individual pathname segments encountered during loads. It contains the cross-references between load files referenced by number (in the jump table) and their pathnames (listed in the file directory).

PC: See **program counter.**

pipeline: (1)To automatically execute two or more programs in sequence, where the output of the first file is the input to the next file and so on. (2)The entire sequential set of programs executed in this way; a program or file being processed by this sequence of programs is said to be *in the pipeline* or *in the pipe.*

pointer: A memory address at which a particular item of information is located. For example, the 65C816 stack register contains a pointer to the next available location on the stack.

position-independent: Code that is written specifically so that its execution is unaffected by its position in memory. It can be moved without needing to be relocated.

position-independent segment: A load segment that is movable when loaded in memory.

prefix: A portion of a **pathname** starting with a volume name and ending with a subdirectory name. It is the part of a full pathname that precedes a **partial pathname**; a prefix and a partial pathname together constitute a full pathname. A prefix always starts with a slash (/) because a volume directory name always starts with a slash.

prefix number: A code used to represent a particular prefix. Under ProDOS 16, there are eight prefix numbers, each consisting of a numeral followed by a slash: 0/, 1/,...,7/.

private code segment: A segment in an object file whose name is available only to other object-code segments within the same object file. The labels within a private code segment are local to that segment.

private data segment: A segment in an object file whose labels are available only to object-code segments in the same object file.

ProDOS: A family of disk operating systems developed for the Apple II family of computers. ProDOS stands for *Professional Disk Operating System* and includes both ProDOS 8 and ProDOS 16.

ProDOS 8: A **disk operating system** developed for standard Apple II computers. It runs on 6502-series microprocessors. It also runs on the Apple IIGS when the 65C816 processor is in 6502 **emulation mode.**

ProDOS 16: A **disk operating system** developed for 65C816 **native mode** operation on the Apple IIGS. It is functionally similar to **ProDOS 8** but more powerful.

program counter: A number, usually expressed in hexadecimal, that indicates the position of a byte in a machine-language program, counting sequentially from the beginning of the program.

purge: To temporarily deallocate a memory block. The Memory Manager purges a block by setting its master pointer to 0. All handles to the pointer are still valid, so the block can be reconstructed quickly. Compare **dispose.**

purge level: An attribute of a memory block that sets its priority for purging. A purge level of 0 means that the block is unpurgeable.

purgeable: A memory block attribute, indicating that the Memory Manager may purge the block if it needs additional memory space. Purgeable blocks have different **purge levels**, or priorities for purging; these levels are set by Memory Manager calls.

RAM disk: A portion of memory (RAM) that appears to the operating system to be a disk volume. Files in a RAM disk can be accessed much faster than the same files on a floppy disk or hard disk.

record: A component of an object module segment. All OMF file segments are composed of records, some of which are program code and some of which contain cross-reference or relocation information. Each record begins with an operation code that indicates the type of information to follow.

RELOC record: A part of a relocation dictionary that contains relocation information for local (within-segment) references.

relocate: To modify a file or segment at load time so that it will execute correctly at the location in memory at which it is loaded. Relocation consists of **patching** the proper values into address operands. The loader relocates load segments when it loads them into memory. See also **relocatable code.**

relocatable code: Program code that includes no absolute addresses, and that can therefore be relocated at load time.

relocatable segment: A segment that can be loaded at any location in memory. A relocatable segment can be static, dynamic, or position independent. A load segment contains a **relocation dictionary** that is used to recalculate the values of location-dependent addresses and operands when the segment is loaded into memory. Compare with **absolute segment.**

relocation dictionary: A portion of a load segment that contains relocation information necessary to modify the memory image immediately preceding it. When the memory image part of the segment is loaded into memory, the relocation dictionary is processed by the loader to calculate the values of location-dependent addresses and operands. Relocation dictionaries also contain the information necessary to transfer control to external references.

reference: (1)The name of a segment or entry point to a segmentl; same as *symbolic reference.* (2)To refer to a symbolic reference or to use one in an expression or as an address.

resolve: To find the segment and offset in a segment at which a symbolic reference is defined. When the linker resolves a reference, it creates an entry in a **relocation dictionary** that allows the loader to **relocate** the reference at load time.

restart: To reactivate a **dormant** program in the computer's memory. The System Loader can restart dormant programs if all their static segments are still in memory. If any critical part of a dormant program has been purged by the Memory Manager, the program must be reloaded from disk instead of restarted.

restartable: Said of a program that reinitializes its variables and makes no assumptions about machine state each time it gains control. Only restartable programs can be resurrected from a **dormant** state in memory.

root filename: The filename of an object file minus any filename extensions added by the assembler or compiler. For example, a program that consists of the object files `MYPROG.ROOT`, `MYPROG.A`, and `MYPROG.B` has the root filename `MYPROG`.

run-time library file: A load file containing program segments—each of which can be used in any number of programs—that the system loader loads dynamically when they are needed.

segment: A component of an OMF file, consisting of a header and a body. In object files, each segment incorporates one or more subroutines. In load files, each segment incorporates one or more object segments.

segment body: That part of a segment that follows the **segment header** and that contains the program code, data, and relocation information for the segment.

segment header: The first part of a program segment, containing such information as the segment name and the length of the segment.

segment kind: See **segment type.**

segment number: A number corresponding to the relative position of the segment in a file, starting with 1.

segment type: A classification of a segment based on its purpose, contents, and internal structure, as defined in the object module format. The segment type is specified by the KIND field in the segment header.

shell: A program that provides an operating environment for other programs and that is not removed from memory when the those programs are running. For example, the APW Shell provides a command processor interface between the user and the other components of APW, and it remains in memory when APW utility programs are running.

shell call: A request from a program to the APW Shell to perform a specific function.

shell-call block: A set of instructions and directives used to make a shell call from an assembly-language program.

shell command line: The line on the screen where the number-sign (#) prompt appears when you are in the APW Shell. When you enter a command, the characters you type appear to the right of the prompt on the command line.

shell load file: A load file designed to be run under a shell program. Shell load files are ProDOS 16 file type $B5.

65C816: The microprocessor used in the Apple IIGS.

source file: An ASCII file consisting of instructions written in a particular language, such as C or assembly language. An assembler or compiler converts source files into **object files.**

stack: A list in which entries are added (pushed) and removed (pulled) at one end only (the top of the stack), causing them to be removed in last-in, first-out (LIFO) order. The term *the stack* usually refers to the top portion of the **direct-page/stack** space; the top of this stack is pointed to by the 65C816's *stack register.*

stack pointer: The contents of the 65C816's stack register, consisting of a memory address pointing to the next available location on the 65C816's **stack.**

standard Apple II: Any computer in the Apple II family except the Apple IIGS. That includes the Apple II, the Apple II Plus, the Apple IIe, and the Apple IIc.

standard input: The default file or device (such as the keyboard) from which input is taken. If your program uses Text Tool Set calls or APW macros and libraries to get input, standard input is used.

standard output: The default file or device (such as the screen) to which output is sent. If your program uses Text Tool Set calls or APW macros and libraries to control output, standard output is used.

static segment: A segment that is loaded at program boot time and is not unloaded or moved during execution. Compare with **dynamic segment**.

string: An item of information consisting of a sequence of text characters (a *character string*) or a sequence of bits or bytes.

subdirectory: A directory within a directory; a file (other than the volume directory) that contains the names and locations of other files. Every ProDOS 16 directory file is either a volume directory or a subdirectory.

symbol: A character or string of characters that represents an address or numeric value; a symbolic reference or a variable.

symbolic reference: A name or label that is used to refer to a location in a program, such as the name of a subroutine. When a program is linked, all symbolic references are **resolved**; when the program is loaded, actual memory addresses are **patched** into the program to replace the symbolic references.

symbol table: A table of symbolic references created by the linker when it links a program. The linker uses the symbol table to keep track of which symbols have been resolved. At the conclusion of a link, you can have the linker print out the symbol table.

System Loader: The program that relocates load segments and loads them into Apple IIGS memory. The System Loader works closely with ProDOS 16 and the Memory Manager.

System Monitor: See **Monitor**.

system program: (1) A software component of a computer system that supports application programs by managing system resources such as memory and I/O devices. Also called *system software*. (2) Under ProDOS 8, a stand-alone and potentially self-booting application. A ProDOS 8 system program is of file type $FF; if it is self-booting, its filename has the extension .SYSTEM.

text-file format: The Apple IIGS standard format for text files and program source files.

token: The smallest unit of information processed by a compiler or assembler. In C, for example, a function name and a left bracket ({) are tokens.

Toolbox: A collection of built-in routines on the Apple IIGS that programs can call to perform many commonly–needed functions. Functions within the toolbox are grouped into **tool sets**.

tool set: a related group of (usually firmware) routines, available to applications and system software, that perform necessary functions or provide programming convenience. The Memory Manager, the System Loader, and QuickDraw II are tool sets.

top of form: The position on the paper in the printer to which the printer scrolls when it receives a form feed (Control-L) command.

unload: To remove a load segment from memory. To unload a segment, the System Loader does not actually "unload" anything; it calls the Memory Manager to either **purge** or **dispose** of the memory block in which the code segment resides. The loader then modifies the memory segment table to reflect the fact that the segment is no longer in memory.

User ID: An identification number that specifies the owner of every memory block allocated by the Memory Manager. User IDs are assigned by the User ID Manager.

utility: In general, an application program that performs a relatively simple function or set of functions such as copying or deleting files. An APW utility is a program that runs under the APW Shell and that performs a function not handled by the shell itself. MakeLib is an example of an APW utility.

volume: An entity that stores data; the source or destination of information. A volume has a name and a volume directory with the same name. Volumes typically reside in **devices**; a *device* such as a floppy disk drive may contain one of any number of *volumes* (on disks).

volume directory: The main directory file of a volume. It contains the names and locations of other files on the volume, any of which may themselves be directory files (called **subdirectories**). The name of the volume directory is the name of the volume. The pathname of every file on the volume starts with the volume directory name.

wildcard character: A character that may be used as shorthand to represent a sequence of characters in a pathname. In APW, the equal sign (=) and the question mark (?) can be used as wildcard characters.

word: A group of bits that is treated as a unit. For the Apple IIGS, a word is 16 bits (2 bytes) long.

zero page: The first page (256 bytes) of memory in a standard Apple II computer (or in the Apple IIGS computer when running a standard Apple II program). Because the high-order byte of any address in this part of memory is zero, only a single byte is needed to specify a zero-page address. Compare **direct page.**