Cortland Programmer's Workshop

Beta Draft

Writer: Paul R. Black
Apple Technical Publications

Version: 00.0 8/21/86
Engineering Part Number: 030-3130
Marketing Part Number: A2L6000

Copyright © 1986 Apple Computer, Inc. All rights reserved.

APPLE COMPUTER, INC.

This manual is copyrighted by Apple or by Apple's suppliers, with all rights reserved. Under the copyright laws, this manual may not be copied, in whole or in part, without the written consent of Apple Computer, Inc. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased may be sold, given, or lent to another person. Under the law, copying includes translating into another language.

© Apple Computer, Inc., 1986
20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010

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

[additional credit lines as needed]

Simultaneously published in the United States and Canada.

Contents

Preface
Roadmap
How to Use This Book
  What This Manual Contains
  What to Read When
  Visual Cues
Other Materials You'll Need
  Introductory Manuals
    The Technical Introduction
    The Programmer's Introduction
  Machine Reference Manuals
    The Hardware Reference Manual
    The Firmware Reference Manual
  The Toolbox Manuals
  The Programmer's Workshop Manual
  The Cortland Programming Language Manuals
  The Operating System Manuals
  All-Apple Manuals

Part I: Getting Started

Chapter 1. About the Programmer's Workshop
Cortland Concepts
Program Descriptions
  Shell
  Editor
  Assembler
  C Compiler
  Linker
  Debugger
  Utility Programs
  System Loader
  Memory Manager
Program Interactions

Chapter 2. How to Use the Shell and Editor
What You Need
Starting the Shell
  Running CPW on Floppy Disks
  Installing CPW on a Hard Disk
Entering and Executing Commands
  Entering Command Names, and Command Scrolling
  Multiple Commands
  Wildcards
  Parameter Prompts
  Partial Pathnames
  Device Numbers and Names
  Help Files
Listing a Directory

The Editor
Calling the Editor
Using the Editor
Using a Printer
Using Exec Files
Compiling (or Assembling) and Linking a Program
Using the Debugger
Using the Utilities
Launching Programs
Advanced Features

Chapter 3: Sample Program
Writing and Editing the Source Code
Creating Object Code: Compiling and Assembling
Creating Load Modules: Linking
Running Your Program

Part II: Cortland Programmer's Workshop Reference

Chapter 4. Shell
Redirecting Input and Output
Standard Prefixes
Pipelines
Partial Assemblies or Compiles
Command Types and the Command Table
Command Descriptions
ALINK
ASM65816
ASML
ASMLG
ASSEMBLE
C
CATALOG
CHANGE
CMPL
CMPLG
COMMANDS
COMPILE
COMPRESS
COPY
CREATE
CRUNCH
DEBUG
DELETE
DISABLE
DUMPOBJ
EDIT
ENABLE
EXEC
EXECUTE
FILETYPE
HELP
INIT

LINK
LINKED
MACGEN
MAKELIB
MOVE
PREFIX
PRODOS
QUIT
RENAME
RUN
SHOW
SWITCH
TEXT
TYPE
Exec Files
Passing Parameters Into Exec Files
Programming Exec Files
Variables
Logic Operators
Comments
Exec-File Commands
Break
Continue
Echo
Execute
Exit
Export
For—End
If—Else If—Else—End
Loop—End
Set
Unset
Example
LOGIN Files

Chapter 5. Editor
Modes
Insert
Escape
Auto Indent
Select Text
Automatic Wrap
Macros
Command Descriptions
Beep the Speaker
Bottom of Screen
Bottom of Screen / Page Down
Change
Clear
Copy
Cursor Down
Cursor Left
Cursor Right
Cursor Up

Cut
Define Macros
Delete
Delete Character
Delete Line
Delete to EOL
Delete Word
End of Line
Find
Help
Insert Line
Insert Space
Paste
Quit
Remove Blanks
Repeat Count
Return
Screen Moves
Scroll Down One Line
Scroll Down One Page
Scroll Up One Line
Scroll Up One Page
Search Down
Search Up
Search and Replace Down
Search and Replace Up
Set and Clear Tabs
Shift Left
Shift Right
Start of Line
Tab
Tab Left
Toggle Auto Indent Mode
Toggle Escape Mode
Toggle Insert Mode
Toggle Select Mode
Toggle Wrap Mode
Top of Screen
Top of Screen / Page Up
Undo Delete
Word Left
Word Right
Setting Editor Defaults

Chapter 6. Debugger
Getting Started
What You Need
Debugger Restrictions
Loading the Debugger
Loading Your Program
Debugger Display Screens
Selecting Displays
The Master Display
Register Subdisplay

Stack Subdisplay
Disassembly Subdisplay
RAM Subdisplay
Breakpoints Subdisplay
Memory Protection Subdisplay
Command Line
Setting Registers
Altering the Contents of Memory
Calculations
Display Configuration
Saving a Display Configuration
Printing
Other Command-Line Commands
Memory Display
Direct Page Display
Help Page
User-Program Display
Running Your Program
Single-Step and Trace Modes
The Command Filter
Memory Protection
Breakpoints
Using Monitor Routines

Chapter 7. Linker
Operation of The Linker
Object Files: Input to the Linker
Library Files
Partial Assemblies and Filename Conventions
Load Files: Output From the Linker
Diagnostic Output
Error Messages
Link Map and Source Listing
Symbol Table
Ending
Linking From a Command Line
Linking With a LinkEd Command File
LinkEd Command Descriptions
APPEND
COPY
EJECT
KEEP
LIBRARY
LINK
LIST
OBJ
OBJEND
ORG
PRINTER
SEGMENT
SELECT
SOURCE
SYMBOL
Examples

Part III: Inside the Cortland Programmer's Workshop

Chapter 8. Adding a Program to CPW
Compilers, Utilities, and Applications
CPW Utilities
Compilers and Assemblers
 Source File Format
 Identifying the Language Type
 Entry and Exit
 Command Precedence
 Output Files
 Partial Compiles
 Help Files
Interpreters

Chapter 9. File Formats
Text File Format
 Text File Specifications
 Examples
Object Module Format
 General Format for OMF Files
  Segment Types and Attributes
  Segment Header
  Segment Body
  Expressions
   Example
 Direct-Page/Stack Segments
 Library Files
 Load Files
  Memory Image and Relocation Dictionary
  Jump Table Segment
   Unloaded State
   Loaded State
  Pathname Table
  Initialization Segment
 Run-Time Library Files
 Shell Load Files

Chapter 10. Shell Calls

Making a Shell Call
The Call Block
Shell-Call Macros
The Parameter Block
Types of Parameters
Setting up a Parameter Block in Memory
Register Values
Call Descriptions
Direction ($010F)
Error ($0105)
Execute ($010D)
Get_Lang ($0103)
Get_LInfo ($0101)
Init_Wildcard ($0109)
Is_Window ($0112)
Next_Wildcard ($010A)
Read ($010B)
Read_Indexed ($01??)
Redirect ($0110)
Set ($0106)
Set_Lang ($0104)
Set_LInfo ($0102)
Stop ($0113)
Switch_Window ($010E)

Appendixes

Appendix A: Command Summary
Language Types
Shell
Exec Files
Editor
Debugger
LinkEd

Appendix B: Error Messages
Linker
Recoverable Errors
Fatal Errors

Glossary

List of Figures

Preface
P-1. Roadmap to the technical manuals

Part I: Getting Started

Chapter 1. About the Programmer's Workshop

1.1. Creating an Executable Program on the Cortland
1.2. OMF File Segmentation

Chapter 2. How to Use the Shell and Editor
2.1. Directory Example

Chapter 3: Sample Program
3.1. Sample C Source Code
3.2. Sample 65816 Source Code
3.3. Sample Symbol Table for C Program
3.4. Sample Symbol Table for Assembly-Language Program
3.5. Sample Symbol Table and Link Map From Link

Part II: Cortland Programmer's Workshop Reference

Chapter 4. Shell
4.1. Sample of a Command Table
4.2. Sample DUMPOBJ Segment Header
4.3. DUMPOBJ OMF-Format Segment Body
4.4. DUMPOBJ Disassembly-Format Segment Body
4.5. DUMPOBJ Hexadecimal-Format Segment Body

Chapter 5. Editor
5.1. Output of an Editor Macro

Chapter 6. Debugger
6.1. Sample Master Display
6.2. Sample Memory Display

Chapter 7. Linker
7.1. Sample Output of a LinkEd Command File

Part III: Inside the Cortland Programmer's Workshop

Chapter 8. Adding a Program to CPW

Chapter 9. File Formats
9.1. OMF File
9.2. Segment Header
9.3. Library Dictionary Segment

Chapter 10. Shell Calls

Appendixes

Appendix A: Command Summary

Appendix B: Error Messages

Glossary

List of Tables

Preface
P-1. The Cortland Technical Manuals

Part I: Getting Started

Chapter 1. About the Programmer's Workshop

Chapter 2. How to Use the Shell and Editor
2.1. Contents of a CPW Disk

Chapter 3: Sample Program

Part II: Cortland Programmer's Workshop Reference

Chapter 4. Shell
4.1. Standard Prefixes
4.2. CPW Language Types
4.3. CPW Commands
4.4. ProDOS Filetypes

Chapter 5. Editor

Chapter 6. Debugger
6.1. Disassembly Operand Formats

Chapter 7. Linker

Part III: Inside the Cortland Programmer's Workshop

Chapter 8. Adding a Program to CPW

Chapter 9. File Formats
9.1. Segment-Body Record Types

Chapter 10. Shell Calls
10-1. Shell Calls
Appendixes

Appendix A: Command Summary

Appendix B: Error Messages

Glossary

Preface

The Cortland Programmer's Workshop (CPW) consists of several programming languages, plus several programs that can be used by developers working with any of these programming languages. This manual describes the components of CPW that can be used with any of the programming languages: the shell, editor, linker, debugger, and utility programs; the programming languages are described in separate manuals. This manual is intended for experienced programmers and developers. It assumes that you are familiar with the Cortland computer and the Cortland operating system. The following section, "Roadmap," shows the relationship between this book and other books in the Cortland technical manuals suite. The section "Other Materials You'll Need" in this preface describes these books and makes recommendations as to which ones you'll need in order to develop programs for the Cortland computer.

Roadmap

The Cortland 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 Cortland, you may need to refer to a select few of the manuals, or you may need to refer to most of them.

The manuals are listed in Table P-1. Figure P-1 is a diagram showing the relationships among the different manuals. See the section "Other Materials You'll Need" in this preface for recommendations as to which of these books you'll need when developing programs for the Cortland computer.

Table P-1. The Cortland Technical Manuals

Title	**Subject**
Technical Introduction to the Cortland	What the Cortland is
Cortland Hardware Reference	Machine internals—hardware
Cortland Firmware Reference	Machine internals—firmware
Programmer's Introduction to the Cortland	Concepts and a sample program
Cortland Toolbox Reference: Part I	How the tools work
Cortland Toolbox Reference: Part II	More toolbox specifications
Cortland Programmer's Workshop	**This book: the development environment**
Cortland Programmer's Workshop Assembler Reference*	Using the CPW Assembler
Cortland Programmer's Workshop C Reference*	Using the CPW C Compiler
ProDOS 8 Technical Reference	ProDOS for Apple II programs
Cortland ProDOS 16 Reference	ProDOS 16 and the System Loader for Cortland
Human Interface Guidelines	Guidelines for all Apple computers
Apple Numerics Manual	Numerics for all Apple computers

*There is a Pocket Reference for each of these.

*****A new version of this figure is being prepared*****

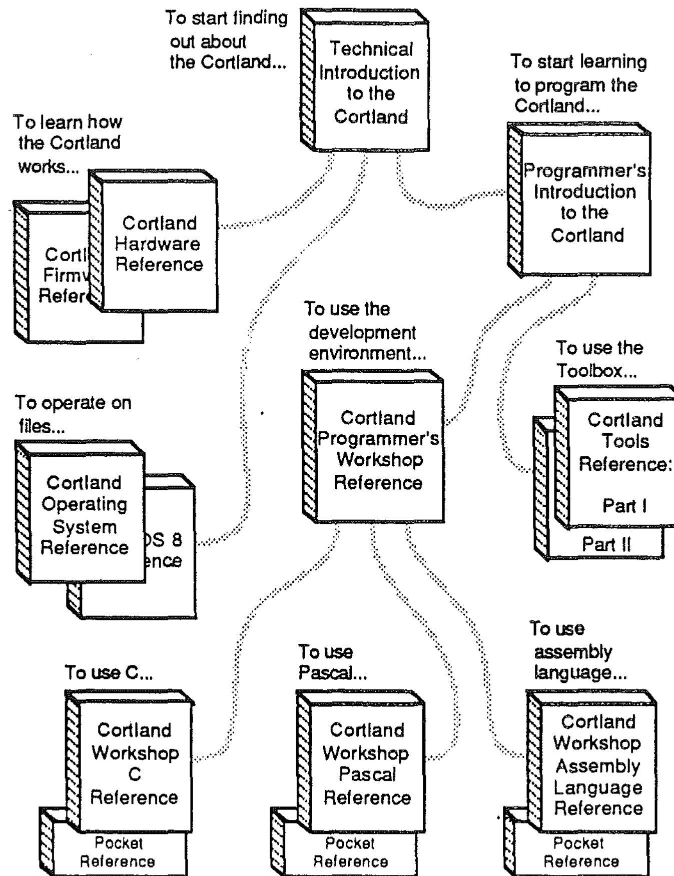


Figure P-1. Roadmap to the technical manuals

How to Use This Book

This section describes the contents of the *Cortland Programmer's Workshop* manual, gives guidelines as to which sections you should read for a specific purpose, and describes the visual cues used in this book to alert you to important material or that a certain word has a special significance other than regular text (for example, it might be a command).

What This Manual Contains

This book is divided into three parts, including 10 chapters, plus two appendixes, a glossary, and an index. The contents of these components are as follows:

Part I: Getting Started. The chapters in this part provide the minumum information you need to get started using the Cortland Programmer's Workshop.

Chapter 1: About the Cortland Programmer's Workshop. This chapter provides a general overview to CPW. It defines concepts that are essential to an

understanding of the CPW environment, and gives brief descriptions of the programs that comprise CPW.

Chapter 2: How to Use the Shell and Editor. This chapter provides an introduction to the abilities of CPW, and describes how you use CPW to write, compile or assemble, link, and run a program.

Chapter 3: Sample Program. This chapter provides a tutorial example that illustrates the creation of a program in the CPW environment. You are provided with C source code, plus source code of a subroutine in assembly language. The example lets you follow along step by step through the process of writing, assembling and compiling, linking, and running a simple multi-language program in the CPW environment.

Part II: Cortland Programmer's Workshop Reference. This part provides reference material on CPW commands, on the editor, debugger, and linker, and on the utility programs that are supplied with the CPW system.

Chapter 4: Shell. This chapter includes complete descriptions of every CPW command, along with descriptions of some CPW features too advanced to be covered in Chapter 2.

Chapter 5: Editor. This chapter includes complete descriptions of every CPW Editor feature and command.

Chapter 6: Debugger. This chapter constitutes an owner's manual for the CPW Debugger. It describes every debugger feature and command, and includes instructions for loading the debugger and the program you want to debug into memory.

Chapter 7: Linker. This chapter is a complete reference to the CPW Linker. It includes descriptions of every linker feature and function, including a complete description of linker command files (LinkEd files), which can be used to control many features of the linker not otherwise available to users.

Part III: Inside the Cortland Programmer's Workshop. This part of the manual contains reference material of use only to those programmers who wish to add a utility program or language compiler, assembler, or interpreter to CPW. It includes descriptions of Cortland file formats and calls to internal CPW functions.

Chapter 8: Adding a Program to CPW. This chapter describes the requirements that a utility or language compiler must satisfy to run under CPW.

Chapter 9: File Formats. This chapter describes the standard formats for text files and object files for the Cortland computer.

Chapter 10: Shell Calls. This chapter describes several internal CPW functions that utilities and compilers can (or must, in some cases) call when operating under CPW. It describes the procedure for calling these functions from assembly language.

Appendixes: The appendixes hold material for quick reference, or that you might want to refer to but that is inappropriate in a regular chapter.

Appendix A: Command Summary. A complete list, with brief descriptions, of all the shell, editor, and debugger commands, plus a list of all the language numbers assigned so far for CPW languages.

Appendix B: Error Messages. A list with descriptions and remedial action for all of the errors you can get while running the CPW Shell, Editor, Linker, or Debugger.

What to Read When

To get the most out of the Cortland Programmer's Workshop, you should use this manual in as efficient manner as possible. Herewith are some suggestions on how to proceed.

1. Whatever your background and experience, start with the "Other Materials You'll Need" section of this preface and Chapter 1. The Cortland is not quite like any other computer, so even if you helped design the Apple IIe and Macintosh, you have to become familiar with the peculiarities of the Cortland before you proceed.
2. If you are familiar with the Macintosh, especially the Macintosh Programmer's Workshop, you will be tempted to jump right in. Be cautious: the CPW command structure is a subset of the MPW commands, and some commands are entirely different. The editor operates in a fashion very unlike MPW. In CPW, there is only one set of commands to compile or assemble programs; for example, you can use the command `COMPILE` to compile a C program, assemble a 65816 assembly-language program, or assemble and compile a program consisting of an assembly-language file and an appended C source file. As a minimum, you should thumb through Chapters 2 and 4 to get an idea of what commands and features are available.
3. Read through the first few sections of Chapter 5, then thumb through the command-reference section to get an idea of what the editor can do.
4. The Cortland Programmer's Workshop's ability to compile multilanguage programs in one step is powerful and unique. If you intend to use more than one language in CPW, look through Chapter 3; you might want to follow along and perform the steps described to get some practice in using CPW.
5. By the time you get through step 4, you are pretty familiar with CPW. At this point you should go ahead and start programming, referring back to the reference material in Part II as necessary.
6. If you are doing assembly-language programming, you will probably find the CPW Debugger very useful. Read the first several sections of Chapter 6 and thumb through the command descriptions to get started using the debugger.
7. The first few sections of Chapter 7 will give you a deeper understanding of the CPW Linker. If you want to understand what is going on during the link process, read these sections. Don't bother with the section on LinkEd, though, unless you really need to do something during the link process that you can't do with the ordinary link commands. (An example would be to perform multiple searches of library files, or to link certain segments and ignore others.)
8. If you are doing complex system programming, or have a need to manipulate program segments or library searches in a nonstandard way, read through the sections on LinkEd in Chapter 7. For routine programming, you will never need this material.
9. If you are writing a program to run under CPW, 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 Cortland works, but unless you are actually adding programs to CPW, these three chapters are not required reading.

Visual Cues

Look for these visual cues throughout the manual:

By the Way: Notes like this contain sidelights or interesting pieces of information.

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

Important: Notes like this contain important information that you should read before proceding.

Caution: A cautionary note directs your attention to something that could cause problems with the software if you are not careful.

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.
```

Italics are used in commands to indicate parameters must be replaced with a value; for example, in the command

`DELETE` *pathname*

the word *pathname* refers to any valid ProDOS pathname; if the file you want to delete is `/CPW/MYPROGS/DONOTHING`, then this command would be:

```
DELETE /CPW/MYPROGS/DONOTHING
```

Other Materials You'll Need

The manuals and software you need in order to develop applications that run on the Cortland depend on the type of programming you are doing. For starters, you must be familiar with use of the Cortland computer, including the control panel. The following manual that came with your computer describes routine operation of the computer:

- *Cortland Owner's Guide*

The following sections describe the manuals in the Cortland technical manuals suite, 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 Cortland owners who need technical information. As introductory manuals, their purpose is to help you understand the features of the Cortland, particularly the features that are different form other Apple computers. Having read the introductory manuals, you should refer to specific refernce manuals for details about a particular aspect of the Cortland.

The Technical Introduction

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

Where the *Cortland Owner's Guide* is an introduction from the pont of view of the user, the *Technical Introduction to the Cortland* describes the Cortland from the point of view of the program. In other words, it describes the things the programmer has to consider while designing a program, such as the operating features the program uses and the environment in which the program runs.

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. To find out all about any one aspect of the Cortland, you should read one of the following specific technical manuals.

The Programmer's Introduction

When you start writing programs that use the Cortland user interface (with windows, menus, and the mouse), the *Programmer's Introduction to the Cortland* provides the concepts and guidelines you need. It is not a complete course in programming; rather, it is a starting point for programmers writing application for the Cortland. It introduces the routines in the Cortland Toolbox and the program environment they run under. It includes a simple event-driven program (that is, a program that waits in a loop until it detects an event such as a click of the mouse button) 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 this manual—the programming example in this manual is repeated in each of the language manuals (in the language appropriate to that manual). If you have never written an event-driven program, or used the Macintosh toolsets, then 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 *Cortland Hardware Reference* and the *Cortland 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 Cortland, especially if you are using a

high-level programming language such as C. If you are doing system programming, or writing programs that are designed to recognize whether they are running on the Cortland or older Apple II computer, then these books are essential. In any case, these books will give you a better understanding of the machine's features. They will also provide the reasons why some of those features work the way they do.

The Hardware Reference Manual

The *Cortland Hardware Reference* is required reading for hardware developers, and 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 *Cortland 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 *Cortland Firmware Reference* includes information about interrupt routines and low-level I/O subroutines for the serial ports, the disk port, and for the DeskTop Bus, which controls the keyboard and th mouse. The *Cortland Firmware Reference* also descirbes the Montior, a low-level programming and debugging aid for assembly-language programs.

The Toolbox Manuals

Like the Macintosh, the Cortland has a set of built-in routines (called the Cortland Toolbox) that can be called by applications to perform many commonly-needed functions. For example, there are Cortland tools that you can use to draw things on the screen, and tools that control 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 Cortland Programmer's Workshop languages.

The *Cortland 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 *Cortland 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 Cortland 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 Cortland graphics routines, however, or to develop an application that uses the Cortland desktop and mouse, then you'll find the Cortland Toolbox to be indispensable.

The Programmer's Workshop Manual

The development environment on the Cortland is the Cortland Programmer's Workshop (CPW). CPW is a set of programs that enable developers to create and debug application programs on the Cortland. The manual that describes CPW is the one you are reading, the *Cortland Programmer's Workshop Reference*. It includes information about the parts of

the workshop that all developers will use, regardless of which programming language they use: the shell, the editor, the linker, the debugger, and the CPW utility programs. It also provides the information you need in order to write a CPW utility or a language compiler or assembler for CPW.

The Cortland Programming Language Manuals

The Cortland 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 8 of this book. You can write different parts of a program in different CPW languages, then link them into a single load file using the Cortland Progammer's Workshop.

There is a separate reference manual for each programming language on the Cortland. Each manual includes the specifications of the language and of the Cortland libraries for the language, and describes how to write a program in that language. The manuals for the languages Apple provides are the *Cortland Programmer's Workshop Assembler Reference* and the *Cortland Programmer's Workshop C Reference*. Each of these manuals includes a sample program that performs the same functions as the program described in the *Programmer's Introduction to the Cortland* manual.

The Operating System Manuals

There are two operating systems that run on the Cortland: ProDOS 16 and ProDOS 8. Each operating system is described in its own manual: *ProDOS 16 Reference* and *ProDOS 8 Reference*. ProDOS 16 uses the full power of the Cortland and is not compatible with earlier models of Apple II. The *ProDOS 16 Reference* manual includes information about the System Loader, which works closely with ProSOS 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 programmer's, it is essential.

ProDOS 8, previously just called *ProDOS*, is compatible with the models of Apple II that use 8-bit CPUs. 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 Cortland manuals mentioned above, there are two manuals that apply to all Apple computers: *Human Interface Guidelines* and *Apple Numerics Manual*. The *Human Interface Guidelines* manual describes Apple's standards for the human interface to any program that runs on an Apple computer. If you are writing a commercial application for the Cortland, you should be fully 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 floating-point arithmetic. The functions of the Cortland 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 Cortland. The *Cortland Tools 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.

Part I

Getting Started

NOTES

Chapter 1

About the Programmer's Workshop

The Cortland Programmer's Workshop (CPW) is a development environment for the Cortland computer; it includes the following components:

- shell
- editor
- linker
- utility programs
- 65816 assembler
- C Compiler
- debugger

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

- ProDOS 16
- Cortland System Loader
- Cortland Memory Manager

> **Note:** Further support for developers is provided by the Cortland Tools. The Cortland Tools consist of a variety of routines in ROM and RAM that can be called from a program 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 CPW environment, but are *not* considered to be part of the Cortland Programmer's Workshop.

The Cortland Programmer's Workshop, then, consists of several programs that can be used by developers working with any of a variety of programming languages, plus several programming languages. This manual describes the components of CPW that can be used by any of the programming languages: the shell, editor, linker, debugger, and utility programs; the programming languages are described in separate manuals.

Cortland Concepts

This section defines the basic concepts and components of the Cortland system that you must be familiar with to use this manual. The terms shown in boldface are also included in the glossary at the end of the book.

The Cortland Programmer's Workshop uses three fundamental types of files: **source files**, **object files**, and **load files**. Source files consist of code and data following the conventions of a particular programming language; source files (and text files) correspond to the Cortland **text file format** defined in Chapter 9. Object files and load files conform to the Cortland **object module format** (OMF) defined in Chapter 9.

To understand the natures of source, object, and load files, you must first be familiar with the sequence by which you create an executable program from your source code. As illustrated in Figure 1.1, each source file must be converted into the 65816 machine language understood by the Cortland CPU. This conversion is done in several steps, as follows:

1. The source code is assembled or compiled. Depending on the programming language used in the source file, the CPW 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. (A symbolic reference is the name or label of a routine or block of data to which the program passes control during program execution.) Before the program is actually executed by the Cortland, all symbolic references must be **resolved**; that is, the location of the routine or data being referred to must be determined. 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 CPW programming languages. Each source file is converted into one or more object files by the CPW assembler and compilers.

2. The object files are input to the CPW **Linker**, which combines all of the object files into a single load file. 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, then it searches through any available **library files** for the missing routines. The linker removes the symbolic references and replaces them with entries in special tables it creates called **relocation dictionaries**. The load file, then, consists of blocks of machine-language code that can be loaded directly in memory (called **memory images**), with spaces reserved for the memory addresses of referenced routines, plus relocation dictionaries that contain the information necessary to patch the addresses into the memory images when the program is loaded into memory.

   The load files do not contain the actual memory addresses of references because, in general, Cortland programs are **relocatable**, that is, they can be loaded at any address in memory. The linker, therefore, does not know the final address at which any routine will be loaded.

3. At program execution time, the load file is loaded into memory by the **System Loader**. The loader calls the Cortland **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. The entire load file is not necessarily loaded into memory at one time; all OMF files are divided into **segments**, which can be processed independently. OMF-file segmentation is a fundamental Cortland concept, which we consider next.

   The Memory Manager is the Cortland toolset that allocates blocks of memory as needed, and keeps track of which blocks of memory are available. All applications should request blocks of memory from the Memory Manager rather than loading data directly into a preselected memory location.

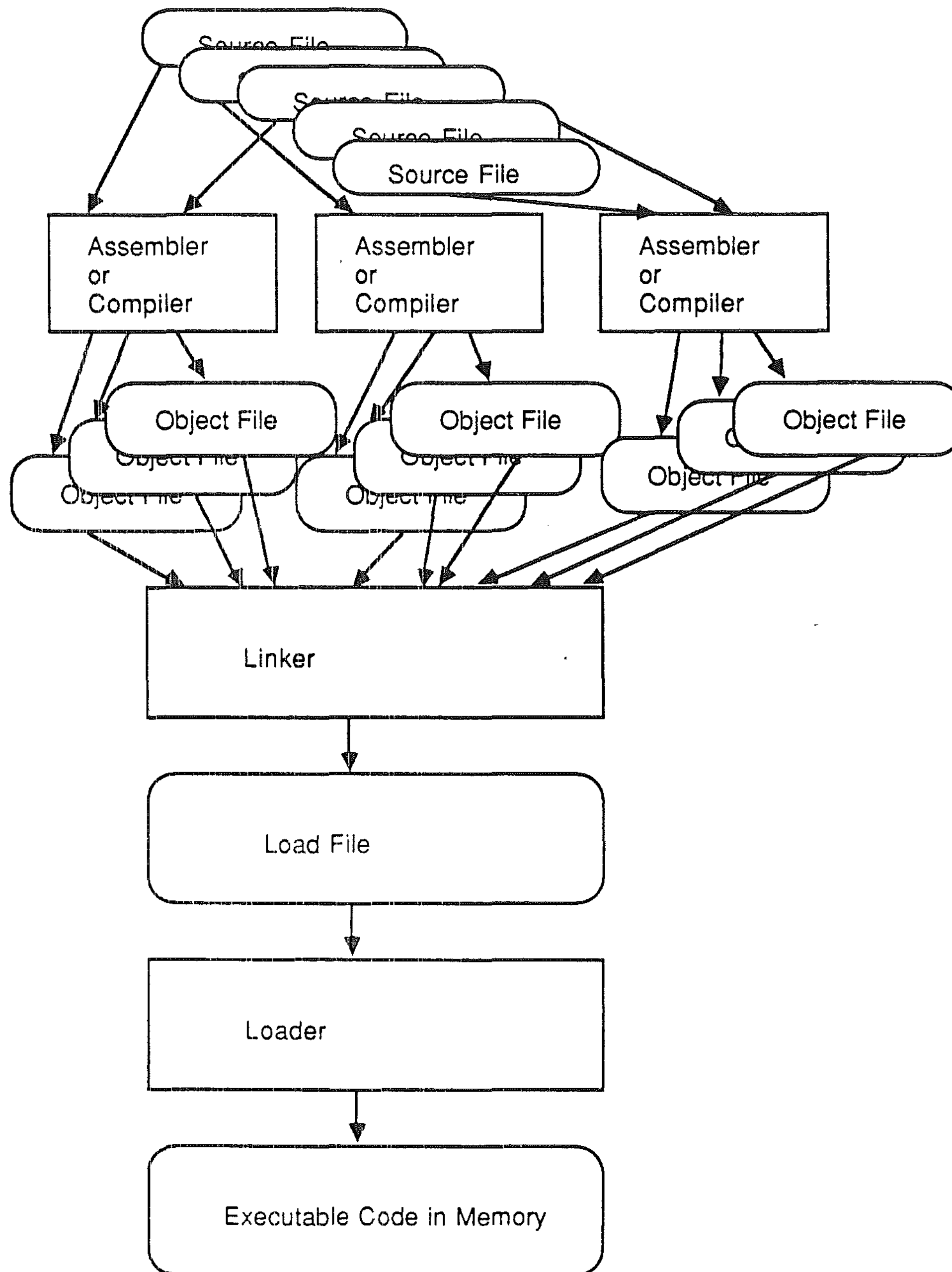


Figure 1.1. Creating an Executable Program on the Cortland

Every OMF file consists of one or more segments, as illustrated in Figure 1.2. Each segment consists of a **segment header** and the **segment body**. The segment header includes the following information:

- The size of the segment.
- The type of segment. (We'll get to segment types shortly.)
- The lengths of the label- and number fields in the segment body.

- 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, this may be the name of the subroutine contained in the segment, or a name you specify, depending on the programming language you used to write the source code.
- The name of the load segment that will contain the code generated by the CPW Linker for this segment (see the following discussion).
- Several other fields that need not concern us here (see the section "Segment Header" in Chapter 9 for a full description).

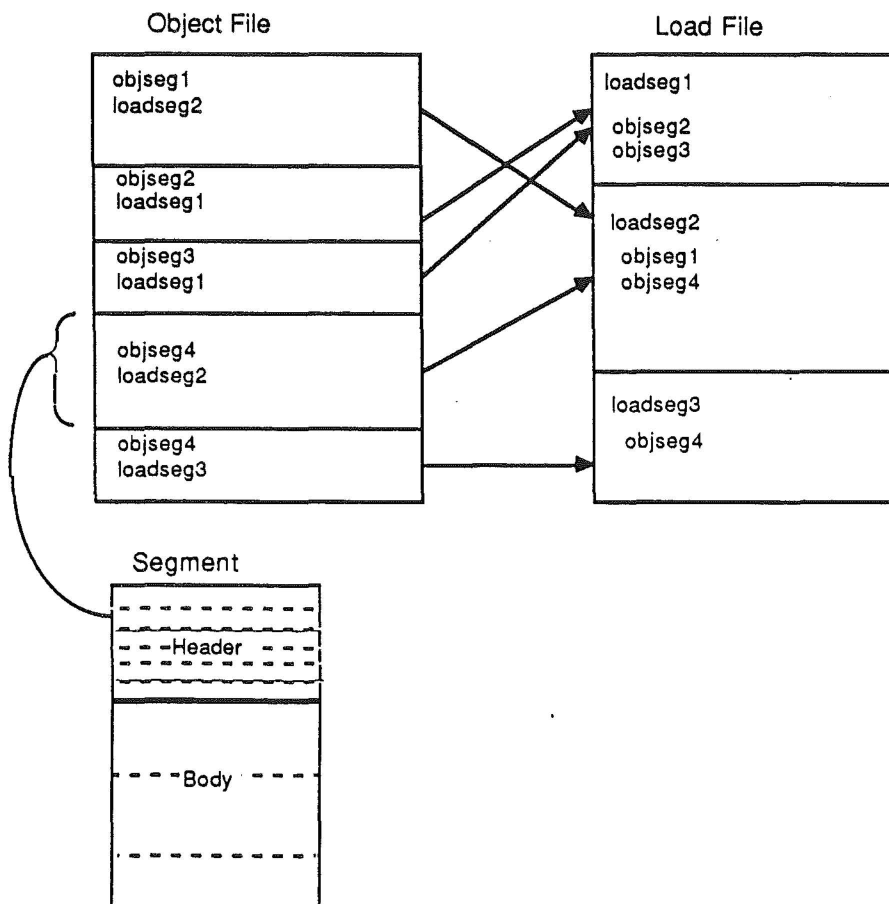


Figure 1.2. OMF File Segmentation

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. The object-segment name is used by the linker in resolving references; also, you specify the names of object segments when performing partial assemblies or compiles (described in the section "Partial Assemblies or Compiles" in Chapter 4) or when using LinkEd to extract specific segments for linking (see the section "Linking with a LinkEd Command File" in Chapter 7). Each segment in a program must have a unique object-segment name.

Each object segment is also assigned a load-segment name. Some programming languages, such as CPW Assembly Language and CPW C, let you assign your own load-segment name to an object segment; some compilers assign a load-segment name to the object segment for you. Any number of object segments can have the same load-segment name. The linker (unless instructed otherwise by LinkEd commands) places all object segments that share the same load-segment name into the same load segment.

For example, suppose your object file contains the following segments:

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 linker processes this file, 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.

On the Cortland computer, no single block of code can occupy more than 64 Kbytes of contiguous memory. 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 divided into one or two load segments wouldn't fit. The Cortland 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.

The object module format defines several segment types. A few are special-purpose segments used by the linker or the loader for processing files. The segment types with which you should be familiar in order to get the most out of this manual are as follows:

Object Segment: A segment in an object file.

Load Segment: A segment in a load file. Load segments contain memory images that the System Loader can load directly into memory, followed by a relocation dictionary that provides relocation information to the System Loader. Load segments can be static or dynamic.

Code Segment: An object segment that contains program code as opposed to data or variable definitions.

Data Segment: An object segment that contains program data.

Absolute Segment: A segment that can be loaded only at one specific location in memory. Programs written for older Apple II computers were often absolute; Cortland program ssegments should rarely if ever be absolute.

Relocatable Segment: A segment that can be loaded at any location in memory. A relocatable segment can be static or dynamic (see the next two definitions). 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.

Relocatable Code contains no references to absolute addresses, and so can be loaded at any location in memory. Code written to be run under ProDOS 16 on the Cortland computer must be relocatable to take advantage of the ability of the Memory Manager and System Loader to optimize the use of Cortland memory.

Static Segment: 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 system will run more efficiently if they are dynamic.

Dynamic Segment: A segment that can be loaded and unloaded during execution as needed. 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. As implemented on the Cortland computer, however, dynamic segments are much more versatile than overlays: 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.

The Cortland Memory Manager determines which dynamic segments to unload to make room for a new dynamic segment, and at what location in memory to load the new segment. A dynamic segment, therefore, uses only as much memory as that segment requires to run (for overlays, you must reserve a block of memory large enough for the *largest* overlay file used by the program), and frees the memory for *any* use when it is removed.

To specify that a load segment is dynamic, you must use a LinkEd command, as described in the section "Linking with a LinkEd Command File" in Chapter 7.

Library Dictionary Segment: A library file contains several segments, each of which can be used in any number of programs. The linker can search a library file for segments that have been referenced in the program source file. A library dictionary segment is the first segment of a library file; it contains the names and locations of all the other segments in the file. The linker uses the library dictionary segment to find the segments it needs.

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 a program.

The 65C816 processor 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 Cortland 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, you can use only the first 128 Kilobytes of Cortland memory, and cannot take advantage of the System Loader or Memory Manager. Native and emulation modes are discussed in the *Technical Introduction to the Cortland*, and described in detail in the *Cortland 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 Cortland System Loader. You must use ProDOS 8 to load such a file.

Program Descriptions

This section describes each of the programs included in the Cortland Programmer's Workshop, plus the System Loader and Memory Manager.

Shell

The shell program provides the interface that allows you to execute the desired CPW 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, and 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 Programmer's-Workshop programs. The format for making these calls is exactly like making a ProDOS 16 call.

Editor

The CPW Editor is the user interface to the shell. This full-screen text editor is designed for use with CPW assemblers and compilers. The editor's keyboard commands can be customized to the preferences of the user.

To use the CPW Editor, you use the shell `EDIT` command. If you are editing a pre-existing file, the editor is automatically set to the language of that file. If you are opening a new file, the editor is set to the last language used, or the last language selected with a shell command.

Assembler

This full-featured assembler allows users to write 65816 assembly-language programs for the Cortland computer, with full support for the Cortland object-module format, segmented object files, and library files. The Cortland Programmer'sWorkshop Assembler includes macros to facilitate assembly-language programming, and allows users to write their own macros and library files.

The CPW Assembler is specifically designed for writing relocatable code, since the CPW Linker, System Loader, and Memory Manager are all designed to work most efficiently with relocatable code.

The CPW Shell commands for assembling a 65816 assembly-language program are described in Chapters 2 and 4 of this manual; CPW Assembly Language is described in the *Cortland Programmer's Workshop Assembler Reference* manual.

C Compiler

The Cortland Programmer's Workshop C compiler is ***???***. The CPW Shell commands for compiling a C program are described in Chapters 2 and 4 of this manual; CPW C and C compiler options are described in the *Cortland Programmer's Workshop C Reference* manual.

Linker

The CPW Linker takes object files and file segments created by the CPW Assembler or any of the CPW compilers, and generates load files. The linker resolves external references and creates relocation dictionaries, which allow the System Loader to relocate code at load time. Normally, the linker is called by a shell command that provides a limited number of linker options.

In addition to the shell commands that call the linker, all functions of the CPW Linker can be controlled by using a language-like set of commands called *LinkEd*. LinkEd commands allow you to do such things as append or insert LinkEd source files in other LinkEd files, place specific object segments in specific load segments, create dynamic or static load segments, set load addresses for nonrelocatable code, search libraries, and control the output printed by the linker. The LinkEd commands can be appended to the last file of the source code, or can be compiled and executed separately by using the `ASSEMBLE`, `COMPILE`, or `ALINK` commands of the Cortland shell. LinkEd is provided for programmers who require maximum flexibility from the system; for most purposes, the ordinary link commands are completely adequate.

Since all Cortland Programmer's Workshop assemblers and compilers create object code that conforms to the same format, the CPW Linker can link together object files written in any combination of the development-environment languages.

Debugger

To facilitate the debugging of assembly-language programs, a debugger is provided that works with 65816 machine code. The CPW 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. It 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 Cortland registers, and the contents of the program's stack.

The debugger displays in 80-column mode only, but it 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.

Utility Programs

The Cortland Programmer's Workshop includes several programs, called CPW Utilities, that perform functions not built in to the shell. Utilities include:

- `DUMPOBJ`: lists an object-module-format file to standard output (usually the screen)
- `INIT`: initializes a disk
- `MACGEN`: generates a macro file
- `MAKELIB`: creates a library file from object files

Most utilities, referred to as *external commands* , are executed like built-in shell commands and are described in Chapter 4. A few utility programs may require more complex command sequences; if you add such a program to your system, refer to the documentation that came with it for instructions.

System Loader

The System Loader is a Cortland toolset that reads the files generated by the CPW Linker, relocates them (if necessary), and loads them into memory. The 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.

Memory Manager

This Cortland program allocates and frees blocks of memory as they are needed. It does the bookkeeping to keep track of what memory is used and which program owns each block of memory. The 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 together with relocatable code and the System Loader allows the Cortland to use memory in an extremely efficient manner, so that your program can be run under a shell program such as CPW, along with memory-resident utilities, character-font data files, and so on, without conflicts and without running out of available memory.

Program Interactions

This section illustrates the interactions among the various programs in the Cortland Programmer's Workshop by presenting a typical sequence of procedures and events. For this purpose, we assume that you are developing an application written mostly in C, with some routines written in 65816 assembly language. In this section, only the sequence of operations is listed; see Chapter 3 for step-by-step tutorial instructions to perform the

sequence described here. See the *Cortland ProDOS 16 Reference* manual for a complete description of the program load process.

1. Using a CPW Shell command, set the **current language** for CPW to C. (Every CPW file has a CPW language type; if you open a new file, it is given the current CPW language type.)
2. Call the CPW 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. You do not have to return to the shell between files; you can save one file and open another within the editor. Until you use a shell command to change it, or open a non-assembly-language file, the current language remains C.
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. In CPW Assembly Language, you can 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.

   Until you use a shell command to change it, or open a non-assembly-language file, the current language remains ASM65816.

5a. Quit the editor, and use the CPW Linker to link the object files into a load file. Normally, you can use the linker's default execution options to link the program. When you do not specify otherwise, the linker places all object segments with the same load-segment name into a single load segment.

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

   a. Using the editor, tie all of your source files together by placing an `APPEND` directive (in CPW Assembly Language) or a `#append` function (in C) at the end of each file but the last.

   b. From the shell, execute the compile-and-link command (`CMPL`). The object files output from the C compiler and those output from the CPW Assembler are all in the same format, and so are indistinguishable to the linker.

   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 CPW Assembler. When the assembler is finished, it returns control to the shell again, which calls the linker. The linker combines the object files, resolves references, writes the load file, and returns control to the shell.

5b. If you want to change load-segment assignments, or if you want to use dynamic load segments, you must use a LinkEd file. Write the LinkEd file like a language source file: first set the system language to `LINKED`, then use the editor to write the file.

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

   a. Using the editor, tie all of your source files together by placing an `APPEND` directive (in CPW 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.

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 CPW Assembler. When the assembler gets to the LinkEd file, it returns control to the shell again, which calls the linker. The linker, controlled by the commands in the LinkEd file, does the following:

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

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

6. 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 Cortland, 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 through the jump table. If the segment is already in memory, the loader transfers control to the segment. If not, the loader uses the jump table to locate the load file, segment, and offset of the subroutine, loads the segment into memory, and transfers control to the segment. The System Loader creates and maintains a table (the **memory segment table**) to keep 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. The System Loader keeps track of which segments are active (that is, which segments must be returned to by subroutines that have not completed executing), and does not allow those segments to be unloaded by the Memory Manager.

7. If the program does not run correctly, you can use the CPW 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.

8. Correct the source code and reassemble (or recompile) the program. You can do a partial assembly or compile to reassemble or recompile only the routine containing the bug.

9. Relink the program and rerun it. If you have used partial assembiles or partial compiles, the linker selects only the most recent version of each segment to put in the load file.

10. When the program is completely debugged, you can use the CRUNCH command to compress the files created by partial assemblies into a single object file. Then link the program one last time. Using CRUNCH is optional; if you have performed several partial assemblies, compressing the object files speeds up the link process.

Chapter 2

How to Use the Shell and Editor

The Cortland Programmer's Workshop Shell is the interface between CPW and the Cortland operating system. The CPW Editor is your interface with CPW. The shell provides a command interpreter to perform such functions as copying, moving, and deleting files, and running programs. The shell also provides an editor that you can use to write source code. You can assemble, compile, link, and run your programs with shell commands. In the Cortland 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.

This chapter introduces you to the use of the shell and editor. The most commonly-used commands and features are described, starting with the most fundamental and proceding to the more advanced. The most advanced and rarely-used features of CPW are deferred to the reference section of this manual.

What You Need

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

- A Cortland computer, or an Apple IIe computer with an installed Cortland-conversion kit.
- An installed Cortland memory-expansion board with 256K bytes of RAM, for a total of 512K bytes of RAM.
- A 3.5 inch disk disk containing the files shown in Table 2.1.
- Two UniDisk 3.5 disk drives, or one UniDisk 3.5 disk drive and a hard disk.
- Disks containing any other CPW languages you intend to use with this system. The files on these disks must be installed on the Cortland disk as described in the manuals that came with them.

> **Note:** If you haven't yet read the preface (who reads prefaces?), go back and read it now. In addition to providing a list of the manuals you'll need to develop programs for the Cortland, it explains the layout of this book, the interrelationships of the books in the Cortland reference-manual suite, and the notation and syntax used to describe commands in this book.

The CPW disk contains the files shown in Table 2.1. ***Please update as necessary. What are the final names for these files? Is this subject to change rapidly enough that we should not put it in the manual? Put it in an appendix?*** Use the index of this manual to get more information on any of these

files. To examine the contents of your CPW disk, boot the disk, type CATALOG, and press Return. To examine the contents of a subdirectory, include the pathname of the subdirectory; for example, to obtain a listing of the files in the subdirectory /CPW/UTILITIES/HELP/, use the following command:

```
CATALOG /CPW/UTILITIES/HELP
```

Table 2.1. Contents of a CPW Disk

File	Description
/CPW/	CPW subdirectory.
PRODOS	ProDOS system startup.
CPW.SYS16	The CPW Shell program.
SYSTEM/	Operating system and CPW system subdirectory.
P16	ProDOS 16 operating system kernel.
LOADER1	The System Loader.
LOADER2	The System Loader.
EDITOR	CPW Editor.
SYSCMND	List of CPW command names and command numbers. You can edit this file to add or delete commands.
LOGIN	CPW command file executed on startup (optional).
SYSEMAC	Editor macro file.
SYSTABS	Editor defaults file. You can edit this file to set editor defaults for any CPW language.*****?is this on the disk?*****
LIBS/	A subdirectory containing standard ProDOS 16 system libraries. *****?is this on the disk?*****
TOOLS/	A subdirectory containing all the RAM-based Cortland toolsets.
FONTS/	A subdirectory containing the fonts needed by CPW.
DESK.ACCS/	Cortland desk accessories.
SYSTEM.SETUP/	A subdirectory containing system programs to be executed at system boot time. *****what's in here for CPW??*****
TOOL.SETUP	A subdirectory containing ROM patches and the program that installs them.
LANGUAGES/	CPW Languages subdirectory. All compilers must be installed in this subdirectory.
ASM65816	CPW Assembler.
LINKED	CPW Linker.
UTILITIES/	CPW utilities subdirectory.
DEBUG	CPW Debugger.
DUMPOBJ	External command; CPW object-file dump routine.
INIT	External command.
MACGEN	External command.
MAKELIB	External command.
SWITCH	External command.*****?is this on the disk?*****
HELP/	Help-file subdirectory. This directory contains one help file for each CPW command.
LIBRARIES/	Linker libraries. The linker can search these libraries during a link.
A816.MSCN	Miscellaneous math libraries.
B816.IO	I/O libraries.
C816.INT2	2-byte math libraries.
D816.INT4	4-byte math libraries.
E816.INT8	8-byte math libraries.
H816.MSC	Miscellaneous libraries.
MACROS/	CPW Assembler macros subdirectory.
M65816.IO	I/O macros.
M65816.INT2MATH	2-byte math macros.
M65816.LONGMATH	4- and 8-byte math macros.
M65816.MSC	Miscellaneous macros.

Starting the Shell

The Cortland Programmer's Workshop disk is self-booting. To start up the CPW system, you need only insert the disk in the disk drive, and turn on the computer. You can also boot the system when the computer is already on by putting the CPW disk in the disk drive and pressing Apple-Control-Reset.

Important: Do *not* run CPW from the original product disk. To make a working copy of your CPW disk, use the following procedure.

1. Write-protect the original CPW disk, insert it in the disk drive, and turn on the computer. CPW should load from the disk. If you have a hard disk, turn it on and let it warm up. If you have two 3.5-inch disk drives, insert a blank disk in the second drive.

2. Type `SHOW UNITS` Return. A list of the disk drives attached to your system, together with the names of the volumes in those drives, appears on the screen. Make a note of the device numbers of each of your drives.

2. Type `INIT` *device* `/CPW2` Return, where *device* is the device number of the disk drive containing the disk you want to initialize. For example, if you have two 3.5-inch disk drives, this command should be `INIT .D2/CPW2`. (We are temporarily naming the new disk `/CPW2` to avoid any confusion with the original disk. The last step of the copy process will be to rename the disk `/CPW`.)

3. The prompt `Insert disk and hit any key to continue` appears on the screen. If you have only one disk drive, remove the CPW disk and insert the blank disk. If you have two disk drives, the blank disk is already in the second drive; press any key to initialize the disk.

4. When the initialization is complete, you can copy CPW. If you have two 3.5-inch disk drives, you can copy CPW directly from one drive to the other. If you have one 3.5-inch disk drive and a hard disk, it is easiest to copy CPW first onto the hard disk, then onto the new disk. If you have only one 3.5-inch disk drive, you will have to swap disks when you are prompted to do so.

   Note: It takes as long as 15 minutes or more to copy the CPW disk when you have to swap disks, and you have to sit there inserting and removing disks the whole time. Go get the beverage of your choice and put some music on the radio before you start.

   To copy the disk onto another floppy disk, place the CPW disk in the first (or only) disk drive, and enter the following command:

   ```
   COPY /CPW/= /CPW2/=
   ```

   To copy the disk using a hard disk as an intermediary, use the following commands (substitute the volume name of your hard disk wherever *hardisk* appears):

   ```
   CREATE /hardisk/CPW
   COPY /CPW/= /hardisk/CPW
   COPY /hardisk/CPW/= /CPW2
   ```

5. Remove the original CPW disk from the drive (if it's still in a drive), insert the new disk in the drive, and enter the following command:

   ```
   RENAME /CPW2 /CPW
   ```

   You now have a working copy of your CPW disk. Put the original in a safe place.

Each time you start CPW, it looks for file named `LOGIN` in the `/CPW/SYSTEM/` prefix. The `LOGIN` file should have a language type of `EXEC`; you can include any valid CPW command in this file. If it finds such a file, CPW executes it before doing anything else. You need not have a `LOGIN` file in your system; if there is no `LOGIN` file, CPW uses default settings for system parameters. You can use a `LOGIN` file to set system defaults such as setting a printer slot, to change default prefix assignments, to read a command table

containing command-name aliases, or even to execute a utility program. The next section gives instructions for creating a `LOGIN` files for using a hard disk, the section "Using a Printer" in this chapter tells you how to create a `LOGIN` file for initializing a printer. CPW commands and Exec files are described in Chapter 4.

Running CPW on Floppy Disks

You can run CPW on a single 800K 3.5-inch disk. The files included on the CPW disk take up most of the disk space, however, so you will probably find it more satisfactory to have a second 3.5-inch disk drive attached to your Cortland. You can set the system to use the volume name of the disk in the second disk drive as the default prefix to pathnames used in commands; to do this, use the `PREFIX` command. (The prefix that is assumed when none is specified is called the **current prefix.**) For example, if your programs are on a disk called `/MYPROGS` in the second disk drive, type the following command and press Return:

```
PREFIX /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 `/MYPROGS/`, you could use the following command to obtain a directory listing of the subdirectory `/MYPROGS/CSOURCE/`:

```
CATALOG CSOURCE
```

Note: Do *not* include a slash (/) before the pathname when you omit the current prefix from a pathname, or CPW 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`.

Keep the CPW disk in the first disk drive while you are running CPW, so the system can have access to the utility programs and help files on that disk.

If you have only one disk drive, you can load CPW, then remove the CPW disk and insert your program disk in its place. Execute the `PREFIX` command as described above. You are prompted to place the CPW disk online when it is needed. If the CPW disk is in the drive when you execute a command that requires your program disk, the message `Volume not found` appears on the screen. Place your program disk in the disk drive, and reexecute the command.

Installing CPW on a Hard Disk

To transfer CPW to a hard disk, first make a working copy of CPW on a floppy disk and copy CPW onto your hard disk as described in the beginning of the "Running the Shell" section. Then use the following procedure to create a `LOGIN` file on your working copy of CPW that will set up the system to run from the hard disk each time you load CPW. Substitute the volume name of your hard disk wherever the word *hardisk* appears in this procedure.

1. Boot CPW from the CPW working disk.

2. Type the following commands (press the Return key after each command):
```
EXEC
EDIT SYSTEM/LOGIN
```
3. You are now in the editor. 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 0 /hardisk /CPW
PREFIX 4 /hardisk /CPW/SYSTEM
PREFIX 5 /hardisk /CPW/LANGUAGES
PREFIX 6 /hardisk /CPW/UTILITIES
PREFIX 2 /hardisk /CPW/LIBRARIES
PREFIX 3 /hardisk
```
4. Press Apple-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 CPW. You should be able to remove the CPW working disk and run CPW from the hard disk after booting from the floppy disk.

Entering and Executing Commands

There are two main methods of entering commands in CPW:

1. Type in any CPW command, and press the Return key while in the CPW Shell command interpreter (that is, any time you are not in the CPW Editor or running another program).
2. Create a file of CPW commands with the language type EXEC. When you enter the name of an Exec file as a command, CPW executes the commands in the file as if they were typed from the keyboard.

Entering Command Names, and Command Scrolling

CPW requires every command to be entered in full, exactly as it appears in the **command table** (except that the command interpreter is not case sensitive). (The command table is a file containing every command name recognized by CPW; the shell consults the command table each time you enter a command. The command table is described in the section "Command Types and the Command Table" in Chapter 4.) It is not necessary for you to type in the entire command, however. Type in the first letter, or first few letters of the command, 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, if you type

```
CO
```

and press the Right-Arrow key, then the shell finds the first command-name in the command table that begins with CO, and prints the full command name:

```
COMMANDS
```

You can press the Up-Arrow and Down-Arrow keys to scroll through the last 20 commands that you have entered. For example, if you execute the `CATALOG` command, then execute the `COPY` command, and then press the Up-Arrow key, the CATALOG command reappears on the command line. Press the Down-Arrow to return the COPY command to the command line.

> **Note:** The CPW 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 because they are listed that way in the command table and help files.

You can add command aliases to the command table, if you like. 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 4 for instructions on modifying the command table.

Multiple Commands

You can enter several commands on one line; to do so, separate the commands with a semicolon (;). For example, to change the name of the file `WHITE` to `BLACK`, then open the file for editing, type in the following command line, and press the Return key:

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

You can use this technique in Exec files as well.

Wildcards

Many of the CPW commands require you to enter a filename. In many cases, you can substitite a special character, called a **wildcard**, for one or more of the characters in the filename. CPW recognizes two wildcard characters: the equal sign (=), and the question mark (?). These characters are used in an identical fashion to substitute for filenames or parts of filenames; the difference between them is that, if you use the question mark, then each time CPW finds a match for the character, it pauses and asks for confirmation before carrying out the command.

For example, suppose you want to write-protect every file in a directory called `/CPW/MYFILES`. The command `DISABLE W` *pathname* write-protects *pathname*; where *pathname* represents the prefix and filename of the file you want to write protect. To write-protect these files, use the following command:

```
DISABLE W /CPW/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 CPW delete it. To delete files in the directory `/CPW/MYFILES/` that have the extension `.BKUP`, with CPW asking for confirmation before deleting each file, use the following command:

```
DELETE /CPW/MYFILES/?.BKUP
```

Each time CPW finds a filename in the directory /CPW/MYFILES that ends in .BKUP, it stops and writes the name of the file to the screen. A cursor appears after the filename. To go ahead and delete the file, press Y (for yes). To leave the file alone and find the next match, press N (for no). To leave the operation alone and terminate the command, press Q (for quit).

You can specify as many or as few characters with a wildcard 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, CPW stops and waits for confirmation for every match.

You cannot use wildcards for directory pathnames or for the directory portion of a filename (that is, the prefix). You cannot use wildcards in filenames in certain commands. For example, you cannot use wildcards in the ASSEMBLE command or in the second filename of a RENAME command. Some commands accept wildcards, but use only the first filename matched; for example if you use a wildcard 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. Words shown in boldface are the ones you type in:

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

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

Parameter Prompts

If you enter a CPW command that requires one or more parameters (such as a filename), and do not include a required parameter, then CPW 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. Words shown in boldface are the ones you type in:

```
RENAME
File Name:   /CPW/MYFILE
File Name:   /CPW/MYFILE.OLD
```

Since the RENAME command requires two filenames as parameters, you are prompted for each in turn. If a wildcard would have been allowed in the command line, you can use one in response to the prompt.

Since you are not prompted for optional parameters, there are some operations you can not 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
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 (since no target directory was specified), CPW asks if you want to replace an existing file.

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

```
COPY
File Name:   MYFILE  /MYPROGS/CSOURCE
```

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

Partial Pathnames

When you execute a CPW command that requires a pathname, you can enter the full pathname, or you can enter a partial pathname, consisting of the filename and optionally one or more subdirectory names. If the pathname in the command does not begin with a slash (/), CPW assumes that a partial pathname is being used, and places the current prefix in front of the pathname in the command. When you first boot CPW, the current prefix is the boot volume; you can change the current prefix at any time with the `PREFIX` command.

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

```
CATALOG /CPW
CATALOG
```

When you follow the directions in the section "Installing CPW on a Hard Disk" in this chapter, the default current directory is */hardisk/*`CPW/`. If the name of the hard disk is HARDISK, then the current directory is `/HARDISK/CPW/` and the following two commands are equivalent:

```
PREFIX /HARDISK/CPW/MYPROGS
PREFIX MYPROGS
```

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

CPW uses other standard prefixes to find the CPW system files it needs, so CPW commands and utilities continue to work correctly when you change the current prefix. For

example, when you execute the `MAKELIB` command on a standard CPW floppy disk, CPW loads the file `/CPW/UTILITIES/MAKELIB`, no matter what the current prefix is set to. The prefixes that CPW searches for CPW system files can also be changed with the PREFIX command, as discussed in the sections "Standard Prefixes" and "PREFIX" in Chapter 4.

Device Numbers and Names

The Cortland Programmer's Workshop Shell assigns a device number to each I/O device currently online. Use the `SHOW UNITS` command to obtain a list of the device numbers and the ProDOS volumes currently in those devices. The unit names `.CONSOLE` and `.PRINTER` can also be used as device names. For example, suppose you have two UniDisk 3.5's, one Disk II drive, a hard disk, and a RAM disk on line. The `SHOW UNITS` command gives the following response (words shown in boldface are the ones you type in):*****Is this correct? Close? Vaguely related to the truth?*****

```
SHOW  UNITS
Units Currently On Line:

Device              Name

.D1                 /CPW
.D2                 /MYFILES
.D3                 /BLANK04
.D5                 /HARDISK
.D6                 /RAM5
.PRINTER
.CONSOLE
```

Note that device number .D4 is not listed, since ProDOS 16 expects disk drives to come in pairs. *****is that true?***** You can substitute a device number or device name anywhere you would have used a volume name. For example, to get a directory listing of the `LIBRARIES/` subdirectory, you could use the following command:

```
CATALOG .D1/LIBRARIES
```

The device name `.CONSOLE` represents the keyboard for input and the screen for output. The `.CONSOLE` and `.PRINTER` device names are primarily used in redirection of input and output. See the section "Redirecting Input and Output" in Chapter 4 for details.

Help Files

CPW includes a help file for each CPW command. To obtain a listing of the CPW commands, use the `HELP` command. To display a help file on any command, use the following command:

`HELP` *command*

where *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 CPW help files are all contained in the `HELP/` subdirectory in the `/CPW/UTILITIES/` subdirectory. They are standard ASCII text files, so you can edit them if you wish. If you create an alias for a command, 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 /CPW/UTILITIES/HELP/COMPILE /CPW/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`.

Listing a Directory

To obtain a directory listing, use the `CATALOG` command. For example, to get a listing of the contents of the `/CPW/` directory, enter:

```
CATALOG /CPW
```

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

```
/CPW/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   A=$2000
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 follows:

Name — The name of the file. Names are not case sensitive.

Type — The ProDOS 16 filetype. ProDOS 16 filetypes are described in the *Cortland ProDOS 16 Reference* manual. The filetypes most commonly used in CPW are as follows:

`SRC`	CPW source file
`OBJ`	CPW object file
`LIB`	Library file
`S16`	Load file that runs independently of any shell program
`EXE`	Load file that runs under a shell program
`STR`	Startup load file
`TXT`	ASCII text file

Blocks — The number of blocks on the disk occupied by this file. 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 you disable this attribute, the file will not be flagged as having been changed since the last time it was backed up.
- `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 a CPW source file, this field shows the CPW 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 wildcards, to list a specific subset of files on a subdirectory. You can use device numbers to list the directory on a volume even if you don't know the name of the volume.

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
```

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 `/CPW/MYPROGS`, use the following command:

```
CATALOG /CPW/MYPROGS/MYFILE
```

The Editor

The Cortland 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
- Execute shell commands

You control the editor with keyboard commands. All of the editor's features are described in detail in Chapter 5. 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*

where *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, then a new file with that name is opened.

Every CPW file has a CPW language type; if you open a new file, it is given the current CPW language type. If you open a preexisting file, CPW'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 CPW language type of any existing CPW source file with the `CHANGE` command, described in Chapter 4.

Files can have the following language types:

- `EXEC`: A CPW command file
- `TEXT`: An ASCII text file (ProDOS 16 filetype $B1)
- `PRODOS`: An ASCII text file (ProDOS 16 filetype $04)
- `ASM65816`: CPW 65816 Assembly-Language source code
- `C`: CPW C source code
- `LINKED`: CPW Linker command file
- others—each language compiler, assembler, interpreter, text formatter, or linker you add to CPW has a language name that can be assigned to 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. For example, if you want to create a C source file, type

   C |RETURN|

2. Type `EDIT MYFILE` |RETURN|. The editor opens a new file, named `MYFILE`.
3. Press |CTRL|-Q or ♂-Q. The Quit menu appears on the screen. Use the S selection to save the file.

Using the Editor

The CPW Editor allows you to enter and modify source files for all CPW programming languages, and to write text files with the CPW `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 5 (and summarized in Appendix A). In this section, enough commands are described to get you started using the editor.

The editor recognizes the Cortland extended ASCII character set. To get an extended character (high bit on), press the OPTION key and the character key at the same time.

When you press the |ESC| key, the editor enters a special mode called escape mode. Escape mode has the following features:

- Every letter key executes a command. If no other command is defined for a key, pressing the key terminates escape mode and returns you to text-entry mode. You cannot enter text while in 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 |ESC| and before you execute the command. For example, to scroll down 10 lines, type |ESC| `10 C`. 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 |ESC| again.

To get started using the editor, use the following commands.

Help ♂-?: Use this command to see the help files for editor commands.

Cursor Movement	↑ ↓ ← →: Use the arrow keys to move the cursor around on the screen.
Top of Screen /Page Up	♂-↑: The cursor is moved to the top of the screen. If it 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	♂-↓: The cursor is moved to the bottom of the screen. If it is already at the bottom of the screen, the entire screen is scrolled down one page.
Toggle Insert Mode	♂-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. 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, then 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	CONTROL-♂-I: 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 CPW language type; see the section "Setting Editor Defaults" in Chapter 5.
Scroll Down One Line	ESC C: Use this command to scroll the screen down one line (that is, to move all the *text* on the screen *up* one line). This is an escape-mode command; you must press Esc again after executing the command to return to editing mode.
Scroll Up One Line	ESC E: Use this command to scroll the screen up one line. This is an escape-mode command.
Delete Character Left	DELETE: Press this key to delete the character to the left of the cursor.
Delete	♂-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 Delete operation without deleting the text from the file, press |ESC| instead of RETURN.)
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's are possible.
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 (normally `CPW/`). (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

ȯ-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. (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

ȯ-V: The contents of the `SYSTEMP` file are copied to the current cursor position.

Search Down

ȯ-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. Type in the string for which you wish to search, and press Return (press Esc to cancel the operation). Searches are not case sensitive, and include all occurrences of the string, whether it is imbedded 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

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

ȯ-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.

Search and Replace Down

ȯ-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. Type in the string for which you wish to search, and press Return. Searches are not case sensitive, and include all occurrences of the string, whether it is imbedded 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 (press Esc instead of Return to cancel the operation). Enter the string with which you want to replace the search 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` Return to replace the string and search for the next occurrence; `N` Return to leave this occurrence of the string unchanged and search for the next occurrence; or `Q` Return to leave the string unchanged and terminate the search and replace operation. When the operation is finished, the cursor returns to its starting point.

Type `Q` 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 type `A` Return or `M` Return, 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. If the string is not found, the following message appears on the screen:

```
String Not Found
```

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 5 for a complete description of all the options.

As you become familiar with CPW, you can study Chapter 5 to learn the full capabilities of the editor, and the fastest way to obtain results. Advanced features described in Chapter 5 include the following:

- Editor macros: A macro allows you to substitute a single keystroke for up to 128 predefined keystrokes. A macro can contain text, editor commands, and (by including the Enter key) shell commands.
- Eding modes : The operation of the editor depends on several modes that can be toggled between different states; each CPW language has a default setting for each mode. You can toggle the modes while in the editor, and can change the default setting for any language (see the section "Setting Editor Defaults" in Chapter 5).
- 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 CPW output that would normally go to the screen. You can use this facility together with the `TYPE` command to print out 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. There must be at least one space between *pathname* and the output redirection operator (>).

By default, CPW attempts to print from a printer connected to slot 1, sends no initialization string to the printer, sends a form-feed command to the printer after every 66 lines, does not add a line feed after a carriage return, and does not count the characters in each line.*****what are the defaults, like, really?***** You can use the following commands to override these defaults:

```
SET PRINTERSLOT slotnum

SET PRINTERINIT string

SET PRINTERLINES linenum

SET PRINTERLINEFEED value

UNSET PRINTERLINEFEED

SET PRINTERCOLUMNS colnum
```

Where:

slotnum The number of the slot containing your printer-driver PC board; an ASCII number from 0–7. The default value for *slotnum* is 1, the built-in printer port on the Cortland.

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 caret (^) to indicate a control character. A space is interpreted as a space character, $20.

> **Caution**: 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
```

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

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, then no form-feed characters are sent.

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

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. The printer driver uses this parameter to count lines on a page in the case that your printer automatically inserts a carriage return–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 *colnum* to 0 and the printer driver will count a new line only when a carriage return is sent.

You can include these commands in a `LOGIN` file so they are executed each time you load CPW. If you have created a `LOGIN` file to run CPW from a hard disk, place the printer-setup commands at the end of that `LOGIN` file. If you are running CPW from a flopppy disk, use the following procedure to create a LOGIN file:

1. Boot CPW from the CPW working disk.
2. Type the following commands (press the Return key after each command):
   ```
   EXEC
   EDIT SYSTEM/LOGIN
   ```
3. You are now in the editor. Type the printer-setup commands, one per line, 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.
4. After the printer-setup commands, type the following command:
   `EXPORT` *variablelist*
   where *variablelist* is the list of printer variables you just set. For example, if you used the `SET PRINTERSLOT` and `SET PRINTERINIT` commands, you must follow them with the command
   `EXPORT PRINTERSLOT PRINTERINIT` *****IS THAT RIGHT??*****
5. Press Apple-Q. When the editor's Quit menu appears, press S to save the file, then E to return to the shell.
6. To test the setup, reboot CPW. Turn on your printer, and type the following command:
   ```
   TYPE SYSTEM/LOGIN >.PRINTER
   ```
   The contents of the `LOGIN` file should be sent to your printer.

To redirect to the printer the output of any command, use the output redirection operator, >, anywhere on the command line. For example, to send a listing of the directory `/CPW/MYPROGS/` to the printer, use the following command:

```
CATALOG /CPW/MYPROGS  >.PRINTER
```

See the section "Redirecting Input and Output" in Chapter 4 for more information on sending output to the 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 current 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.
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 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 4.

To execute an Exec file, type the pathname of the file as if it were a CPW 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
```

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

Exec-file variables, such as parameters passed into the file or those defined with `SET` commands, are normally local to that 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 to all Exec files. The `EXPORT` and `EXECUTE` commands are described in detail in the section "Exec Files" in Chapter 4.

Compiling (or Assembling) and Linking a Program

The Cortland Programmer's Workshop uses a single format for object files, and a single set of commands for compiling or assembling programs written in any CPW source language. Therefore, you can write different modules or routines of your program in different CPW languages, and compile, link, and run the program all in one step. For the vast majority of programs written in the CPW environment, the compiler and linker defaults are quite adequate; the following is a typical sequence for writing, compiling, and linking a program. A tutorial example of this procedure is given in Chapter 3. See the discussion of the `ASML` command in Chapter 4, the section "Partial Assemblies or Compiles" in Chapter 4, and the section "Linking with a LinkEd Command File" in Chapter 7 for more versatile (and complicated) ways to control assemblies, compiles, and links.

Note: To enhance rhetorical simplicity, the words *compiler* and *compile* are used in this section to include *assembler* and *assemble*.

1. Set the system language to the language type of the source code you intend to write, open a file for editing, and write the source code for the first module of your program. Save the file to disk.

2. Execute the shell `COMPILE` (or `ASSEMBLE`) command.

   Important: If you do not specify a `KEEP` filename in either the source file or the `COMPILE` command, then no object file is saved to disk.

   a. If the CPW compiler finds a fatal error (one that prevents the compile from continuing), it writes out an error message to standard output (normally the screen) and waits for you to press any key. When you press a key, the compiler passes control to the CPW Editor, which loads the source file that the compiler was working on, placing the line that caused the error at the top of the screen.

   b. If the compiler finds a nonfatal error, it finishes processing the program, writes out the error messages, and returns control to the shell.

3. If your first attempt was not successful, correct the source code and try again; repeat this process until the module compiles (or assembles) successfully. Remember to save the source file each time you make changes; the disk file is updated only when you save it.

   When the compiler processes the file, it takes the first segment that will be executed when the program is run, and places it in an object file with the `KEEP` filename you specified, and the extension `.ROOT`. All other segments (if any) are placed in a second object file with the same `KEEP` filename and the extension .A.

   You now have three files on disk: the source code and two object-code files (one with the extension `.ROOT` and one with the extension `.A`). For example, if the source file is named `MYFILE` and the filename you specified in the `KEEP` parameter is `KEEPFILE1`, then you have the following files on disk:

   - `MYFILE` the source file
   - `KEEPFILE1.ROOT` the object file containing the first segment to be executed
   - `KEEPFILE1.A` the object file containing the remaining segments of the program

4. Write the next module. This module need not be in the same programming language as the first module. Give this module a different source filename than the first module, and a different `KEEP` filename.

5. Execute the shell `COMPILE` command. Debug the module and recompile as necessary until successful.

6. Repeat steps 4 and 5 for each module of the program, until you are sure that each module compiles successfully.

7. Execute the `LINK` command, specifying the filenames of all of the object files in the program. Include a `KEEP` filename in the `LINK` command.

   Important: If you do not specify a `KEEP` filename in the `LINK` command, then no load file is saved to disk.

The CPW Linker combines all object segments that have the same load segment name into the same load segment, and places the entire program into a single load file with the KEEP filename you specified. (If you are confused by all this talk about object segments and load segments, go back and reread the section "Cortland Concepts" in Chapter 1.)

In this example, you now have the following files on disk:

- `MYFILE` the source file
- `KEEPFILE1.ROOT` the object file containing the first segment to be executed
- `KEEPFILE1.A` the object file containing the remaining segments of the first module of the program
- `KEEPFILE2.ROOT` the object file containing the first segment of the second module of the program
- `KEEPFILE2.A` the object file containing the remaining segments of the second module of the program
- ... object files containing segments of the other modules
- `KEEPFILE` the load file

If you prefer, you can write the entire program, including modules in several languages, and compile and link them all at once. Each module except the last should end in an APPEND directive (or the equivalent). Use the CMPL command to compile and link the program. Every time a CPW compiler executes an APPEND directive, it checks the CPW language type of the file being appended. If the language doesn't match that of the compiler, then the compiler returns control to the shell, which calls the appropriate compiler to continue processing the program. If the compile is successful, the CPW Linker is called automatically. The linker processes the file, writes out any errors, and (if the link was successful), writes the load file to disk.

By the way: The compiler may check the language type of a file when executing a COPY directive, but does not return control to the shell; instead, the compiler returns an error if any file being copied into the program does not match the language of the compiler.

Using the Debugger

Once you have created an executable load file, you can use the CPW Debugger as an aid in debugging it. The debugger can execute a load file in memory one instruction at a time, showing you the contents of Cortland 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 Cortland 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 CPW Debugger is described in detail in Chapter 6.

Using the Utilities

The Cortland Programmer's Workshop Shell includes most of the functions that you need to write, compile, link, run, and debug programs. A few functions, however, are implemented as separate routines designed to be run under the shell; these are referred to as CPW utility programs, or utilities. Most CPW utilities, such as `INIT` (which formats disks), require no more input than any other shell command; in this manual these functions are referred to as **external commands.** You use them just like other CPW commands, but they must be present in the `CPW/UTILITIES/` subdirectory. A few utilities may perform more complex functions, and 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.

Launching Programs

Under ProDOS 16 on the Cortland Computer there are two principal types of executable load files: system load files (filetype $B3), and shell load files (filetype $B5). After you have written a program, you can use the `FILETYPE` command (described in Chapter 4) to assign a filetype to it.

- Programs of filetype $B3 take over complete control of the computer; they do not operate under a shell program. CPW itself is an example of such a program. To call a program of filetype $B3, the calling program executes a ProDOS 16 `QUIT` call, shutting itself down and clearing the screen. 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 *Cortland ProDOS 16 Reference* manual.) A type $B3 file must make Cortland tool calls to set up the environment it needs in which to run, including the graphics or text screen it needs, and the input it accepts.
- Programs of filetype $B5 run under a shell program (such as the CPW Shell); they do not remove the shell from memory. The shell uses System Loader calls to load the program, and transfers control to it in full native mode via a JSL instruction; when the program terminates, it returns control to the shell via a ProDOS `QUIT` call. For more information on writing a program to run under a shell, see the sections "CPW Utilities" in Chapter 8 and "Shell Load Files" in Chapter 9. A shell load file uses the environment set up for it by the shell under which it runs; if a shell load file is launched by ProDOS 16, then ProDOS 16 sets up the standard 80-column screen and keyboard input for it.

To launch a program of either filetype from the CPW shell, enter the prefix and filename of the file as a command. For example, if you want to run a program called `STAR.WARP`,

which is in the subdirectory /MYPROGS/GAMES/, type the following line and press Return:

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

ProDOS 8 BIN files are not executed by the CPW Shell.

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 Cortland Programmer's Workshop. CPW has many additional capabilities not covered in this chapter. The following list gives some indication of other functions and where to find them in this manual. See the Preface for a description of this book, chapter by chapter. Use the table of contents and the index to find the specific topics in which you are interested.

- You can **pipline** commands; that is, you can automatically use the output of one command as the input of another. See "Piplining" in Chapter 4.
- 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 4.
- You can link two or more object files that have different root filenames into the same load file. See the discussion of the LINK command in Chapter 4.
- You can use CPW Assembly-Language directives to control which object segments go into which load-file segments. See the section "Load Segments" in Chapter 7, and the *Cortland Programmer's Workshop Assembler Reference* manual.
- You can list the segments, segment-header contents, and segment contents of any file on disk in object file format. See the discussion of the DUMBOBJ command in Chapter 4.
- You can control the CPW 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 includes the following capabilities (see the section "Linking With a LinkEd Command File" in Chapter 7).
  - append one LinkEd source file to another
  - copy one LinkEd source file into another
  - cause the printer to skip to a new page
  - open a file for output
  - search a library
  - link any number of program files
  - control the listing of segment names
  - start segments at specified locations
  - set the program counter
  - control the printed output

- start a load segment
- place specific object segments in a load segment
- choose specific segments to link
- control output of the symbol table

• You can specify which subdirectories are searched for specific routines; for example, you can change the subdirectory searched for utility programs from `/CPW/UTILITIES/` to `/PRODOS/WORKSHOP/EXT.COM/`. See the section "Standard Prefixes" and the discussion of the `PREFIX` command in Chapter 4.

• You can use shell commands to initialize disks, alphabetize or otherwise reorder directories, move, copy, and rename files, and create subdirectories. See the section "Command Descriptions" in Chapter 4.

• You can read in a new command table at any time to define new command names or aliases, or 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 4.

• You can create your own link-time library files. See the discussion of the `MAKELIB` command in Chapter 4.

Copyright © 1986 Apple Computer, Inc. All rights reserved.

Chapter 3

Sample Program

This provides a tutorial example that illustrates the creation of a program in the CPW environment. It includes a main routine in C, and a subroutine in assembly language. You are shown how to use the CPW Editor to create source files in bothe languages, and how to compile, assembl, link, and run the program. Each language manual includes an example in the language of that manual alone, but only this chapter gives an example that uses more than one language. Of course, if you have only the CPW Assembler, then you won't be able to use this example; but then you don't need to know how to do a multi-language program either.

*****Whenever I get some source code for this program, I'll try out these procedures and see if they work as advertized. Meanwhile, please read through them to see if I did anything wrong, left anything out, or put anything in that shouldn't be there.*****

Writing and Editing the Source Code

Use the following steps to write the source code for the C routine shown in Figure 3.1:

1. Boot CPW, and type the following command to set the system default language type (the **current language**) to C. (To execute a CPW command, press the Return key.)

   ```
   C
   ```

2. Call the editor to open file called SAMPLEC with this command:

   ```
   EDIT SAMPLEC
   ```

3. Type in the program in Figure 3.1. Use the cursor keys to move around in the file. The Delete key deletes the character to the left of the cursor. The Tab key moves the cursor for indenting subroutines. Other basic editor commands are given in the section "The Editor" in Chapter 2.
4. Press Control-Q or ♂-Q to quit the editor. Press S to save the file to disk, then press E to exit the editor and return to the shell.
5. Type the following command to set the current language to 65816 assembler.

   ```
   ASM65816
   ```

6. Call the editor to open file called SAMPLEA with this command:

   ```
   EDIT SAMPLEA
   ```

7. Type in the program in Figure 3.2. The Tab key is now set for assembly-language syntax.
8. Press Control-Q or ♂-Q to quit the editor. Press S to save the file to disk, then press E to exit the editor and return to the shell.

These examples should be fairly short, but as interesting as possible. The program should do something you can test, like taking an input text file and changing all the text to uppercase. It should probably be a $B5 file, and should follow the rules for $B5 files, including checking the CPW Shell identifier, the command line, and its userID. It must have several segments.

Figure 3.1. Sample C Source Code

Figure 3.2. Sample 65816 Source Code

Creating Object Code: Compiling and Assembling

To compile and assemble your programs, use the following commands:

```
COMPILE +L +S SAMPLEC KEEP=SAMPLEOBJC
```

```
ASSEMBLE +L +S SAMPLEA KEEP=SAMPLEOBJA
```

The +L and +S options provide a listing of the source code and a symbol table, respectively. The source code should look like Figures 3.1 and 3.2. The symbol tables

should look like Figures 3.3 and 3.4. *****Does C provide source-code listings and symbol tables? Do I need any other options for the C routine?*****

The following files should be on your disk after using these commands:

`SAMPLEC`	The C source code
`SAMPLEA`	The 65816 source code
`SAMPLEOBJC.ROOT`	The first object segment created by the C compiler
`SAMPLEOBJC.A`	The rest of the object segments created by the C compiler
`SAMPLEOBJA.ROOT`	The first object segment created by the assembler
`SAMPLEOBJA.A`	The rest of the object segments created by the assembler

Alternately, you can compile and link both files in one operation. To do this, you must add a line to the file `SAMPLEC` as follows:

1. Reopen the file in the editor with the following command:

   ```
   EDIT SAMPLEC
   ```

2. Press ⌘-9 to jump to the end of the file. Add the following line to the file:

   ```
   #append SAMPLEA
   ```

3. Press ⌘-Q to quit the editor, S to save the file, and E to exit the editor.
4. Now when you use the following command, the shell calls the C compiler to compile the C routine, then calls the CPW Assembler to assemble the 65816 routine:

   ```
   COMPILE +L +S SAMPLEC KEEP=SAMPLEOBJ
   ```

   The output should be the same as before (with the addtion of the one extra line in the C routine). The following files should be on your disk after using this command:

`SAMPLEC`	The C source code
`SAMPLEA`	The 65816 source code
`SAMPLEOBJ.ROOT`	The first object segment created by the C compiler
`SAMPLEOBJ.A`	The rest of the object segments created by the C compiler
`SAMPLEOBJ.B`	The object segments created by the assembler

Figure 3.3. Sample Symbol Table for C Program

Figure 3.4. Sample Symbol Table for Assembly-Language Program

Creating Load Modules: Linking

Here are three ways to link the object files you have just created:

1. If you did *not* add the `#append` command to the end of the C routine, use the following command to link the object files into a single executable load file:

```
LINK +L +S (SAMPLEOBJC SAMPLEOBJA) KEEP=SAMPLE
```

The +L and +S options cause the linker to print out a link map and symbol table for the link, as shown in Figure 3.5. The load file is named SAMPLE.

The following files should be on your disk after using this command:

SAMPLEC	The C source code
SAMPLEA	The 65816 source code
SAMPLEOBJC.ROOT	The first object segment created by the C compiler
SAMPLEOBJC.A	The rest of the object segments created by the C compiler
SAMPLEOBJA.ROOT	The first object segment created by the assembler
SAMPLEOBJA.A	The rest of the object segments created by the assembler
SAMPLE	The load file

2. If you *did* add the #append command to the end of the C routine, use the following command to link the object files into a single executable load file:

```
LINK +L +S SAMPLEOBJ KEEP=SAMPLE
```

The following files should be on your disk after using this command:

SAMPLEC	The C source code
SAMPLEA	The 65816 source code
SAMPLEOBJ.ROOT	The first object segment created by the C compiler
SAMPLEOBJ.A	The rest of the object segments created by the C compiler
SAMPLEOBJ.B	The object segments created by the assembler
SAMPLE	The load file

3. To compile, assemble, and link the two routines all in one step, add the #append command to the end of the C routine and use the following command:

```
CMPL +L +S SAMPLEC KEEP=SAMPLE
```

The following files should be on your disk after using this command:

SAMPLEC	The C source code
SAMPLEA	The 65816 source code
SAMPLE.ROOT	The first object segment created by the C compiler
SAMPLE.A	The rest of the object segments created by the C compiler
SAMPLE.B	The object segments created by the assembler
SAMPLE	The load file

Figure 3.5. Sample Symbol Table and Link Map From Link

Running Your Program

To run the program you just created, use the following command:

`SAMPLE` ***Does it need any parameters? Does it prompt for any?***

To compile, assemble, and link the two routines all in one step, add the `#append` command to the end of the C routine and use the following command:

```
CMPLG +L +S SAMPLEC KEEP=SAMPLE
```

Part II

Cortland Programmer's Workshop Reference

Chapter 4

Shell

The Cortland Programmer's Workshop Shell provides the interface between the user and the Cortland Programmer's Workshop, and between CPW and the Cortland operating system. The shell provides the following features:

- A command interpreter for interactive keyboard input of commands.
- Facilities for copying, renaming, deleting, and moving files.
- A full-featured text editor.
- 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 standard prefixes.
- Commands for assembling, compiling, linking, and running programs.
- A debugger that you can use to step through and debug code in memory.
- The ability to execute other Cortland programs.

This chapter provides a reference to the CPW Shell; it describes the shell commands and Exec files. It explains how to redirect input and output, set standard prefixes, and pipeline commands. The editor, linker, and debugger are described in separate chapters. See Chapter 2 for general instructions on starting and running the shell and editor, and using other CPW features. See Chapters 9 and 11 for the information you need in order to add a program to the Cortland Programmer's Workshop.

Redirecting Input and Output

Standard input is usually through the keyboard, though it can also be from an Exec file or the output of a program; standard output is usually to the screen though it can be to a printer or another program or disk file. You can redirect standard input and output for any command by using the following conventions on the command line:

<*inputdevice*	Redirect input to be from *inputdevice*.
>*outputdevice*	Redirect output to go to *outputdevice*.

>>outputdevice	Redirect output to be appended to the current contents of *outputdevice.*

The input device can be the keyboard or any text or source file. To redirect input to be from the keyboard, use the device name .CONSOLE.

The output device can be the screen, the printer, or any file. If the file named does not exist, CPW opens a file with that name. To redirect output to the screen, use the device name .CONSOLE; to redirect output to the printer, use .PRINTER.

> **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, then the current system language is used as the CPW language type of that file. Use the SHOW LANGUAGE command to find out what the current system language is. To change the current system language, type the name of the language you wish to use, and press Return. Use the SHOW LANGUAGES (note the difference: LANGUAGES rather than LANGUAGE) command to see which language types are defined in the command table. CPW language types are described in the section "Command Types and the Command Table" in this chapter.

> **Important:** If a disk file is used for input or output, the disk must remain online 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.

> **Important:** The input or output redirection symbol *must* be preceded by a space, even if it's the first character in the command line.

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

```
ASSEMBLE MYPROG >.PRINTER

ASSEMBLE >.PRINTER MYPROG

 >.PRINTER ASSEMBLE MYPROG
```

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

```
CATALOG >>CATSN.DOGS

 >>CATSN.DOGS CATALOG
```

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

```
 <ANSWER.FILE ASSEMBLE MYPROG

ASSEMBLE <ANSWERS MYPROG
```

```
ASSEMBLE MYPROG <ANSWERS
```

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

> **Important:** The output of programs that do not use standard output cannot be redirected.

Error messages also normally go to standard output, usually the screen, and can be redirected independently of other output. To redirect error output, use the following conventions on the command line:

>&*outputdevice*	Redirect error output to go to *outputdevice*.
>>&*outputdevice*	Redirect error output to be appended to the current contents of *outputdevice*.

Error-output devices follow the same conventions as those described above for standard output.

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

Standard Prefixes

CPW searches for files in certain preset subdirectories. Five such subdirectories are used, and are assigned to five of the ten standard prefixes supported by ProDOS 16. You can assign any subdirectory to the eight ProDOS 16 prefixes by using the CPW-Shell `PREFIX` command. The standard prefixes are used as shown in Table 4.1.

Table 4.1. Standard Prefixes

Prefix Number	Use	Default
0	ProDOS 16 system	*boot.prefix/*
1	program execution	*boot.prefix/****??***
2	CPW library	*boot.prefix*/LIBRARIES/
3	CPW work	*boot.prefix/*
4	CPW system	*boot.prefix*/SYSTEM/
5	CPW language	*boot.prefix*/LANGUAGES/
6	CPW utility	*boot.prefix*/UTILITIES/
7	undefined	
8	undefined	
9	undefined	

The boot prefix is the one from which the Cortland is started; the program execution prefix is the one from which CPW is loaded. If you have CPW on a 3.5 inch disk, the boot prefix and program execution prefixes are normally `/CPW/`. If you boot your Cortland from a 3.5-inch CPW disk, but run CPW on a hard disk, the boot prefix would still be `/CPW/` but the program execution prefix might be `/HARDISK/CPW/`. The ProDOS 16 system prefix (also called the "current prefix") is the one that is assumed when you use a

partial pathname in a shell command. The default for this prefix is the program execution prefix. If you are using a diskette, the default ProDOS 16 system prefix is usually the volume name of the disk; if you are using a hard disk, you should set this prefix to the name of the CPW subdirectory. The following files and subdirectories are found in the CPW Disk or subdirectory. A full list of files on the CPW disk is shown in Table 2.1.

- `PRODOS`
- `CPW.SYS16`
- `SYSCMND`
- `SYSTEM/`
- `LANGUAGES/`
- `UTILITIES/`
- `LIBRARIES/`
- `MACROS/`

CPW looks in the CPW system prefix for the following files:

- `EDITOR`
- `SYSTABS`
- `SYSEMAC`
- `SYSCMND`
- `LOGIN`

> **Note:** The `LOGIN` file is an Exec file that is executed automatically at load time if it is present. See the section "Exec Files" in this chapter for instructions on creating a `LOGIN` file.

The language prefix contains the CPW Linker command language (LinkEd), the CPW Assembler, and any other assemblers, compilers, and text formatters that you have installed in your copy of CPW.

The utility prefix contains all of the CPW utility programs except for the editor, assembler, and compilers. It includes the debugger and 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" in this chapter.

The files in the CPW library prefix are automatically scanned by the linker to resolve any references not found in the object files being linked. (You can also use the LinkEd `LIBRARY` command to scan library files in other prefixes as described in Chapter 7.) CPW comes with several library files that support the CPW Assembler macros; you can also create your own library files. For more information on creating and using CPW Assembler library files, see the discussion of the `MAKELIB` command in this chapter, and the *Cortland Programmer's Workshop Assembler* manual.

The work prefix is used by CPW programs for temporary files. For example, the `MACGEN` command writes macros to a temporary file on the work prefix called `SYSMAC` as an

intermediate step in creating a macro library. The editor uses the work prefix to create the SYSTEMP file, which it uses for the Copy, Cut, and Paste commands. Several CPW commands work faster if you set the work prefix to a RAM disk. If you have sufficient memory in your system to do so (256K bytes should be sufficient), use the Cortland control panel to set up a RAM disk, then use the `PREFIX` command to change the work prefix. If the RAM disk is named `/RAM5`, for example, use the following command:

```
PREFIX 3 /RAM5
```

Pipelines

CPW lets you automatically execute two or more programs in sequence, directing the output of one program to the input of the next. 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 CPW. The first temporary file opened is assigned an *n* of 0; if a second `SYSPIPE`*n* file is opened while `SYSPIPE0` is still open, then it is named `SYSPIPE1`, and so forth.

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.

For example, if you had a utility program called `UPPER` that took characters from standard input, converts them to uppercase, and writes them to standard output, you could use the following command line to write the contents of the text file `MYFILE` to the screen as all uppercase characters:

```
TYPE MYFILE|UPPER
```

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

```
TYPE MYFILE|UPPER >MYUPFILE
```

The `SYSPIPE`*n* files are not deleted by CPW after the pipeline operation is complete; thus, you can use the editor to examine the intermediate steps of a pipeline as an aid to finding errors. The next time a piplene 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 considerably by taking advantage of CPW's ability to perform partial compiles or assemblies. 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; for example, the files `MYFILE.ROOT`, `MYFILE.A`, and `MYFILE.B` all have the same root filename : "`MYFILE`".

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` automatically link the program after compiling or assembling it; `ASMLG`, `CMPLG`, and `RUN` 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; see the description of the `ASML` command for discussions of all the command parameters.

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 a CPW Assembly-Language source file, the label of a `START`, `PRIVATE`, or `DATA` 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.
- In high-level languages (such as CPW C) there is normally a one-to-one correspondence between subroutines and object segments: each subroutine becomes a separate object segment, and the object-segment name is the same as the subroutine name. See the reference manual for the CPW language you are using for any exceptions to this rule.
- 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, then, you must list all object-segment names in uppercase. For case sensitive languages, on the other hand, you must list all object-segment names exactly as they appear in the source code. CPW C is case sensitive. CPW Assembly Language is case insensitive unless you have used the CASE ON or OBJCASE ON directives in the source file.

- You can include in one NAMES parameter list the names of all the segments you want to use, whether the segments are all in one file, or you are compiling several files at once (by appending one source file to another with APPEND directives).
- Be sure to include a KEEP directive in the file, or a KEEP parameter in the command line, with the same root filename for the object file as you used for the original compile or assembly.

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

When you assemble or compile a program, you must use a KEEP directive (or equivalent) in the source code, or the KEEP parameter in the command line, to specify a filename for the output (if you don't, the program is compiled, but the output is not saved). 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 (by the compiler) in a file with the filename extension .ROOT, and the remaining segments are placed in a file with the extension .A. If the filename you specify is MYPROG, for example, then the file containing the first segment to be executed is named MYPROG.ROOT and the file containing the remaining segments is named MYPROG.A.

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

1. If you include a NAMES parameter on the command line to request a partial assembly or compile, then 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.
2. If the compile involves more than one language, then the first compiler or assembler usually creates the .ROOT and .A files, the second compiler creates the .B file, and so on.

Note: You can use the CRUNCH command described in this chapter to combine all the alphabetic-extension files into one .A file.

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 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 7 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 case occur, because (in this example) the linker would start the next link with the file `MYFILE.D`.

As an example of a partial compile and assembly, assume you have written a program in 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
```

The CPW Shell processes the program as follows:

1. The shell checks the language type of `MYPROG`, and calls the CPW 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 CPW 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 assembly-language segments are entered in uppercase, since the assembler is not case sensitive (unless you use the `CASE ON` or `OBJCASE ON` directives), while the segment names for the C routine must be entered exactly as they appear in the source code. 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; 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 CPW 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, you use the following command:

```
LINK 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 Cortland Programmer's Workshop includes a large number of commands that perform functions from listing a disk directory to compiling a program. There are three types of commands in CPW: **internal**, **external**, and **language**.

- An *internal* command is one included in the CPW Shell; internal commands are resident in memory whenever you are in the Cortland Programmer's Workshop.
- An *external* command is a separate CPW **utility** program; these programs are in the utility prefix (normally `/CPW/UTILITIES/`), and are loaded from disk when you execute the commands.
- A *language* command is the name of a language recognized by CPW. When you execute a language command, that language becomes the current default. Any new file opened for editing with the `EDIT` command uses the default language. If you open an existing file for editing, the default language changes automatically to match that file. The language 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.

  Note: The existance 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 `LANGUAGES/` subdirectory to see which compilers are installed in your copy of CPW.

The CPW 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 CPW language types include all assemblers and compilers recognized by CPW, plus ASCII text files (language-type `PRODOS`), LinkEd command files (`LINKED`), and shell command files (`EXEC`). Compilers and assemblers plus `LINKED` are included in the `CPW/LANGUAGES/` subdirectory on disk; the code to support `EXEC`, `COMMAND`, and `PRODOS` text files is included in the shell and editor. All CPW compiler and assembler source files, `LINKED`, `EXEC`, and `COMMAND` files are ProDOS 16 file type $B0; and `PRODOS` text files are ProDOS 16 file type $04.

Table 4.2 shows some of the language types that CPW recognizes; of these, `PRODOS`, `TEXT`, `ASM65816`, `LINKED`, and `EXEC` are included when you purchase CPW. For a complete list of language numbers that have been assigned for CPW, see Appendix A. The assignment of a language number does not necessarily imply that that language is currently available; to find out which CPW-compatible language compilers are currently available, see your authorized Apple dealer:

Table 4.2. CPW Language Types

Language	Number	Use
ASM6502	2	6502 assembler source
ASM65816	3	65816 assembler source
BASIC	4	CPW BASIC source
C	10	CPW C source
EXEC	6	Command file
LINKED	13	CPW Linker command language
PRODOS	0	ProDOS 16 text file (ProDOS 16 filetype $04)
TEXT	1	CPW text file

All CPW source files have a ProDOS 16 filetype of $B0; the CPW language type is not recognized by ProDOS 16. CPW `TEXT` files are standard-ASCII files with ProDOS 16

filetype $B0 and a CPW language type of `TEXT`. ProDOS 16 files are standard-ASCII files with ProDOS 16 filetype $04; these files are recognized by ProDOS 16 as text files. See the *Cortland ProDOS 16 Technical Reference* for a discussion of ProDOS 16 filetypes.

Commands can be added, deleted, and renamed by editing the command table. The command table is read at load time from a file called `SYSCMND` in the system prefix. You can also 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 aliases for commands, or to delete commands. To change the contents of the 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 or tabs, as illustrated in Figure 4.1. 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 (*), then 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. Then 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.)

   Caution: For a program to be restartable, it must reinitialize all variables and arrays each time it starts. 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 will occur the first time the shell tries to restart that program.

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 A. For external commands, the third field is blank.

Warning: Using a number other than an existing one for an internal shell command will almost certainly cause the system to crash when you try to execute the command.

The third field can be followed by a space or tab and a comment. Blank lines are ignored; you can also create a comment line by starting it with an asterisk (`*`), an exclamation point (`!`), or a semicolon (`;`).

```
ALINK        C    3
ASM65816     L    3
ASML         C    1
ASMLG        C    2
ASSEMBLE     C    3
COMMAND      L    12
COMMANDS     C    13
COMPILE      C    3
COMPRESS     C    32
COPY         C    5
DUMPOBJ      U
ECHO         C    29
```

Figure 4.1. Sample of a Command Table

In Figure 4.1, you can see that `ALINK`, `ASSEMBLE`, and `COMPILE` all have the same command number; they are aliases. `ASM65816`, on the other hand, is a language command; language number 3 is not related to command number 3.

The commands delivered with CPW are shown in Table 4.3.

Table 4.3. CPW Commands

Command	Use	Type
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
C	Change default language to CPW C	Language
CAT	List the disk directory	Internal
CATALOG	List the disk directory	Internal
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
COMPILE	Compile the program	Internal
COMPRESS	Compress and/or alphabetize the disk directory	Internal
CONTINUE	Exec-file command	Internal
COPY	Copy a file, directory, or volume	Internal
CREATE	Create a new subdirectory	Internal
CRUNCH	Combine object modules formed by partial compiles or assemblies into a single file	Internal
DEBUG	Execute the CPW Debugger program	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
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
FILETYPE	Change filetype to type specified	Internal
FOR	Exec-file command	Internal
HELP	Provide on-screen help for commands, or list all available commands	Internal
IF	Exec-file command	Internal
INIT	Initialize a disk	External
LINK	Link an object module	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
MAKELIB	Generate a library file from an object module	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 CPW	Internal
RENAME	Change a filename	Internal
RUN	Same as ASMLG or CMPLG	Internal
SET	Set shell variables	Internal
SHOW	Show languages, system default language, prefixes, time, volumes on line	Internal
SWITCH	Change the positions of two files in a directory	Internal

TEXT	Change default language to TEXT	Language
TYPE	Type a file to standard output	Internal
UNSET	Delete a shell variable	Internal

See Chapter 8 for instructions on adding CPW 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 shell is not case sensitive; that is, you can enter commands in any combination of uppercase and lowercase letters.

italics Italics indicate a variable, such as a filename or address.

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

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

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 pathename, in which the current prefix is assumed. For example, if a file is named FILE in the subdirectory DIRECTORY on the volume VOLUME, then the *pathname* parameter would be: /VOLUME/DIRECTORY/FILE. If the current prefix were /VOLUME/, then you could use DIRECTORY/FILE for *pathname* . 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 unit names .CONSOLE and .PRINTER can be used as filenames; the device numbers .D1, .D2, ...D*n* can be used for volume names.

| 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.

... Elipses 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 CPW Shell command interpreter:

- There is no command-line prompt; whenever the cursor appears in the left margin, 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 "Executing 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 previous section).
- All commands and parameters (except for segment names) can be entered in any combination of uppercase and lowercase characters.
- Segment names must be entered as all uppercase for source languages that are not case sensitive; for case-sensitive source languages, segment names must be entered exactly as they appear in the source code.
- Where 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.
- If you fail to enter a required parameter, you are prompted for it, as described in Chapter 2.
- Any of these commands can be placed in an Exec command file for automatic execution; Exec files are described in the section "Exec Files" in this chapter.

ALINK

`ALINK [+L|-L] [+S|-S]` *sourcefile* `[KEEP=`*outfile*`]`

This internal command calls the CPW Linker to process a file of LinkEd commands. `ALINK` is a synonym for `ASSEMBLE`; you can use the `ASSEMBLE` or `COMPILE` commands instead of `ALINK` if you prefer.

> **Note:** 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.

`+L|-L` If you specify `+L`, the linker generates a listing of the LinkEd source code; and a listing (called a **link map**) of the segments in the object module, 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 module (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.

sourcefile The full pathname or partial pathname (including the filename) of a LinkEd source file.

KEEP=*outfile* You can use this parameter to specify the pathname or partial pathname of the load file.

This parameter has the same effect as placing a KEEP command in your LinkEd source file. If you have a KEEP command in the LinkEd file and you also use the KEEP parameter, then the KEEP command in the LinkEd file takes precedence.

Important: Keep the following points in mind regarding the KEEP parameter:

- If you use neither the KEEP parameter nor the KEEP command, then no load file is produced.
- If a file named *outfile* already exists, it is overwritten without a warning when this command is executed.

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" 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, or CMPLG. The linker is described in detail in Chapter 7.

ASM65816

ASM65816

This language command sets the shell default language to CPW 65816 Assembly Language.

ASML

ASML [+L | -L] [+S | -S] *sourcefile* [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2* [...]])] [*language1*=(*option* ...)
[*language2*=(*option* ...) ...]]

This internal command assembles (or compiles) and links a source file. The CPW Shell checks the language of the source file and calls the appropriate assembler or compiler. If the maximum error level returned by the 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), then the resulting object module is linked. The linker is described in Chapter 7. Assembler error levels are described in Appendix B.

Notes: You can use `APPEND` directives (or the equivalent) to tie together source files written in different computer languages; CPW compilers and assemblers check the language type of each file and return control to the shell when a different language must be called. See the section "Assembling or Compiling a Program" in Chapter 2 for a description of the assembly and compilation process.

Not all compilers or assemblers make use of all the parameters provided by this command (and the `ASSEMBLE`, `ASMLG`, `COMPILE`, `CMPL`, `CMPLG`, and `RUN` commands, which use the same parameters). The CPW 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 used `APPEND` directives to tie together source files in more than one language, then all parameters are passed to every compiler, and 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.

In general, command-line parameters (those described here) override source-code options when there is a conflict.

`+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 module; the assembler or compiler may also produce a symbol table. The CPW 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.

sourcefile The full pathname or partial pathname (including the filename) of the source file.

KEEP=*outfile* You can use this parameter to specify the pathname or partial pathname (including the filename) of the output file. For a one-segment program, CPW names the object module *outfile*.ROOT. If the program contains more than one segment, CPW 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), then 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, then the load file is named *outfile*.

This parameter has the same effect as placing a KEEP directive in your source file. If you have a KEEP directive in the source file and you also use the KEEP parameter, two object modules are produced with the extension .ROOT; one corresponding to the parameter and one to the directive. However, the pathname in the KEEP directive takes precedence; other files with .A or other extensions are created only with the filename used in the directive, and the linker uses only the pathname given in the KEEP directive.

Important: Keep the following points in mind regarding the KEEP parameter:

- If you use neither the KEEP parameter nor the KEEP directive, then the object modules are not saved at all. In this case, the link cannot be performed, because there is no object module to link.
- The filename you specify in *outfile* must not be over 10 characters long. This is because the extension .ROOT is appended to the name, and ProDOS 16 does not allow filenames longer than 15 characters.
- If a file named *outfile* already exists, it is overwritten without a warning when this command is executed.

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. Separate the segment names with one or more spaces. The CPW Linker automatically selects the latest version of each segment when the program is linked.

In CPW Assembly language, you assign names to segments with START or DATA directives. In most high-level languages, each subroutine becomes an object segment; the segment name is the same as the subroutine name. The object file created when you use the NAMES parameter contains only the specified segments. 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. Therefore, you must use the same output filename for every partial compilation or assembly of a program.

For example, if you specify the output filename as OUTFILE for the original assembly of a program, then the assembler creates object modules 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.

Note: No spaces are permitted immediately before or after the equal sign in this parameter.

See the section "Partial Assemblies or Compiles" in this chapter for more information on partial assemblies.

language1=(*option* ...) ... This parameter allows you to pass parameters directly to specific CPW compilers or assemblers. For each compiler or assembler for which you want to specify options, type the name of the language (exactly as defined in the command table), 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; the CPW Shell does no error checking on this string, but passes it through to the compiler or assembler. You can include option strings in the command line for as many languages as you wish; if that language compiler is not called, then the string is ignored.

Note: No spaces are permitted immediately before or after the equal sign in this parameter.

Listings and error messages are sent to the screen unless you 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" 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 assembles and links a source file named `MYFILE`, and writes the load file to disk as the file `MYPROG`. A symbol table is produced and no source listing is produced regardless of whether 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`; if `MYFILE` or a file appended to MYFILE is a C program, then the C-compiler option that specifies a prefix for Include files is passed to the C compiler. After the program is assembled or compiled, it is linked and the load file is written to disk as the file `MYPROG`.

```
ASML MYFILE KEEP=MYPROG C=(-I/CPW/CINCLUDES/)
```

Note: The `ASML`, `ASMLG`, `CMPL`, and `CMPLG` commands first assemble or compile the source file (or files), then send the object file specified in the `KEEP` parameter (or in a `KEEP` directive in the source file) to the linker as its only input. These commands cannot be used to send several object files with different root filenames to the linker. To link two or more object files, use the `LINK` command.

Important: If you are using a LinkEd file to take advantage of the advanced link capabilities it provides, do *not* use the ASML command. Instead, use either the ASSEMBLE or COMPILE command to assemble or compile your program. You can process the LinkEd file automatically by appending it to the end of your program with an APPEND directive (or the equivalent), or you can process it independently with the ALINK command. The linker is described in detail in Chapter 7.

ASMLG

ASMLG [+L|-L] [+S|-S]*sourcefile* [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2* [...]])] [*language1*=(*option* ...)
[*language2*=(*option* ...) ...]]

This internal command assembles (or compiles), links, and runs a source file. Its function is identical to that of the ASML command, except that once the file has been successfully linked, it is executed automatically. See the ASML command for a description of the parameters.

ASSEMBLE

ASSEMBLE [+L|-L] [+S|-S]*sourcefile* [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2* [...]])] [*language1*=(*option* ...)
[*language2*=(*option* ...) ...]]

This internal command assembles (or compiles) a source file. Its function is identical to that of the ASML command, except that the ASSEMBLE command does not call the linker to link the object modules it creates; therefore, no load module 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 description of the parameters.

C

C

This language command sets the shell default language to CPW C.

CATALOG

CATALOG [*prefix*]
CATALOG [*pathname*]

This internal command lists to the screen the directory of the volume or subdirectory you specify.

prefix The pathname or partial pathname of the volume, directory, or subdirectory for which you want a directory listing. If the prefix is omitted, then the contents of the current directory are listed.

pathname The full pathname or partial pathname (including the filename) of the file for which you want directory information. You can use wildcards 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 `/CPW/UTILITIES/`, use the following command:

```
CATALOG /CPW/UTILITIES
```

To get directory information about the `MAKELIB` file in the `UTILITIES/` subdirectory when the current prefix is `/CPW/`, use the following command:

```
CATALOG UTILITIES/MAKELIB
```

To list every file beginning with a *M* in the `UTILITIES/` subdirectory, use the following command:

```
CATALOG /CPW/UTILITIES/M=
```

Or, if `/CPW/UTILITIES/` were the current directory, you could use the following command to achieve the same result:

```
CATALOG M=
```

You can alphabetize ProDOS 16 directories with the `COMPRESS` command, and change the positions of files in a directory with the `SWITCH` command. See the section "Listing the Directory" in Chapter 2 for a description of the fields in the directory listing.

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 CPW, each source or text file is assigned the current default language type when it is created. When you assemble or compile the file, CPW 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 previous section, "Command Types and the Command Table" includes a discussion of language types and language commands.

You can use the CHANGE command to correct the CPW 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 CPW language type to an ASCII text file (ProDOS 16 filetype $04) created with another editor.

CMPL

CMPL [+L | -L] [+S | -S] *sourcefile* [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2* [...]])] [*language1*=(*option* ...)
[*language2*=(*option* ...) ...]]

This internal command compiles (or assembles) and links a source file. Its function and parameters are identical to those of the ASML command. See your compiler manual for the language-specific options available.

CMPLG

CMPLG [+L | -L] [+S | -S] *sourcefile* [KEEP=*outfile*]
[NAMES=(*seg1* [, *seg2* [, ...]])] [*language1*=(*option* ...)
[*language2*=(*option* ...) ...]]

This internal command compiles (or assembles), links, and runs a source file. Its function is identical to that of the ASMLG command. See the ASML command for a description of the parameters. See your compiler manual for the language-specific options available.

COMMANDS

COMMANDS *pathname*

This internal command causes CPW 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 CPW, it reads the command-table file named SYSCMND in the system prefix. 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" in this chapter.

COMPILE

COMPILE [+L|-L] [+S|-S] *sourcefile* [KEEP=*outfile*]
[NAMES=(*seg1* [,*seg2* [,...]])] [*language1*=(*option* ...)
[*language2*=(*option* ...) ...]]

This internal command compiles (or assembles) a source file. Its function is identical to that of the ASML command, except that it does not call the linker to link the object modules it creates; therefore, no load module 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 description of the parameters. See your compiler manual for the language-specific options available.

COMPRESS

COMPRESS A|C|A C [*prefix*[/]]

This internal command compresses and alphabetizes directories.

A — Use this parameter to alphabetize the file names in a directory. The filenames appear in the new sequence whenever you use the CATALOG command.

C — Use this parameter to compress a directory. When you delete a file from a directory, a "hole" is left in the directory that ProDOS 16 fills with the file entry for the next file you create. Use the C parameter to remove these holes from a directory, so that the name of the next file you create is placed at the end of the directory listing instead of in a hole in the middle of the listing.

A C — You can use both the A and C parameters in one command; if you do so, you must separate them with one or more spaces.

prefix — The pathname or partial pathname of the directory you wish to compress or aphabetize, *not* including any filename. If you do not include a volume or directory path, then the current directory is acted on.

This command works only on ProDOS 16 directories, not on other file systems such as DOS or Pascal. To interchange the positions of two files in a directory, use the SWITCH command.

Note: When you "delete" a file, the file and its directory entry remain on the disk, but the directory entry and the blocks on the disk containing the file are marked by ProDOS 16 as being available; that is, they can be overwritten. Most programs that "recover" deleted files take advantage of this fact by reading the old directory information off the disk (assuming you haven't written any new information to the disk since deleting the file). The COMPRESS command removes this information from the directory, making it harder to recover deleted files.

COPY

COPY [-C] *pathname1* [*prefix2*/] [*filename2*]
COPY *prefix1 prefix2*
COPY *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 filename, then COPY does not prompt you if the target filename (*filename2*) already exists.

pathname1 The full or partial pathname (including the filename) of the file to be copied. Wildcards may be used in the filename.

prefix1 The pathname or partial pathname of a directory that you wish to copy; if you do not include a filename in the first parameter, then the directory (including all the files, subdirectories, and files in the subdirectories) is copied.

prefix2 The pathname or partial pathname of the directory you wish to copy the file or directory to. If you do not include the pathname of a volume or subdirectory, then the current directory is used.

filename2 The filename to be given to the copy of the file. Wildcards can *not* be used in this filename. If you leave this parameter out, then the new file has the same name as the file being copied.

volume1 The name of a volume that you want to copy onto another volume. If both parameters are volume names, then a block-by-block disk copy is performed. You can use a device number instead of a volume name.

volume2 The name of the volume that you want to copy onto. If both parameters are volume names, then a block-by-block disk copy is performed. You can use a device number instead of a volume name.

If you do not specify *filename2*, and a file with the filename specified in *pathname1* exists in the target subdirectory, or if you do specify *filename2* and a file named *filename2* exists in the target subdirectory, then 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. 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, then 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 CPW prompts you for any required parameters. However, since the target prefix and filename are not required parameters, 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 filename prompt.

If you use volume names for both the source and target, then 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. This command performs a block-by-block copy, so it makes an exact duplicate of the disk; both disks must be the same size for this command to work. You can use device numbers rather than volume names to perform a disk copy; device numbers are described in the section "Device Numbers and Names" in Chapter 2.

The following command makes a copy of the file `FILEA` on the system prefix, gives the copy the filename `FILEB`, and places it in the same prefix:

```
COPY FILEA FILEB
```

The following command copies the file `MYPROG` from the directory `CPW/` into the subdirectory `CPW/PROGRAMS/` without changing the name of `MYPROG`:

```
COPY /CPW/MYPROG /CPW/PROGRAMS/
```

The following command copies the subdirectory `/CPW/UTILITIES/HELP/` into the subdirectory `/HARDISK/DOCUMENTS/`, renaming the `HELP/` subdirectory `HELPFILES/`:

```
COPY /CPW/UTILITIES/HELP/ /HARDISK/DOCUMENTS/HELPFILES/
```

The following command copies the volume CPW onto the volume in disk drive `.D2`:

```
COPY /CPW/ .D2
```

CREATE

`CREATE` *prefix* `[/]`

This internal command creates a new subdirectory.

prefix The pathname or partial pathname of the subdirectory you wish to create.

CRUNCH

`CRUNCH` *rootname*

This internal command combines the object modules created by partial assemblies or compiles into a single object module. For example, if an assembly and subsequent partial assemblies have produced the object modules `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 modules 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 modules you wish to compress. For example, if your object modules are named FILE.ROOT, FILE.A, and FILE.B in subdirectory /HARDISK/MYFILES/, then use /HARDISK/MYFILES/FILE for *rootname*.

Note: CRUNCH requires a few blocks of workspace on the disk; if your disk is nearly full, delete some unneccessary files or copy your object modules to a clean disk before trying to use this command.

CRUNCH lists the type of each segment it finds; the segment types it recognizes are:

Static Code
Dynamic Code
Static Data
Dynamic Data

Use the DUMPOBJ command to obtain a listing of the segments in any object or load file. See the section "Partial Assemblies or Compiles" in this chapter for more information on partial assemblies. Segment types are discussed in the section "Cortland Concepts" in Chapter 1.

DEBUG

DEBUG

This external command calls the CPW Debugger.

The debugger is described in detail in Chapter 6.

DELETE

DELETE *pathname*

This internal command deletes the file you specify.

pathname The full pathname or partial pathname (including the filename) of the file to be deleted. Wildcards may be used in the filename.

DISABLE

DISABLE D|N|B|W|R *pathname*

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.

B "Backup required" flag. If you disable this attribute, the file will not be flagged as having been changed since the last time it was backed up.

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.

pathname The full pathname or partial pathname (including the filename) of the file whose attributes you wish to disable. You can use wildcard characters in the filename.

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

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 *ProDOS 16 Technical Reference Manual.* Directory listings are described in the section "Listing a Directory" in Chapter 2.

DUMPOBJ

`DUMPOBJ [`*option ...*`]` *pathame* `[NAMES=(`*seg1 seg2 ...*`)]`

This external command 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.

- `+X` Write the file dump in hexadecimal codes rather than OMF records. Segment headers are always printed in ASCII text unless you also select the `-H` option.
- `+D` Write the file dump as a 65816 disassembly rather than OMF records.
- `-H` If the output format is hexadecimal codes (`+X` option), then this option causes the headers to also be listed as hexadecimal codes. For all other output formats, the headers are not printed at all.
- `-O` Don't show the contents of the segments; list the headers only.

`-F` Suppress the checking of the filetype. 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.

`-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.

`-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.

`-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.

`-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.

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 CPW object module format. If you use the `-F` option, you can specify a file of any filetype.

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 `DUMPOBJ` with the `-O` and `-S` options. Segment-name searches are case sensitive (that is, *seg1, seg2,* ... must match the segment names exactly, including the case of the characters).

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 4.2. The fields in the segment header are described in the section "Object Module Format" in Chapter 9.

```
Block count     : $00000001             1
Reserved space : $00000000             0
Length          : $0000000F            15
Kind            : $00         static code
Label length    : $0A                  10
Number length   : $04                   4
Version         : $01                   1
Bank size       : $00010000         65536
Org             : $00000000             0
Alignment       : $00000000             0
Number sex      : $00                   0
Segment number : $0000                 0
Disp to names   : $002C                44
Disp to body    : $003B                59
Load name       :
Segment name    :  Second
```

Figure 4.2. 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 4.3. The first column shows the actual displacement into the segment, in bytes, of that record. The segment header takes up 59 ($3B) bytes, so the op code of the first record in the segment is at $3C, and the first record 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 9.

> **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. However, you can 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
000048 000000 | CONST   ($01) | A2
00004A 000001 | EXPR    ($EB) | 01 : MSG4MSG3-
000064 000002 | CONST   ($03) | A000B9
000068 000005 | BEXPR   ($ED) | 02 : MSG3
000076 000007 | CONST   ($01) | 20
000078 000008 | BEXPR   ($ED) | 02 : COUT
000086 00000A | CONST   ($05) | C8CAD0F660
00008C 00000F | END     ($00)
```

Figure 4.3. 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 4.4. The first column shows the actual displacement into the segment, in bytes, of the first byte in the line. The segment header takes up 59 ($3B) bytes, so the first record in the segment starts at $3C. 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. CPW Assembly Language is described in the *Cortland Programmer's Workshop: Assembly Language 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 with an immediate operand, 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 CPW Assembler whether to use long or short operands for transfer instructions.) LONGA and LONGI directives are also placed at the beginning of every segment 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, then the disassembly might lose track of the setting of the status register.

```
00003C 000000 |           LONGA  ON
00003C 000000 |           LONGI  OFF
00003C 000000 | SECOND    START
00003C 000000 |           USING  DATA
000047 000000 |           LDX    #(MSG4-MSG3)
00004F 000003 |           LDY    #$00
000052 000006 |           LDA    MSG3,Y
000057 000006 |           JSR    COUT
00005D 000009 |           INY
00005F 00000A |           DEX
000060 00000B |           BNE    *+$F6
000062 00000D |           RTS
000063 00000E |           END
```

Figure 4.4. 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 4.5. The first column shows the actual displacement into the segment, in bytes, of the first byte in the line. The segment header takes up 59 ($3B) bytes, so the first byte in the segment body is at $3C. The next four columns show the next 16 bytes in the file. The last column shows the ASCII equivalents of those bytes.

*****complete this figure with up-to-date hex dump*****

```
00003C  |  E4444154 41202020 20202001 A2EB0183  |  dDATA        "k
00004C  |  4D534734 20202020 2020834D 53473320  |  MSG4         MSG3
00005C  |                                       |
00006C  |                                       |
00007C  |                                       |
00008C  |                                       |
00009C  |                                       |
0000AC  |                                       |
0000BC  |                                       |
0000CC  |                                       |
0000DC  |                                       |
0000EC  |                                       |
0000FC  |                                       |
00010C  |                                       |
00011C  |                                       |
00012C  |                                       |
00013C  |                                       |
00014C  |                                       |
00015C  |                                       |
00016C  |                                       |
00017C  |                                       |
00018C  |                                       |
00019C  |                                       |
0001AC  |                                       |
0001BC  |                                       |
0001CC  |                                       |
0001DC  |                                       |
0001EC  |                                       |
0001FC  |                                       |
```

Figure 4.5. 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 DUMBOBJ attempts to scan the file for segments as if it were in OMF.

DUMPOBJ is extremely useful for debugging compilers and assemblers, but 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 a CPW 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 7 for a discussion of LinkEd commands.

EDIT

`EDIT` *pathname*

This external command calls the CPW 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 created and opened. If you use a wildcard character in the filename, the first file matched is opened.

The CPW default filetype changes to match the filetype of the open file. If you open a new file, that file is assigned the current default filetype. Use the `CHANGE` command to change the filetype of an existing file. To change the CPW default filetype before opening a new file, type the name of the language you wish to use, and press RETURN.

The editor is described in Chapter 5.

ENABLE

`ENABLE D|N|B|W|R` *pathname*

This internal command enables one or more of the access attributes of a ProDOS 16 file, as described in the discussion of the `DISABLE` command. 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. The parameters are the same as those of the `DISABLE` command.

EXEC

`EXEC`

This language command sets the shell default language to the `EXEC` command language. When you type the name of a file that has the `EXEC` filetype, the shell executes each line of the file as a shell command. Exec command files are described in the section "Exec Files" in this chapter.

EXECUTE

`EXECUTE` *pathname* [*paramlist*]

This internal command executes an Exec file. If this command is executed from the CPW Shell command line, then 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. *****can this filename include wildcards?*****

paramlist The list of parameters being sent to the Exec file.

Normally, variables defined within an Exec file or passed into that Exec file as parameters are local to that file. If you use the `EXECUTE` command, however, then variables defined in or passed to an Exec file are valid in the Exec file that called that file. If you use an `EXECUTE` command on the shell command line to execute an Exec file, then the variables defined in that Exec file are global; they are valid in any Exec file. See the section "Exec Files" in this chapter for a more complete discussion of this command.

FILETYPE

`FILETYPE` *pathname filetype*

This internal command changes the ProDOS 16 filetype of a file.

pathname The full pathname or partial pathname (including the filename) of the file whose filetype you wish to change.

filetype The ProDOS 16 filetype 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 filetype used in disk directores; for example, `S16`, `OBJ`, `EXE`. A partial list of ProDOS 16 filetypes is shown in Table 4.4. See the *Cortland ProDOS 16 Reference* for a complete list of filetypes.*****Is it in there?*****

You can change the filetype of any file with the `FILETYPE` command; CPW does not check to make sure that the format of the file is appropriate. However, the ProDOS 16 call used by the `FILETYPE` command may disable some of the access attributes of the file. Use the `CATALOG` command to check the filetype and access-attribute settings of the file; use the `ENABLE` command to reenable any attributes that are disabled by ProDOS 16.

Table 4.4. ProDOS Filetypes *****please check, correct, and add as necessary*****

Decimal	Hex	Abbreviation	Filetype

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 system load
180	$B4	RTL	Run-time library
181	$B5	EXE	Shell load
182	$B6	STR	Startup load
127	$FF	SYS	ProDOS 8 system load

HELP

`HELP` [*commandname*]

This internal command provides on-line help for all the commands in the command table provided with the CPW Development Environment. If you omit *commandname*, then a list of all the commands in the command table are listed on the screen.

commandname The name of the CPW Shell command about which you want information.

When you specify *commandname*, the shell looks for a text file with the specified name in the subdirectory `CPW/UTILITIES/HELP/`. 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, and 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.

INIT

`INIT` *device* `[`*name*`]`

This external command formats a disk as a ProDOS 16 volume.

device The device number of the disk drive containing the disk to be formatted; or, if the disk being formatted already has a volume name, you can specify the volume name instead of a device number.

name The new volume name for the disk. If you do not specify *name*, then the name `BLANK` is used.

CPW 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.

Warning: `INIT` destroys any files on the disk being formatted.

LINK

LINK [+L|-L] [+S|-S] *objectfile* [KEEP=*outfile*]
LINK [+L|-L] [+S|-S] (*objectfile1 objectfile2* ...) [KEEP=*outfile*]

This internal command calls the CPW Linker to link object modules to create a load file. You can use this command to link object modules created by CPW assemblers or compilers . The linker is described in Chapter 7.

+L|-L If you specify +L, the linker generates a listing (called a link map) of the segments in the object module, 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 module (called a symbol table). If you specify -S, the symbol table is not produced.

objectfile The full or partial pathname, minus filename extension, of the object files to be linked. All files to be linked must have the same filename (except for extensions), and must be in the same subdirectory. For example, the program TEST might consist of object files named TEST.ROOT, TEST.A, and TEST.B, all located in directory /CPW/MYPROG/. In this case, you would use /CPW/MYPROG/TEST for *objectfile*.

(*objectfile1 objectfile2,...*) You can link several object files into one load file with a single LINK command. Enclose in parentheses the full pathnames or partial pathnames, minus filename extensions, of all the object files to be included; separate the filenames with spaces. The first file named, *objectfile1*, 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 /CPW/MYPROG/. In this case, you would use (/CPW/MYPROG/TEST1 /CPW/MYPROG/TEST2) for *objectfile*.

KEEP=*outfile* Use this parameter to specify the pathname or partial pathname of the executable load file.

Important: If you do not use the KEEP parameter, then 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 CPW prompts you for any required parameters. However, since the output pathname is not a required parameter, you are *not* prompted for it. Consequently, the link is performed, but the load file is not saved. To save the results of a link, you *must* include the KEEP parameter in the command line (or following the pathname you enter in response to the Filename prompt).

As an example of the first form of the LINK command, suppose you want to link the object file /CPW/TEST1, consisting of files TEST1.ROOT, TEST1.A, and TEST1.B. The following command creates the load file /CPW/MYTEST; no the link map or symbol table are produced:

```
LINK /CPW/TEST1 KEEP=/CPW/MYTEST
```

As an example of the second form of the LINK command, suppose you want to link the object file /CPW/MYPROG/TEST1 consisting of files TEST1.ROOT, TEST1.A, and TEST1.B, and the object file /CPW/MYPROG/TEST2 consisting of files TEST1.A and TEST1.B, combining them into a single load file. The following command creates the load file /CPW/MYTEST, printing the link map but suppressing the symbol table:

```
LINK +L -S (/CPW/TEST1,/CPW/TEST2) KEEP=/CPW/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.

> **Note:** The ASML, ASMLG, CMPL, and CMPLG commands call the linker with the object file specified in the KEEP parameter (or in a KEEP directive in the source file) as its only input; these commands cannot be used to send several object files with different root filenames to the linker. To link two or more object files, use the LINK command.

If you need to take advantage of the advanced link capabilites provided by the CPW 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 7.

> **Important:** The LINK command can be used only to process object 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 CPW 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 7.

MACGEN

MACGEN [+C|-C] *infile outfile macrofile1* [*macrofile2 ...*]

This external command creates a custom macro file for a CPW 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 specify +C (the default value), then all excess spaces are removed from the macro file to save space. If you use the GEN ON directive (to include expanded macros in your sourcefile listing), or the TRACE ON directive (to include conditional execution directives in your sourcefile listing), then use the -C parameter in MACGEN to improve the readability of the listing.

infile The full pathname or partial pathname (including the filename) of the CPW 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. Wildcards 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. MACGEN 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. It then opens a temporary file called SYSMAC on the work prefix, scans *macrofile1* for referenced macros, and writes the macros it finds to SYSMAC. If there are still unresolved references to macros, MACGEN 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, then 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), SYSMAC is copied to *outfile*.

The following example scans the file /CPW/MYPROG for macro names, searches the macro libraries /LIB/MACROS and /LIB/MATHMACS for the referenced macros, and creates the macro file /CPW/MYMACROS:

```
MACGEN /CPW/MYPROG /CPW/MYMACROS /LIB/MACROS /LIB/MATHMACS
```

Since the macros are written to a temporary file instead of directly to *outfile*, 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 MY.MACROS, 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.

MAKELIB

`MAKELIB [-F] [-D]` *libfile* `[+|-|^`*objectfile1* `+|-|^`*objectfile2* `. . .]`

This external command creates 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 produced.
- *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 as many object files to add as you wish. 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 as many component files to remove as you wish. 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 as many files to be written out as object files as you wish. Separate filenames with spaces.

A CPW library file (ProDOS 16 filetype $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. Library files can be searched faster than object files by the CPW Linker. It is easier for the linker to resolve references between object files when those files are incorporated into the same library file.

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; it 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 CPW 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.

To create an OMF library file using the CPW Assembler, use the following procedure:

1. Write one or more source files in which each library subroutine is a separate segment.
2. Assemble the programs, specifying a unique name for each program with the `KEEP` parameter in the `ASSEMBLE` command. Each multi-segment program is saved as two object files, one with the extension `.ROOT`, and one with the extension `.A`.

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

4. Place the new library file in the LIBRARIES/ subdirectory. (You can accomplish this in step 3 by specifying /CPW/LIBRARIES/LIBFILE for the library file, or you can use the MOVE command after the file is created.)

CPW OMF library files and library-dictionary segments are described in the section "Object Module Format" in Chapter 9. The CPW Linker is described in Chapter 7.

MOVE

MOVE [-C] *pathname1* [*prefix*/] [*filename2*]

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 (*filename2*) already exists.

pathname1 — The full pathname or partial pathname (including the filename) of the file to be moved. Wildcards may be used in this filename.

prefix — The pathname or partial pathname of the directory you wish to move the file to. If you do not include a volume or the path of a subdirectory, then the current directory is used. Wildcards can *not* be used in this pathname. If the subdirectory of *filename2* is the same as that of *pathname1*, then the file is only renamed.

filename2 — If you specify *filename2*, the file is renamed when it is moved. If you leave this parameter out, then the file is not renamed.

If *pathname1* and the target directory are on the same volume, then CPW calls ProDOS 16 to move the directory entry (and rename the file, if *filename2* is specified). If the source and destination are on different volumes, then the file is copied; if the copy is successful, then the original file is deleted. If *filename2* already exists and you complete the move operation, then the old file named *filename2* is deleted and replaced by the file that was moved.

PREFIX

PREFIX [*n*] *prefix*[/]

This internal command sets any of the eight standard ProDOS 16 prefixes to a new subdirectory.

n — A number from 0 to 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.

prefix The pathname or partial pathname of the subdirectory to be assigned to prefix *n*.

Prefix 0 is the ProDOS 16 system prefix (also called 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 5 are used for specific purposes by CPW; see the section "Standard Prefixes" in this chapter for details. The default settings for the prefixes are shown in Table 4.1. Use the `SHOW PREFIX`*n* command to find out what the prefixes are currently set to.

The prefix assignments are reset to the defaults each time CPW is booted. To use a custom set of prefix assignments every time you start CPW, 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" in this chapter for instructions on writing an Exec file.)

PRODOS

`PRODOS`

This language command sets the CPW Shell default language to ProDOS 16 text. ProDOS 16 text files are standard-ASCII files with ProDOS 16 filetype $04; these files are recognized by ProDOS 16 as text files. CPW TEXT files, on the other hand, are standard-ASCII files with ProDOS 16 filetype $B0 and a CPW language type of `TEXT`. The CPW language type is not used by ProDOS 16. See the *Cortland ProDOS 16 Reference* for a discussion of ProDOS 16 filetypes.

QUIT

`QUIT`

This internal command terminates the CPW program and returns control to ProDOS. If you called CPW from another program, ProDOS 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 wildcards 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 wildcards in the filename.

If you specify a different subdirectory for *pathname2* than for *pathname1* , then 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 [+L|-L] [+S|-S]sourcefile [KEEP=outfile]
     [NAMES=(seg1 [,seg2 [,...]])] [language1=(option ...)
     [language2=(option ...) ...]]
```

This internal command compiles (or assembles), links, and runs a source file. Its function is identical to that of the ASMLG command. See the ASML command for a description of the parameters. See your compiler or assembler manual for the default values of the parameters and the language-specific options available.

SHOW

SHOW LANGUAGE| LANGUAGES| PREFIX *n*| 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 language table, including their language numbers.

PREFIX *n* Shows the current subdirectory to which PREFIX n is set, where *n* is a number from 0 to 7. If you omit *n*, then PREFIX 0 is shown. Note that there is a space between PREFIX and *n*. See the section "Standard Prefixes" in this chapter for a discussion of CPW prefixes.

TIME Shows the current time.

UNITS Shows the available units, including device numbers and volume names.

More than one parameter can be entered on the command line; to do so, separate the parameters by one or more spaces.

SWITCH

SWITCH *pathname1 pathname2*

This internal command interchanges two filenames in a directory.

pathname1 The full pathname or partial pathname (including the filename) of the first filename to be moved. If you use wildcards in the filename, the first filename matched is used.

pathname2 The full pathname or partial pathname (including the filename) to be switched with *pathname1* . The prefix in *pathname2* must be the same as the prefix in *pathname2*.You cannot use wildcards in this filename.

For example, suppose the directory listing for /CPW/MYPROGS/ is as follows:

```
/CPW/MYPROGS/=

Name            Type   Blocks   Modified             Created              Access   Subtype

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
```

To reverse the positions in the directory of the last two files, use the following command:

```
SWITCH /CPW/MYPROGS/COMMAND.FILE /CPW/MYPROGS/ABS.OBJECT
```

Now if you list the direcatory again, it looks like this:

```
/CPW/MYPROGS/=

Name            Type   Blocks   Modified             Created              Access   Subtype

C.SOURCE        SRC    5        26 MAR 86 07:43      29 FEB 86 12:34      DNBWR    C
ABS.OBJECT      OBJ    8        12 NOV 86 15:02       4 MAR 86 14:17       NBWR
COMMAND.FILE    SRC    1         9 APR 86 19:22      31 MAR 86 04 22      DNBWR    EXE
```

You can alphabetize ProDOS 16 directories with the COMPRESS command, and list directories with the CATLOG command. This command works only on ProDOS 16 directories, not on other file systems such as DOS or Pascal.

TEXT

TEXT

This language command sets the CPW Shell default language to CPW TEXT. CPW text files are standard-ASCII files with ProDOS 16 filetype $B0 and a CPW language type of TEXT. The CPW language type is not used by ProDOS 16.

Use the PRODOS command to set the language type to ProDOS 16 text; that is, standard-ASCII files with ProDOS 16 filetype $04. PRODOS text files are recognized by ProDOS 16 as text files. See the *Cortland ProDOS 16 Technical Reference* manual for a discussion of ProDOS 16 filetypes.

TYPE

TYPE [+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 — This option causes CPW to precede each line with a line number.

pathnamen — The full pathname or partial pathname (including the filename) of the file to be printed. You can use wildcards in this filename, in which case every text or source file matching the wildcard filename specification is printed. You can specify more than one pathname in the command; separate pathnames with spaces.

startlinen — The line number of the first line of this file to be printed. If this parameter is omitted, then the entire file is printed.

endlinen — The line number of the last line of this file to be printed. If this parameter is omitted, then the file is printed from *startline* to the end of the file.

CPW text files, ProDOS 16 text files, and CPW source files can be printed with the `TYPE` command. To redirect output to a printer or file, use output redirection as described in the section "Redirecting Input and Output" in this chapter.

Exec Files

You can execute one or more CPW Shell commands from a command file. To create a command file, set the system language to `EXEC` by typing `EXEC` Return, and 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 anticipate command prompts and include responses to them. For example, you could use the following Exec file, called `NEWCOPY`, to make a copy named `NEWTEST` of the file `TEST` on the current directory:

```
COPY
TEST
N
NEWTEST
```

When you execute this Exec file by typing `NEWCOPY` and pressing RETURN, the following sequence appears on the screen:

```
NEWCOPY

COPY
File Name: TEST

File Exists. Replace it? N
New File Name: NEWTEST
```

CPW then copies the file `TEST` to the filename `NEWTEST` on the current directory.

If you execute an interactive utility, such as the CPW 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, then you are returned to the Exec file when you quit the utility.

Exec files are programmable; that is, CPW 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 files; to do so, separate the commands with semicolons (`;`). For example, the Exec file shown above could be written as follows:

```
COPY;TEST;N;NEWTEST
```

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 `EXEC` pathname 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 single or double quotes. Quote 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`
- `tom's cat`

In this case, you would enter the following command on the command line:

```
FARM cow chicken 'one egg' 'tom''s cat'
```

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, the logic operators used to form Boolean (logical) expressions, and the `EXEC` command language.

Variables

Any alphanumeric string up to eight characters long can be used as a variable name in an Exec file. (If you use more than eight characters, only the first eight are significant.) All

variable values and parameters are ASCII strings of 256 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 pass them into the Exec file as parameters, or include them in a `FOR` command or a `SET` command as described in the section "Conditional-Execution Commands." To assign a null value to a variable (a string of zero length), use the `UNSET` command. Variable names are always enclosed in curly brackets (`{ }`), except when being defined in the `SET`, `UNSET` and `FOR` commands.

Variables can be defined within an Exec file, or on a shell command line before the Exec file is executed, by using the `SET` command. 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.

> **Note:** Several of these variables may have the values `0` or `1`; keep in mind that these values are literal ASCII strings. A null value (a string of zero length) is considered undefined; that is, neither `0` nor `1`.

The following variable names are reserved:

`{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 (the ASCII character) `0`, or null (undefined), then string comparisons are not case sensitive. If you set this variable to any other value, then string comparisons are case sensitive. The default value is null.

`{Command}` — The last command executed; if the command was a filename (as for a CPW utility), then `Command` is a full *****??or partial??***** pathname. *****are command parameters included?*****

`{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).

`{Exit}` — If you set this variable to a non-null value, and if any command or nested Exec file returns a non-zero error status, then execution of the Exec file is terminated. The default value for `Exit` is (the ASCII character) `1`. Use the `UNSET` command to set `Exit` to a null value (that is, to delete its definition).

`{FileType}` — A hexadecimal number (represented as an ASCII string) corresponding to a load filetype. If `FileType` is undefined or set to a nonvalid filetype, then $B5 (shell load file) is used. The most common alternative is $B3 (system load file). Valid load filetypes are $B3–$BE.

`{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 the case that your printer automatically inserts a carriage return–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 caret (^) to indicate a control character. A space is interpreted as a space character, $20.

> **Caution:** 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
```

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, then 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`, then no form-feed characters are sent.

`{PrinterSlot}` The number of the slot containing your printer-driver PC board; an ASCII number from `0–7`. The default value for `PrinterSlot` is `1`.

`{Status}` The error status returned by the last command or Exec file executed. This variable is the ASCII character `0` ($30) if the command completed successfully. For most commands, if an error occurred, the error value returned by the command is the ASCII string `255`.

Logic Operators

CPW includes two operators that you can use to form Boolean (logical) expressions. String comparisons are case sensitive if `{CaseSensitive}` is (the ASCII character) `1` (or any value other than `0` or null). If an expression result is true, then the expression

returns the character `1`. If an expression result is not true, then the expression returns the character `0`. There must be one or more spaces before and after the comparison operator.

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`.

Comments

To enter a comment into an Exec file, start the line with the number-sign character (#). For example, the following Exec file sends a catalog listing to the printer:

```
CATALOG >.PRINTER
#Send a catalog listing to the printer
```

Use a semicolon followed by the number-sign character to put a comment on the same line as a command:

```
CATALOG >.PRINTER  ;#Send a catalog listing to the printer
```

Exec-File Commands

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 CPW Shell is not case sensitive; that is, you can enter commands in any combination of uppercase and lowercase letters.
italics	Italics indicate a variable, such as a filename or address.
[ ]	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.

•
•
•

Vertical elipses indicate that any number of shell commands can be inserted between the two commands shown.

Break

`BREAK`

This command terminates the innermost `FOR`, `LOOP`, or `IF` statement currently executing. For example, if an `IF` statement is executing inside a `FOR` loop and a `BREAK` statement is encountered, then control passes to the statement following the next `END` statement. A `BREAK` statement must be used to terminate a `LOOP` loop.

Continue

`CONTINUE`

This command causes control to skip over following statements to the next `END` statement. It does not cause termination of the loop (unless the last value of *value* has been used).

Echo

`ECHO` *string*

This 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 non-space 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.

Execute

`EXECUTE` *pathname* [*paramlist*]

This command executes an Exec file, treating the variables defined in the file as if they were defined in the Exec file that contains the `EXECUTE` command. If this command is executed from the CPW Shell command line, then 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. *****can this filename include wildcards?*****

paramlist The list of parameters being sent to the Exec file.

Normally, variables defined within an Exec file or passed into that Exec file as parameters are local to that file. If you use the `EXECUTE` command, however, then variables defined in or passed to an Exec file are valid in the Exec file that called that file. If you use an `EXECUTE` command on the shell command line to execute an Exec file, then the variables

defined in that Exec file are global; they are valid in any Exec file. 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 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
```

Note: When the CPW Shell finds an Exec file named LOGIN in the CPW system prefix during system load, the shell treats the variables defined in LOGIN as if they were typed from a shell command line. To reexecute LOGIN without reloading CPW (to reset system parameters to your selected defaults, for example), use the command EXECUTE LOGIN.*****Is that true?*****

Exit

EXIT [*number*]

This 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, then the predefined variable {status}==*number* . This parameter is useful only for nested Exec files, and allows you to terminate an Exec file if an Exec file it called terminated with an error.

Export

EXPORT [*variable* ...]

This command makes the listed variables available to Exec files called by the current exec file. *****Can more than one variable be listed, as in MPW, or not?*****

variable The names of variables you wish to make available to enclosed Exec files. Variable names are not case sensitive, and only the first eight characters are significant. If you omit *variable*, then a list of all exported variables (for the current Exec file) is written to standard output.

Variables included in an EXPORT statement in a shell command line can be used within any Exec file. Variables included in an EXPORT statement in an Exec file can be used in any Exec file called *by* that file (and are passed on to any Exec files enclosed at lower levels);

they do *not* affect the values of variables in an Exec file that *called* that file. Variables defined within an Exec file and not exported are local to that file. Variables exported and redefined within an enclosed Exec file revert to their original values when the enclosed Exec file is terminated.

Variables included in an `EXPORT` statement in the `LOGIN` file are exported to the shell command level.

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*}==*value1*; the second time through the loop {*variable*}==*value2*; and so forth. The values of *value* must be separated by one or more spaces.

If IN is omitted, then 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 statements between the FOR statement and the END statement 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 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—Else If—Else—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` statement are executed. All other statements between the `IF` and `END` statements are skipped. If none of the expressions evaluate as true, and if an `ELSE` statement is included, then the statements between the `ELSE` and the `END` statement are executed.

expression Any expression formed with one of the logical operators discussed in the previous section.

Loop—End

```
LOOP
   •
   •
   •
END
```

This command sequence defines a loop that repeats continuously until a `BREAK` command is encountered.

Set

`SET [`*variable*` [`*value*`]]`

This 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 eight characters are significant. If you omit *variable*, then 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 256 characters. All characters, including spaces, starting with the first non-space character after *variable* to the end of the line, are included in *value*. Ifyou include *variable*but omit *value*, then 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 EXPORT command to make a variable available to an enclosed Exec file. Variables defined within an Exec file and not exported are local to that file. Use the EXECUTE command to make variables defined in an Exec file available to the file (or the shell command level) that called that file. Use the UNSET command to delete the definition of a variable.

Important: Certain variable names are reserved; see the subsection "Variables" in this section for a list of reserved variable names.

Unset

UNSET *variable*

This 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 eight 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. Variables exported and then deleted within an enclosed Exec file revert to their original values when the enclosed Exec file is terminated. See the EXPORT command for a discussion of exporting variables.

Example

When the following Exec file is executed, it attempts to assemble and link a source file; if the operation is unsuccessful, then it attempts to assemble and link a different source file; if neither program can be assembled and linked, then 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.
END                         ;#End of IF statement
IF {status}==0              ;#If second prog was successful
TEST2                       ;#run the prog and
EXIT                        ;#quit.
ELSE                        ;#If both progs failed
```

```
ECHO {message}              ;#send message.
END                         ;#End of second IF statement.
```

Note: Equal signs (=) can have three different functions in Exec files: 1) As a wildcard character in a filename; 2) As part of a CPW command parameter (for example, `KEEP=TEST`); 3) In the string-comparison operators == and != .

LOGIN Files

Each time you start CPW, it looks for an Exec file named `LOGIN` on the `CPW/SYSTEM/` prefix. If it finds such a file, CPW 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 CPW 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. To reexecute `LOGIN` without reloading CPW (to reset system parameters to your selected defaults, for example), use the command `EXECUTE LOGIN`.*****Is that true?*****

SNIDE COMMENTS

Chapter 5

Editor

The CPW Editor allows you to write and edit source and text files for use with CPW 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: in this chapter, all keyboard-based 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 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 third section, "Commands," describes each editor command and gives the key or key combination assigned to the command. The fourth section, "Setting Editor Defaults," describes how to set the defaults for editor modes and tab settings for each language.

Modes

The behavior of the CPW Editor depends on the settings of several modes, as follows:

- Insert
- Escape
- Auto Indent
- Text Selection
- Word 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" in this chapter; for example, to learn how to toggle wrap mode, look up "Toggle Wrap Mode."

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. The overstrike-mode cursor is a blinking block alternating with the character it is on. In insert mode, the cursor becomes a vertical bar that is located between two character positions; any characters you type are inserted at the cursor location, and any characters to the right of the cursor are moved to the right.

The maximum length of a line in the CPW Editor is 255 characters (including spaces). (The editor displays only the first 80 characters of the line.) If you continue to insert characters in a line after the line is 255 characters long, the new characters overstrike the

old characters in the line. If the editor is both in insert and automatic-wrap modes, then when the characters being inserted reach the end-of-line marker (usually at column 80, see the section "Setting Editor Defaults" 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" in this chapter for an example.

Escape

When you press the |ESC| key, the editor enters escape mode. Escape mode has several special features:

- Every letter key executes a command. If no other command is defined for a key, pressing the key terminates escape mode and returns you to text-entry mode. You cannot enter text while in 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 |ESC| and before you execute the command. For example, if the editor is programmed so that the sequence |ESC| `Y` deletes a line of text, then to delete 10 lines of text (starting with the line the cursor is on), type |ESC| `10 Y`. 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. For example, if you type |ESC| `50 E` to move the cursor up 50 lines when it is only 10 lines from the top of the file, it moves up 10 lines and stops.
- In escape mode, the cursor is a plus sign (+) alternating with the character the cursor is on.
- To exit escape mode, press |ESC|, or press any letter not defined as an escape-mode command.

Auto Indent

You can set the editor so that |RETURN| moves the cursor to the first column of the next line, or so that it follows indentations already set in the text. If the editor is set to put the cursor on column 1 when you press |RETURN|, then changing this mode causes the editor to put the cursor on the first nonspace character in the next line; if the line is blank, then the cursor is placed under the first nonspace character in the first nonblank line above the cursor.

Select Text

The Cut, Copy, and Delete commands require that you first select a block of text. The CPW 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 commands are active:

- bottom of screen

- top of screen
- cursor down
- cursor up
- start of line
- screen moves

In addition, while in character-oriented select mode, the following cursor-movement commands are active:

- cursor left
- cursor right
- 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.

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 automatically inserts a carriage return, moves the cursor to the next line down, and continues entering text when you reach the end of the line. You can set the CPW Editor to either mode of operation.

In non-wrap mode, when you reach the end-of-line mark (usually at column 80—see the section "Setting Editor Defaults" in this chapter), any additional characters you type overwrite the last characater 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 80. In non-wrap mode, the additional characters overwrite the last character on the line, and the line ends with `pneumas`; in automatic-wrap mode, the entire word `pneumatolysis` is moved to the beginning of the next line.

> **Note:** The CPW 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 continuation line does *not* move back up to maintain the length of the first line. To remove the carriage return you must enter insert mode, then move the cursor to the beginning of the second line and execute a Delete Character Left command.

If the editor is both in insert and automatic-wrap modes, then when the characters being inserted reach the end of the line (usually column 80, see the section "Setting Editor

Defaults" 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. For example, suppose you begin inserting characters in the following line (the last 20 characters of the line are shown, with column numbers included for clarity). (The editor displays only the first 80 characters on the line; additional columns are shown in the following illustrations for clarity.)

```
0000000001111111111222222222233333333334444444444555555555566666666667777777777888
1234567890123456789012345678901234567890123456789012345678901234567890123456789012

tated 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:

```
22222222223333333333444444444455555555556666666666777777777788888888889999999999
01234567890123456789012345678901234567890123456789012345678901234567890123456789

inerals in this specimen appear to have formed as a result of pneumatolysis.
```

The *f* in *of* (the last character of the newly-inserted text) has reached column 80, and the line wraps as follows:

```
0000000001111111111222222222233333333334444444444555555555566666666667777777777888
1234567890123456789012345678901234567890123456789012345678901234567890123456789012

ated earlier, the minerals in this specimen appear to have formed as a result
of pneumatolysis.
```

The effect of inserting characters is the same whether you type them in or paste them in. The Insert Space command, on the other hand, can extend a line past the end-of-line marker.

Macros

You can define up to 16 *****I'm assuming all keypad keys, not just numbers, can be used***** macros for the CPW Editor; a macro allows you to substitute a single keystroke for up to 128 predefined keystrokes. The macro can contain text, editor commands, and (by including shell |ENTER|) commands.

To create a macro, press ⌘-|ESC|. The current macro definitions appear on the screen. To replace a definition, press the key on the numeric keypad that corresponds to that macro, then type in the new macro definition. Press |OPTION|-|ESC| to terminate the macro definition. You can include |CTRL|-*key* combinations, ⌘-*key* combinations, |OPTION|–*key* combinations, and the |RETURN|, |ENTER|, |ESC|, and arrow keys. The following conventions are used to display keystrokes in macros:

|CTRL|–*key*	The uppercase character *key* is shown in inverse.
⌘–*key*	An open apple followed by *key* (for example, ⌘K)
|OPTION|–*key*	The extended-ASCII character corresponding to that key.
|OPTION|–*keypad key*	A closed apple followed by *keypad key* (for example, 8)

ESC	Control-[; an inverse left bracket.
RETURN	Control-M; an inverse M.
ENTER	Control-J; an inverse J.
↑	An inverse K. up arrow (↑??)
↓	An inverse J. down arrow (↓??)
←	An inverse H. left arrow (←??)
→	An inverse U. right arrow (→??)
DELETE	A block (■).

Note: Each ♂-*key* combination or OPTION-*key* combination counts as two keystrokes in a macro definition.

When you are finished entering macros, press OPTION-ESC to terminate the last option definition, then press OPTION to end macro entry. The following prompt appears on the screen:

```
Write macros to disk?
```

Type Y and press RETURN to save the new macro definitions on disk. Type N and press RETURN to return to the file without saving the macros. Macros are saved on disk in the file SYSEMAC in the CPW system subdirectory (usually /CPW/SYSTEM/; see the section "Standard Prefixes" in Chapter 4).

To execute a macro, hold down OPTION and press the keypad key corresponding to that macro. For example, assume you assign the following keystroke sequences to the *1*, *2*, and *3* keys on the keypad:

```
1: [B[♂, -------------------------------------------

2: [B[♂, |                                          |

3: ♠1♠2♠2♠2♠2♠1♂,↓→
```

When you press OPTION-1 (using the *1* on the keypad), the editor enters escape mode, inserts a blank line (above the line the cursor was on), exits escape mode, moves the cursor to the beginning of the line, and inserts a space followed by a string of hyphens.

When you press OPTION-2, the editor inserts a blank line, moves the cursor to the beginning of the line, then inserts a vertical bar, a row of spaces, and another vertical bar.

When you press OPTION-3, the following sequence occurs:

1. The editor calls macro 1, which inserts a blank line in the file, moving the line the cursor was on and all subsequent lines down to make room, then puts a space in column 1 followed by a string of hyphens

2. The editor calls macro 2 four times in a row. Each time macro 2 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 1 again, which inserts another blank line at the top of the four lines just written, 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 5.1).

The output of macro 3 looks like this:

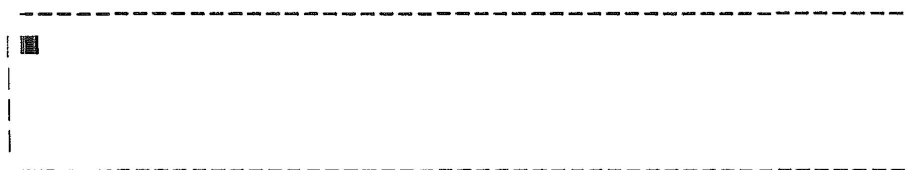


Figure 5.1. Output of an Editor Macro

The cursor is at the upper left corner of the box that the macro drew on the screen.

Command Descriptions

This section describes the functions that can be performed with editor commands. The key assignments for each command are shown with the command description. See Appendix A for a summary of all the default command-key assignments.

> **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

|CTRL|-G

The ASCII control character BEL ($07) is sent to the output device. Normally, this causes the speaker to beep.

Bottom of Screen

|CTRL|-B

The cursor moves to the first column in the last line on the screen.

Bottom of Screen / Page Down

|CTRL|-⌘-J
⌘-↓

The cursor moves to the last visible line on the screen, preserving the cursor's horizontal 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, then the screen scrolls down 22 lines).

Change

See Search and Replace.

Clear

See Delete.

Copy

|CTRL|-C
⌘-C

When you execute the Copy command, the editor enters select mode, as discussed in the section "Select Text" in this chapter. 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. (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

|CTRL|-J
↓

The cursor is moved down one line, preserving its horizontal position. If it is on the last line of the screen, the screen scrolls down one line.

Cursor Left

|CTRL|-H
←

The cursor is moved left one column. If it is in column one, the command is ignored.

Cursor Right

|CTRL|-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

|CTRL|-K
↑

The cursor is moved up one line, preserving its horizontal 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

|CTRL|-X
Ⴚ-X

When you execute the Cut command, the editor enters select mode, as discussed in the section "Select Text" in this chapter. 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. (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

Ⴚ-|ESC|

The editor enters the macro-definition mode. Press |OPTION|-|ESC| to terminate a definition, and |OPTION| to terminate macro-definition mode. The macro-definition process is described in the section "Macros" in this chapter.

Delete

Ⴚ-|DELETE|

When you execute the Delete command, the editor enters select mode, as discussed in the section "Select Text" in this chapter. 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 and put in the Undo buffer (see the description of the Undo command). (To cancel the Delete operation without deleting the block from the file, press |ESC| instead of RETURN.)

Delete Character

|CTRL|-F
Ó-F
|ESC| G

The character that the cursor is on is deleted and put in the Undo buffer (see the description of the Undo 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|
|CTRL|-D

The cursor is moved left one column, and a Delete Character command is executed. If the cursor is in column one and the overstrike mode is active, no action is taken. If the cursor is in column one and the insert mode is active, then 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

|ESC| Y

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 command).

Delete to EOL

|CTRL|-Y
Ó-Y

The character that the cursor is on, and all those 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 command).

Delete Word

|ESC| |DELETE|

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, then for as long as the cursor is on a space. This command thus deletes the word plus all punctuation and spaces up to the beginning of the next word. If the cursor is on a space, 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 command).

End of Line

Ć-.
Ć->

If the last column on the line is not blank, the cursor moves to the last column. If the last column is blank, then 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.

Find

See Search.

Help

Ć-?
Ć-/

The contents of the SYSHELP file in the system prefix appear on the screen. Press |RETURN| or |ESC| to return to the editor. Any other key is ignored.

Insert Line

|ESC| B

A blank line is inserted at the cursor position, and the lines the cursor was on and below are scrolled down to make room. The cursor remains in the same position on the screen.

Insert Space

|ESC| H

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 the Insert Space command can extend a line past the end-of-line marker.

Paste

|CTRL|-V
Ć-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. If the editor is in character-oriented select mode, the material is copied at the cursor column.

Warning: If enough characters are inserted to make the line longer than 255 characters, the excess characters are lost.

Quit

|CTRL|-Q
Ꮛ-Q

Exit to the editor Quit menu, which gives you the following options:

- R Return control to the editor. You are returned to the same position in the file you were at when you quit it, in the same editing mode.
- S Save the file to the filename used when the editor was entered and return to the editor menu.
- N Save the file to a new filename. You are prompted for a new filename, and the file is saved to that filename. You are returned to the editor menu.
- 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 want to quit it before the new file is loaded. When the file you specify is loaded, the editor places the cursor on the first character in the file, set to edit mode (as opposed to escape mode), and set to the default parameters in the SYSTABS file that correspond to the language of the file.
- 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.

If you press |RETURN| without entering any other data in response to a prompt, the command is aborted and control returns to the menu.

Remove Blanks

|CTRL|-R
Ꮛ-R

If the cursor is on a blank line, that line and all subsequent blank lines up to the next non blank 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 a *repeat count* (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 "Modes" in this chapter.

Return

|RETURN|
|CTRL|-M

The |RETURN| key works in one of two ways, depending on the setting of the auto-indent mode toggle: 1) to move the cursor to column one of the next line; or 2) to place the cursor on the first nonspace character in the next line, or, if the line is blank, beneath the first nonspace character in the first nonblank line on the screen above the cursor. 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 8 approximately equal sections. The screen-move commands move the file to a boundary between one of these sections. The command Ó-1 jumps to the first character in the file, and Ó-9 jumps to the last character in the file. The other seven Ó-*n* commands cause screen jumps to evenly spaced intermediate points in the file.

Scroll Down One Line

|ESC| C

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

|ESC| X

The screen scrolls down one screen's height (for example, if the screen is 22 lines high, then the screen scrolls down 22 lines). Scrolling can continue past the last line in the file.

Scroll Up One Line

|ESC| E

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

|ESC| W

The screen scrolls up one screen's height (for example, if the screen is 22 lines high, then the screen scrolls up 22 lines). If the top line on the screen is less than one screen's height from the beginning of the file, the screen scrolls to the beginning of the file.

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 include all occurrences of the string, whether it is imbedded 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

The following editing commands are active when you are entering the search string:

←	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
-Z or Control-Z	undo delete
-E or Control E	toggle insert mode
Esc	exit without saving changes
Return	search for the string

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 include all occurrences of the string, whether it is imbedded 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. The prompt `Auto or Manual (A M Q)?` appears.

`A` 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.

`M` 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` Return to replace the string and search for the next occurrence; `N` Return to leave this occurrence of the string unchanged and search for the next occurrence; or `Q` Return 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` Terminate the search and replace operation and return to the file you are editing.

The following editing commands are active when you are entering text in the `Find What` and `Replace With` boxes.

←	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
-Z or Control-Z	undo delete
-E or Control E	toggle insert mode
Esc	exit without saving changes
Return	exit and save changes

When you enter a replacement string and type A Return or M Return, 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. If the string is not found, the following message appears on the screen:

```
String Not Found
```

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; to search between the current cursor location and the end of the file, use the Search and Replace Down command.

Set and Clear Tabs

|CTRL|--I

If there is a tab stop in the same column as the cursor, it is cleared; if there is no tab stop in the cursor column, one is set.

Start of Line

-,
-<

The cursor is placed in column one of the line it is in.

Tab

|TAB|
|CTRL|-I

The cursor is moved to the next tab stop. If there are no more tab stops, then the cursor is moved to the end of the line. If the editor is in overstrike mode, the tab acts only as a cursor-control command; no characters are inserted between the last character on the line and the tab stop. If you type a character at the tab stop, however, space characters are inserted in the file between the last character and the new character. 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.

Tab Left

CTRL-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

CTRL-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; if set to the first nonspace character, it is changed to put the cursor on column 1. Auto-indent mode is described in the section "Modes" 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. When you are in escape mode, pressing any character not specifically assigned to an escape-mode command returns you to edit mode. Escape and edit modes are described in the section "Modes" in this chapter.

Toggle Insert Mode

CTRL-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 "Modes" in this chapter.

Toggle Select Mode

CTRL-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 use individual characters instead; if it is set to character-oriented selects, it is toggled to use whole lines. See the section "Modes" in this chapter for more information on select mode.

Toggle Wrap Mode

CTRL--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 "Modes" in this chapter.

Top of Screen

CTRL-T

The cursor moves to the first column in the first line on the screen.

Top of Screen / Page Up

CTRL--K
-↑

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, then 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.

Undo Delete

CTRL-Z
-Z

The last character or block of characters deleted using the Delete, Delete Character, Delete Character Left, Delete Line, Delete to End of Line, 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.

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 woulld appear that an appppeal to reason would not go
unAanswered.
```

When you execute the Undo command one time, the text deleted last is restored; in this case, an `a`. If you execute a second Undo command, the text deleted before that, `pp`, is restored, and so on. In this example, four Undo commands in a row would put the following text on the screen:

```
Apple
```

A maximum of 1024 characters can be stored in the undo buffer. No warning is issued if you delete more than 1024 characters.

Word Left

⌘-←
CTRL-⌘-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 it is blank, to the last word in the first nonblank line preceding the cursor.

Word Right

⌘-→
CTRL-⌘-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.

Setting Editor Defaults

When you start the CPW Editor, it reads the file named `SYSTABS` (located in the CPW system prefix), which contains the default settings for tab stops, return mode, and select mode. The `SYSTABS` file is an ASCII text file that you can edit with the CPW Editor.

Each language recognized by CPW is assigned a language number. The `SYSTABS` file has three lines associated with each language:

1. The language number.
2. The default settings for return, select, and word-wrapping modes.
3. The default tab and end-of-line-mark settings.

CPW languages are discussed in the section "Command Types and the Command Table" in Chapter 4; a list of CPW languages and language numbers is given in Appendix A.

The first line of each set of lines in the `SYSTABS` file specifies the language that the next two lines apply to. CPW 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. If the first column contains a zero, pressing RETURN in the editor causes the cursor to go to column one in the next line; if it's a one, pressing RETURN sends the cursor to the first nonspace character in the next line (or, if the line is blank, beneath the first nonspace character in the first nonblank line on the screen above the cursor).
2. If the second character is zero, the editor is set to line-oriented selects; if one, it is set to character-oriented selects.
3. If the third character is zero, the cursor stops when it reaches the end of a line; if one, the editor inserts a carriage return and wraps to the next line.
4. The fourth character is reserved for future enhancements.
5. The fifth character is reserved for future enhancements.

The third line of each set of lines in the `SYSTABS` file sets default tab stops. There are 80 zeros and ones in the first line of each pair, 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 CPW Assembly Language and CPW C:

```
003
00000
00000000010000010000000000000000000000001000000100000001000000010000000100000002
010
11100
00000000100000001000000010000000100000001000000010000000100000001000000010000002
```

If no defaults are specified for a language (that is, there are no lines in the `SYSTABS` file for that language), then the editor assumes the following defaults:

- RETURN sends the cursor to column one.
- Character-oriented selects.
- 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.

Note that you can change tabs and editing modes while in the editor.

Chapter 7

Linker

This chapter describes the CPW Linker, including its input, output, options, and commands.

A linker is a program that locates individual machine-language program segments, resolves references between segments, and combines them into a complete, executable program. The CPW Linker is independent of source-code language. It is capable of extracting specific code segments from libraries and programs on multiple disks, and can create segmented load files.

The linker works with any assembler or compiler that generates files conforming to the Cortland object module format (OMF). The linker can join separate files produced by Cortland-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.

The CPW Linker can be executed from a command line with a limited number of options, or the full power of the linker can be exploited by compiling a file of linker commands. CPW Linker command files, called LinkEd files, are described at the end of this chapter. 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
- controling the diagnostic output of the linker

Most users will never need the options provided by LinkEd; the first several sections of this chapter describe the linker functions available through a command-line command only.

The principal tasks of a linker are to assemble the segments needed for a program and to resolve global references. Because most Cortland code is relocatable, however, the CPW Linker must work together with the System Loader to resolve 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 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 sequence in which the linker searches for object files
- The printed output from the linker

Object Files: Input to the linker

Object files are the output from an assembler or compiler, and are the input to a linker. Object files conform to the Cortland object module format, and 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 9 for more detailed information on the Cortland object module format.

Object files (ProDOS 16 filetype $B1) contain data or program code that has been translated (assembled or compiled) into machine language, but which 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. (The filetype of the load files the linker creates is set by the CPW CPW Shell `FileType` variable; the default filetype is $B5, shell load files. To change the filetype of a load file, use the shell `FILETYPE` command; to change the default filetype, use the shell `SET` command.)

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 Chapter 1 and fully described in Chapter 9.

Library Files

Library files (ProDOS 16 filetype $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 CPW library prefix (normally `/CPW/LIBRARIES/`); when you call the linker with a command-line command (as opposed to a LinkEd file), it first links the source code, then automatically searches all files in the library prefix to resolve references.

Library files differ from object files in that library-file segments are not aligned to 512-byte boundaries, and 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 4). Each library file can be created from any number of object files.

The order of subroutines within a library file is not important. If you use more than one library file, however, you must be sure that a subroutine in one library file does not reference a subroutine in another library file that comes before it in the directory.

Partial Assemblies and Filename Conventions

When you assemble or compile a program, you can use a `KEEP` directive (or equivalent) 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`, then the first file is named `MYPROG.ROOT` and the second one is named `MYPROG.A`.

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

1. If the compile involves more than one language, then the first compiler or assembler usually creates the `.ROOT` and `.A` files, the second compiler creates the `.B` file, and so on.
2. If you include a `NAMES` parameter on the command line, then a partial assembly or compile is performed. In this case, only the segments named are compiled, and are placed in a file with the next available alphabetic extension. Partial assemblies are described in the section "Partial Assemblies or Compiles" in Chapter 4.

Note: You can use the `CRUNCH` command described in Chapter 4 to combine all the alphabetic-extension files into one `.A` file.

For a link controlled from the command line, the linker selects the object files to process as follows (for a link controlled from a LinkEd file, segments are processed in the order specified by the LinkEd commands).

1. The linker first scans the output disk for a file name 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 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.

Important: Once a library file has been searched, it is not returned to by the CPW 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 use the `MAKELIB` program to combine as many object files into a single library file as you choose, however, and there are no restrictions on segments referencing each other within a single library 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 type $B5 unless you have set the shell `FileType` variable to another value), ready for loading by the System Loader. Load files are described in the following section.

Load Files: Output From the linker

Load files (types $B3–$BE) are the result of the processing of object files by the linker (and, optionally, the shell `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 Cortland 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 CPW 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. CPW C provides the `overlay` function to allow you to assign subroutines to specific load segments. As a default, most CPW compilers assign one load-segment name (a string of zeros) to all code segments, and another (`~GLOBAL`) to all global variables.

When you call the linker by using a CPW 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 4 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 9 and the description of the System Loader in the *Cortland Prodos 16 Reference* manual.

Local and Global Symbols: The linker recognizes two types of symbols: global and local. Global symbols are universally available, meaning that any segment may access them. Global symbols include code or data segment names (code segments are distinct from data segments in Cortland object module format), and (for CPW Assembly-Language programs) symbols defined in an `ENTRY` or `GEQU` directive. Local symbols, in this context, are labels that are defined only within individual code or data segments.

Local labels (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 linker never puts local symbols in the symbol table, with the exception of local labels in a data segment named in a `USING` directive.

The CPW Linker maintains a single symbol table for the entire link session. Two global symbols (or local symbols in data segments) with the same name cannot appear anywhere in the program.

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 this section.

Error Messages

Error messages list the type of error, the name of the segment, and where in the segment the error occurred. Pass two writes errors in the same way as pass one. Appendix B gives a full listing of error messages and their meanings. Error messages are generated during both pass one and pass two. Error messages cannot be suppressed.

Link Map and Source Listing

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 (static code, static data, dynamic code, or dynamic data), and the name of the segment. If the program is relocatable, the starting address is listed assuming the program starts at $000000. To suppress the link map, use the `-L` command-line parameter (or the LinkEd `LIST OFF` command).

If you call the linker from a LinkEd file, the LinkEd source code is written to standard output. To suppress the source listing, use the –L command-line parameter or the LinkEd `SOURCE OFF` command. A sample LinkEd output listing is shown in Figure 7.1.

Symbol Table

At the conclusion of pass two, an alphabetized global-symbol table is printed. The table presents the following information for each symbol:

symbol name
assigned value (hexadecimal)
classification number

The classification number is a pair of hexadecimal digits. If it is $00, the symbol is a global label or subroutine name; if it is nonzero, it is a data label and the value of the digit is the number of the data segment that defined it.

To suppress the symbol table, use the `-S` command-line parameter (or the LinkEd `SYMBOL OFF` command). A sample symbol table is shown in Figure 7.1.

Ending

When it finishes, the linker prints a message giving the number of errors detected (if any) and the highest error level encountered (see Appendix B). The last line tells where the program starts (if it is absolute code), and how many bytes long it is (in hexadecimal).

```
CPW Linker 4.1

   1 KEEP LINKTEST
   2 LINK/ALL /CPW/TEST
   3 LIBRARY *

0 errors found in source file


00002000 00000012 Static Code: MAIN
00002012 0000001B Static Data: DATA
0000202D 0000000F Static Code: SECOND
0000203C 00000003 Static Code: COUT


Global symbol table:

COUT      0000203C 00   DATA     00002012 01 MAIN         00002000 00
MSG1      00002012 01   MSG2     00002020 01 MSG3         00002020 01
MSG4      0000202D 01   SECOND   0000202D 00

Program starts at $00002000 and is $0000003F bytes long.
```

Figure 7.1. Sample Output of a LinkEd Command File

Linking From a Command Line

You can call the CPW Linker by executing a CPW Shell command; the following commands call the linker without having to execute a LinkEd command file:

- ASML

- ASMLG
- CMPL
- CMPLG
- LINK
- RUN

The LINK command lets you specify more than one object file to be linked into a single load file. The other commands call the linker only after a successful assembly or compile has been completed. Any of these commands let you suppress the link map and symbol table; however, for all but the LINK command, you can suppress the link map only if you also suppress the source listing of the assembler or compiler. The LINK command lets you specify a name for the load file; 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.

The following linker defaults are used when you execute one of these CPW 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 CPW Assembly Language, you can specify the load segment name as the operand of a START or DATA directive. Most CPW compilers use a string of zeros for the load-segment name of all code segments, and put all global label definitions and data in segments with the load-segment name ~GLOBAL.
- Object segments are scanned in the sequence 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 library files in the library prefix (normally /CPW/LIBRARIES/) are searched for unresolved references; no other library files 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 in the command line, or the KEEP directive is used in the source file (the source file KEEP directive has no effect on the LINK command).

If you need to have more control over the link, use a LinkEd file, as described in the following section. All of the CPW Shell commands are described in the section "Command Descriptions" in Chapter 4. The filetype of load files produced by the CPW Linker is set by the FileType shell variable; the default is ProDOS 16 filetype $B5. You can use the shell FILETYPE command to change the filetype of a load file, or the shell SET command to change the default filetype.

Linking With a LinkEd Command File

You can control every aspect of a link by using a LinkEd command file. LinkEd files are CPW source files with a language type of LINKED (see the section "Calling the Editor" in Chapter 2 for instructions on assigning a language type to a source file). To execute a LinkEd file, use one of the following CPW Shell commands (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):

- ALINK
- ASSEMBLE
- COMPILE

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 CPW Shell, which calls the CPW Linker. If you append the LinkEd file to the last file of the source code, then the file is linked automatically every time it is compiled or assembled. When the linker finishes processing the file, it tells the CPW 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 CPW Shell commands are described in the section "Command Descriptions" in Chapter 4.

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 (;). The following commands are recognized by the CPW Linker.

APPEND	append a LinkEd source file
COPY	copy a LinkEd source file
EJECT	skip to a new page if printer is on
KEEP	open a file for output
LIBRARY	search a library
LINK	link an object file
LIST	control subroutine listing
OBJ	set phantom program counter
OBJEND	turn off previous OBJ
ORG	set program counter
PRINTER	control printed output
SEGMENT	start load segment
SELECT	choose specific object segments
SOURCE	control LinkEd source program listing
SYMBOL	control symbol table output

Note: 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: 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.

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 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 entered exactly as shown. LinkEd commands are not case sensitive; that is, you can enter commands in any combination of uppercase and lowercase letters.
italics	Italics indicate a variable, such as a filename or address.
prefix	This parameter indicates any valid directory pathname or partial pathname. It does *not* include a filename. If the volume name is included, *prefix* must start with a slash (/); if *prefix* does not start with a slash, then the current prefix is assumed. For example, if you are copying a file to the subdirectory `SUBDIRECTORY` on the volume `VOLUME`, then the *prefix* parameter would be: `/VOLUME/SUBDIRECTORY/`. If the current prefix were `/VOLUME/`, then you could use `SUBDIRECTORY` for *pathname* . The device numbers `.D1, .D2, . . . .D`*n* can be used for volume names; if you use a device number, do not precede it with a slash. For example, if the volume VOLUME in the above example were in disk drive `.D1`, then you could enter the *prefix* parameter as `.D1/SUBDIRECTORY/`.
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 pathename, in which the current prefix is assumed. For example, if a file is named FILE in the subdirectory `DIRECTORY` on the volume `VOLUME`, then the *pathname* parameter would be: `/VOLUME/DIRECTORY/FILE`. If the current prefix were `/VOLUME/`, then you could use `DIRECTORY/FILE` for *pathname* . 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 unit names `.CONSOLE` and `.PRINTER` can be used as filenames; the device numbers `.D1, .D2, . . . .D`*n* can be used for volume names.
|	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 | <u>B</u>	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` *pathname*

LinkEd appends the LinkEd file with the pathname *pathname* to the present LinkEd source file. Any statements after the `APPEND` command in the present LinkEd file are ignored.

COPY

`COPY` *pathname*

LinkEd stops processing the present LinkEd file temporarily, and processes all statements in the LinkEd file specified by *pathname*. It then resumes processing the present file at the statement immediately following the `COPY` command.

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` *pathname*

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 `SEGMENT` command, below), each of which can be loaded independently and automatically during program execution.

The `KEEP` command opens the output file (load file) specified by *pathname*. All segments subsequently processed by LinkEd are placed in *pathname*, 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.

> **Caution:** 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).

The filetype of load files produced by the CPW Linker is set by the `FileType` shell variable; the default is ProDOS 16 filetype $B5. You can use the shell `FILETYPE` command to change the filetype of a load file, or the shell `SET` command to change the default filetype.

LIBRARY

`LIBRARY` *pathname*

A library file is a file of ProDOS 16 filetype $B2 containing object segments, such as general utilities, that may be called by other programs. The `LIBRARY` command causes the library file specified by *pathname* to be searched 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 4 for instructions on creating your own library files.

If an asterisk (`*`) is used for *pathname*, all the files in the current CPW library prefix (ProDOS prefix 4, normally `/CPW/LIBRARIES/`) are scanned.

LINK

`LINK[/ALL]` *pathname*

This command causes the object file specified by *pathname* to be included in the output file. All segments of the file specified by *pathname* not already included are added to the program. If the `LINK` command follows a `SEGMENT` command, then all the object segments in *pathname* are placed in the load segment defined by the `SEGMENT` command. If the `LINK` command does not follow a `SEGMENT` command, then all object segments are placed in the same load segment (named *****???*****). LinkEd ignores source-code load-segment names, such as those specified by the operand of a CPW Assembler `START` directive.

Use the `SELECT` command to link individual object segments from a given file.

If you use the `/ALL` qualifier, all files with the root filename specified by *pathname* 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 File Name Conventions" in this chapter). For example, suppose you use the following command:

```
LINK/ALL MYFILE
```

Then if files `MYFILE.A` and `MYFILE.B` are in the current directory, then the linker first searches `MYFILE.ROOT`, then `MYFILE.B`, and finally `MYFILE.A`.

If you do not include the `/ALL` qualifier, then you must specify the full pathname (including filename extension, if any).

LIST

`LIST ON|`<u>`OFF`</u>

The `LIST` command controls the output of the link map (the list of segment names). `LIST ON` causes all subsequent segment names to be sent to standard output; `LIST OFF` suppresses output (unless an error occurs). This command is overridden by the `L` option in the shell's `ASSEMBLE` and `COMPILE` command lines.

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*. Unlike `ORG`, `OBJ` has no effect on the actual physical location where the code is initially loaded; it 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*. However, it may 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 Cortand System Loader and memory manager instead; programs that do their own loading and memory management are very unlikely to work successfully with any other Cortland 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 *val*

`ORG` *val*

The `ORG` command sets the value of the program counter. Its operation 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 program-counter value at the start of the code segment, then 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.

> **Warning:** If the source code segment starts with an `ORG` directive, a LinkEd `ORG` command with a different value causes an error.

Val can be specified in either decimal (for example, `126720`) or hexadecimal (for example, `$01EF00`) format.

> **Note:** We strongly recommend that you not use this command, but take advantage of the capabilites of the Cortand System Loader and memory manager instead; programs that do their own loading and memory management are very unlikely to work successfully with any other Cortland routines. Since most code for Cortland is relocatable, the use of the `ORG` command should be restricted to specialized segments, such as graphics images.

PRINTER

`PRINTER ON|OFF`

The `PRINTER` command controls output to the printer. `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 CPW Shell's `ASSEMBLE`, `COMPILE`, or `ALINK` command line.

SEGMENT

`SEGMENT[/DYNAMIC] [/NUMBER=`*kind*`]` *segname*

The `SEGMENT` command defines the beginning of a new load segment in the current load file, and gives it the load-segment name *segname*. You can 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, then all object segments are placed in the same load segment. LinkEd ignores any load-segment assignments in your source code.

> **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. If the language you are using is not case sensitive, then segment names must be entered as all uppercase.

The linker automatically flags segments as static, so that they cannot be removed by the System Loader when additional memory space is needed. However, adding the `/DYNAMIC` qualifier to the `SEGMENT` command permits the loader to remove the segment from memory when necessary.

> **Note:** Dynamic segments are supported so that you can write programs that make highly efficient use of memory. However, keep in mind that any code that is needed at all times (or frequently) by the program cannot be dynamic. See the following note on load segments.

The Cortland 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 `/NUMBER=`*kind* qualifier to specify a special segment type for a

load segment. See the section "Object Module Format" in Chapter 9 for a description of segment types and the `KIND` field of the segment header.

> **Important:** You cannot use both the `/DYNAMIC` and `/NUMBER` qualifiers in the same `SEGMENT` command.

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 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 loader attempts to free memory by unloading dynamic segments not presently being used (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 *Cortland 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]` *pathname* `(`*seg1* `[,`*seg2* `[, . . . ]])`

This command causes the named segment(s) (*seg1*, *seg2*,...) from the file specified by *pathname* 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, then the object segments specified in *pathname* are placed in the load segment defined by the `SEGMENT` command. If the `SELECT` command does not follow a `SEGMENT` command, then all object segments are placed in the same load segment named ***???***.

Use the `LINK` command to link all the segments in a file.

If you include the `/SCAN` parameter, then all files with the root filename of the file specified by *pathname* 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 File Name Conventions" in this chapter). For example, suppose you use the following command:

```
SELECT/SCAN MYFILE
```

If files `MYFILE.ROOT`, `MYFILE.A`, and `MYFILE.B` are in the current directory, then the linker first searches `MYFILE.ROOT`, then `MYFILE.B`, and finally `MYFILE.A`.

If you do not use the `/SCAN` qualifier, then you must specify the full pathname (including filename extension, if any).

SOURCE

`SOURCE ON|OFF`

This command controls the output of LinkEd source code. `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 and link map. The symbol table is an alphabetical listing of all symbolic references (labels). The link map is a listing of each segment, with its starting address and length. All segments share the same symbol table. This command is overridden by the `S` option in the shell's `ASSEMBLE` and `COMPILE` command lines.

Examples

The listings below are all valid LinkEd files. Here all commands are written in upper case to follow the convention used in this book. Note that object-segment names should be entered as all uppercase except for languages (such as C) that are case sensitive, in which case segment names should be entered exactly as they are listed in the source code.

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 CPW 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 the load segment name ***what?***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 `MYFILE 2` and the system libraries are searched for needed subroutines:

```
KEEP MYPROG
LINK MYFILE
LIBRARY MYFILE2
LIBRARY *
```

Part III

Inside the Cortland Programmer's Workshop

Chapter 8

Adding a Program to CPW

This chapter describes how to add a utility program or compiler to the Cortland Programmer's Workshop. None of the information in this chapter is essential for writing programs that are independent of CPW.

Note that, when you add a utility or language to CPW, you should update the CPW command table to include it. CPW will execute a program that is not listed in the command table, but 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 4, and a list of language numbers currently assigned is given in Appendix A.

Compilers, Utilities, and Applications

ProDOS 16 supports two principal kinds of executable load files: ProDOS 16 filetype $B3, and filetype $B5. These two filetypes have the following characteristics:

- Programs of filetype $B3 take over complete control of the computer; they do not operate under a shell program. CPW itself is an example of such a program. When a program of filetype $B3 is called, the calling program executes a ProDOS 16 `QUIT` call, shutting itself down and clearing the screen. When the program finishes and executes a `QUIT` call, ProDOS 16 reboots the calling program.
- Programs of filetype $B5 run under a shell program; they do not remove the shell from memory. The shell calls a program of filetype $B5 in full native mode via a `JSL` instruction; when the program terminates, it returns control to the shell via an `RTL`, or via a ProDOS 16 QUIT call (that is intercepted by the shell).

CPW utility programs are programs of filetype $B5 designed to be run under the CPW Shell program. They perform operations too complex to be performed by the shell itself, but appear to the user to be shell commands. CPW compilers and assemblers are also programs of filetype $B5, and so are technically CPW utilities. Compilers and assemblers make use of special CPW Shell calls described in Chapter 10 to pass parameters between the shell and themselves. Since the requirements for compilers and assemblers are different from those for ordinary utility programs, they are discussed separately in this chapter.

You can write a self-contained program of filetype $B3 intended to be used with CPW; CPW executes any executable load file it finds on disk when you type in the program's pathname. Since CPW quits and ProDOS 16 clears the desktop when such a program is called, however, there are no special requirements for the program (other than those required by the Cortland system in general), and so these programs are not discussed in this chapter. See the *Programmer's Guide to the Cortland* for guidance in writing an event-driven program for the Cortland computer.

CPW Utilities

CPW utilities are applications designed to run under the CPW Shell. They must have ProDOS 16 filetype $B5. By following the guidelines described in this section, a utility can be executed from the CPW Shell with CPW remaining resident in memory.

When you enter a CPW command, the CPW Shell looks for the command name in the command table (see the section "Command Types and the Command Table" in Chapter 4). If the command is listed in the command table as a utility, then the shell loads it from the utility subdirectory (usually `/CPW/UTILITIES/`); if it is not in the command table, then the shell looks for it in the current subdirectory. In either case, the shell strips any I/O redirection information from the command line, and places the command line in a buffer in memory. It then places the address of command-line buffer in the X and Y registers, with the X register holding the most significant word of the address, and the Y register holding the least significant word. (All utility calls are made with the Cortland in full native mode, so both the X and Y registers are two bytes long.) The shell requests a user ID for the program from the User ID manager, and places it in the accumulator.

> **Important:** The command-line buffer is within the shell, and is subject to being overwritten, so it should be read immediately if there are parameters that may have been passed to the utility.

If the utility program does not have a direct-page/stack segment, then when the CPW 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.

Before the CPW Shell calls a utility program, the shell performs a checksum on itself; after the utility returns control to the shell, the shell performs the checksum again. If the two checksums do not match, the shell assumes it was corrupted by the utility and terminates execution with a system error.

Any utility must obey the following rules in order to execute successfully under the CPW Shell.

> **Warning:** If a program with ProDOS 16 filetype $B5 does not obey the following rules, you must quit CPW before calling it. Executing such a program from the CPW Shell can cause the system to crash.

- 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 CPW Shell identifier string `BYTEWRKS`. The utility should check this identifier to make sure that it has been launched by the CPW 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 a ProDOS `QUIT` call.
  2. A null-terminated ASCII string containing the input line for the utility. The CPW 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 4. You can use Cortland Text Toolset calls to read the next input character, to check to see if any more input is available, and to check to see if the input stream has been closed. Tool calls are described in the *Cortland Toolbox Reference* manual.

  Standard input can also be read as if it were a file by opening and reading a file named `.CONSOLE`. If input is coming from the keyboard, the shell echos the input characters at the current cursor location on the screen.

  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.

- 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 4. You can use Cortland Text Toolset calls to send output to standard output, or you can open and write to a file named `.CONSOLE`.
- The utility must handle its own errors. The preferred method is for the utility to open an attention box that reports the error, and take any additional action that is appropriate. Use the shell's `Attention` call, described in Chapter 10, to open the attention box. The utility should place an error-condition code in the accumulator before returning control. 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, then the Exec file terminates. Exec files are discussed in the section "Exec Files" in Chapter 4.
- The utility must use the Memory Manager to request memory; since several programs can be open on the desktop 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 CPW Shell calls described in Chapter 10 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.

  Important: If your utility uses CPW Shell calls, it will not run if called by ProDOS or another shell.

- If the utility launches another program, it must request a user ID from the User ID manager. Then utility is then 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 back to the CPW Shell.
- A utility should use the following procedure to quit:
  1. If the utility has requested any user ID's, it must release all memory buffers with those user ID's.
  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` or a ProDOS 16 `QUIT` call. The CPW Shell intercepts the `QUIT` call and releases all memory buffers associated with the utility.

When you add a utility program to CPW, you should provide a help file to go with it. Help files are ASCII text files (CPW language-type `PRODOS`) that have the same name as the command, and that are kept in the `/CPW/UTILTIES/HELP/` subdirectory. To see an example of a help file for a CPW utility, enter the following command:

```
HELP MAKELIB
```

Compilers and Assemblers

Compilers, assemblers, and interpreters are implemented in nearly identical ways in CPW. 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 Cortland text-file format, as specified in Chapter 9. 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. When you use standard output to send data to the printer, the printer driver converts form-feed characters to the appropriate amount of line feeds for printers that do not accept form-feed characters. Your compiler should handle tabs as discussed in Chapter 9.

All source-text lines in CPW are assumed to be 255 characters long.

Identifying the Language Type

Each language used by the Cortland Programmer's Workshop has a unique language number. Language numbers are discussed in the section "Command Types and the Command Table" in Chapter 4, and a list of the language numbers currently assigned is given in Appendix A. If you need a new language number for your compiler, contact the Apple Developer Technical Support department. 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 volume directory. The CPW 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 CPW Shell `CHANGE` command to assign the appropriate language type to the file. The format of directory entries is described in the *ProDOS 16 Reference* manual.

The language number must be the fourth and fifth bytes of your compiler. Knowledgable users may change this number; for example, to avoid confilicts between two versions of the same compiler. Your compiler should include a command that corresponds to the CPW Assembler `APPEND` directive; this command transfers control from the file being processed to a new file. When this command is used, your compiler must compare the language type of the new file with the fourth and fifth bytes of the compiler; if the language types do not

match, 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 10).

Entry and Exit

Compilers and assemblers that operate under CPW should have ProDOS 16 filetype $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 10) to read the input parameters. Upon completion, the compiler should execute a `Set_LInfo` call, and return control to the CPW Shell via an `RTL` or a ProDOS `QUIT` call. The system is in full native mode when it calls the compiler; it should be in full native mode when control is returned to the shell.

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 10; comments on some of them follow.

- If the compile completes with a non-fatal error, the compiler should return the error number in the `merrf` field of the `Set_LInfo` call. In this case the shell stops processing the program, even if `CMPL`, `CMPLG`, or an equivalent command was used. Use the following error levels for non-fatal errors:

  $02 Warning. Code may execute successfully

  $04 Error. The compiler may be able to correct this error. Examples may be 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. An example would be 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, and this type of syntax error should never occur. In free-format languages, such as Pascal, 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 a $FF in the `merrf` field of the `Set_LInfo` call, and place in the `org` field the displacement into the source file of the last line processed. When the CPW Shell receives a value over $7F for `merrf`, it calls the CPW Editor, which displays the source file; the line containing the error (as indicated by the displacement in the `org` field) is placed at the top 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.
- If a compile completes with a nonfatal error, 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, then the compiler should *not* clear the least significant bit of the `lops` field; this indicates to the shell that the compile is not complete, and it can then call the compiler appropriate to the new file.
- The `kflag` parameter 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 following section, "Ouput Files."
- If any segment names are listed in the buffer pointed to by the `parms` parameter, then a partial compile is to be performed. Partial compiles are discussed in detail in the section "Partial Compiles" in this chapter.
- 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.

Command Precedence

If your compiler includes source-file commands that control functions that can also be controlled from the command line, then 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 CPW must be capable of producing one or more object files that conform to CPW object module format (described in Chapter 9). These files are then processed by the CPW 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 CPW 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. Most CPW compilers assign one load-segment name (a string of zeros) to all code segments, and another (`~GLOBAL`) to all global variables. The CPW 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.

To maintain consistency between compilers, we recommend that your compiler assign all global variables to a load segment with the name `~GLOBAL`. The `~GLOBAL` load segment should obey the following conventions. See the section "Object Module Format" in Chapter 9 for a description of segments, segment types, and segment headers.

- It should contain only variables. Your compiler should place global names and symbols in this segment; it can place data segments, code segments, global variables, local variables or private variables in the ~GLOBAL load segment, as you see fit, but variables only.
- If your compiler has some restriction on the maximum size of the ~GLOBAL load segment, it should specify this fact by setting the code or data alignment factor in the segment header (that is, the BANKSIZE field).
- The load segment name should be ~GLOBAL (all uppercase); you can use any names you like for the object segments that go into this segment. The object segment name goes in the SEGNAME field of the segment header; the load segment name goes in the LOADNAME field of the segment header.

When the CMPL, CMPLG, or COMPILE command (or alias) is executed, the user can specify the name of the output file with the KEEP parameter. The compiler must check the directory for filenames that match the KEEP filename, excluding extensions, and set the kflag parameter in the Get_LInfo call accordingly. The shell places the KEEP 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:

0. 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, then 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.
1. 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 the COMPILE command included the parameter KEEP=MYFILE, and kflag=1, 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 should be put in a file named MYFILE.A.

   Note: The purpose of the .ROOT file is to hold the first code segment to be executed. In many languages, such as assembly language or BASIC, the first subroutine compiled is the first one to be executed. In some languages, such as C and Pascal, however, this is not the case. For such languages, the compiler can open the .A file immediately, and open the .ROOT file when the main program segment is compiled. Alteernatively, the compiler can use the .ROOT file to hold its standard initialization code (that is, the code that the compiler uses to initialize every program).
2. If kflag = 2, then 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.
3. If kflag = 3, then files with the KEEP filename and the extensions .ROOT and .A already exist. 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 use the next one. 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 multi-language compile (the first language creates the .ROOT and .A files, the second language the .B file, and so on) or by partial assemblies.

Note: The paradigm followed by the CPW 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 are in the subdirectory, but MYFILE.C is not, then the assembler never finds MYFILE.D. The next file created by the assembler, then, would be MYFILE.C. The user must be careful not to let such a case occur, because (in this example) the linker would start the link with the file MYFILE.D.

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), pad extra characters with space characters ($20).

Partial Compiles

The Cortland object module format, System Loader, and 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 Cortland Programmer's Workshop environment, it should be capable of creating segments that can be linked to segments output by other compilers, and of using segments created by other compilers. The use of segmented code provides two additional benefits: it facilitates the use of libraries, since the entire library file need not be linked to each program; and 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 are not 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, then 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. In testing the program, you have to make changes to segments SEG2 and SEG4, so you perform a partial compile; the compiler places segments SEG2 and SEG4 in the file MYPROG.B. 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, 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 4, to combine all of the alphabetic-extension files for a program into a single .A file. The CRUNCH command uses the same algorithm as the linker to scan the files for the latest version of each segment.

The following algorithm illustrates the partial-compile procedure:

```
while not defining_a_procedure do
   normal_compiler_functions;
compile_the_procedure_header;
   {does not produce code—only stuff for the symbol table}
if the_procedure_is_in_the_partial_compile_list then
   compile the procedure
else
   skip; {skips to the next procedure heading}
```

Suppose you have a simple C compiler that defines a new code segment for each function definition. This compiler has a function called next_token that returns a token. The skip procedure can be illustrated as follows (leftbracket and rightbracket refer to begin and end statements):

```
procedure skip;

count: integer;

begin
count := 0;
while (token <> leftbracket) and not end_of_file do next_token;
repeat
   if token = leftbracket then count := count+1
   else if token = rightbracket then count := count-1;
   next_token;
until (count = 0) or end_of_file;
if not end_of_file then next_token
end;
```

Help Files

When you add a new language to CPW, you should provide a help file to go with it. Help files are ASCII text files (CPW language-type `PRODOS`) that have the same name as the command, and that are kept in the `CPW/UTILTIES/HELP/` subdirectory. To see an example of a help file for a CPW language, enter the following command:

```
HELP ASM65816
```

If your language includes language-specific parameters for the `COMPILE`, `CMPL`, and `CMPLG` commands, then you should provide replacement `HELP` files for those commands (and their aliases) as well.

Interpreters

Installing an interpreter under CPW 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, since 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 9

File Formats

This chapter describes and defines the Cortland text-file format, which is used for standard ASCII text files and program source files by all CPW programs; and the object-module format, which is used for all CPW object files, library files, and load files. The Cortland System Loader also 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 Cortland, 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 Cortland it is required that a program be capable of reading and writing files in the standard text file format.

A Cortland 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

A Cortland 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; every character code is part of a line.
- The end of a text file is determined by the ProDOS 16 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.

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 a FF like a carriage return, or 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 Cortland 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—$9F and $FF carefully, because the low-ASCII equivalents of these codes ($00—$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—7E, 80—FF) plus CR, LF, FF, HT, and SP.

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.

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

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 Cortland uses a more general format that allows dynamic loading and unloading of file segments while a program is running, and supports the various needs of many languages and assemblers. The CPW Linker and System Loader fully support relocatable code; in general, you do not specify a load address for a Cortland program, but let the loader and Memory Manager determine where to load the program.

The Cortland object module format (OMF) supports language, CPW Linker, library, and System Loader requirements, and is extremely flexible, easy to generate, and fast to load.

This section defines four kinds of files: object files, library files, load files, and run-time library files.

- *Object files* are the output from an assembler or compiler, and are 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 the Cortland development environment, object files also support segmentation of code and partial assemblies and compiles. They support both absolute and relocatable program segments, which can be either static or dynamic, and they support position-independent (moveable) program segments.
- *Library files* contain general object segments that a linker can find and extract to resolve references unresolved in the source code. Only the code needed during the link process is extracted from the library file.
- *Load files* are the output of a linker and contain memory images that a loader loads into memory. Load files must be *very* fast to process. Cortland load files contain load segments that can be relocatable, moveable, 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.
- *Run-time library files* are load files containing general utilities that can be shared between applications. The utilities are contained in file segments that can be loaded as needed by the System Loader, and purged from memory when they are no longer needed. Run-time library files are not currently supported by the System Loader, 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 CPW are as follows:

$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 filetypes	

A CPW source file has an auxiliary type that represents the programming language for which it is to be used.

General Format for OMF Files

Each object module format (OMF) file contains one or more segments. Figure 9.1 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 (usually) relocatable entity that contains all the information needed to load it into memory.

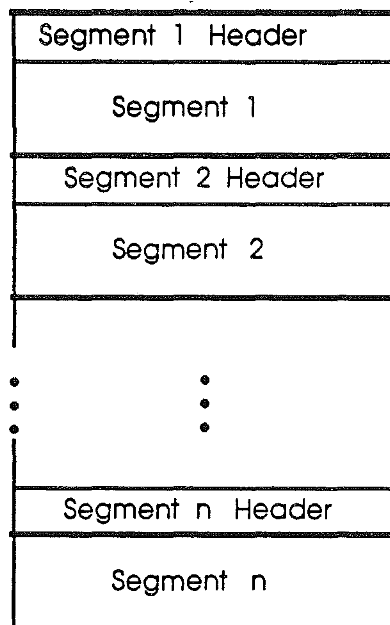


Figure 9.1. OMF File

Each segment 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" 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" 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 up to three attributes. The following segment types are defined by the object module format:

- code
- data
- jump table segment
- pathname segment
- library dictionary segment
- initialization segment
- absolute-bank segment
- zero-page/stack segment

The following segment attributes are defined by the object module format:

- static or dynamic
- position independent
- private

Code and data segments are 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, it is a data segment.

Jump table segments and **pathname segments** facilitate the dynamic loading of segments; they are described in the section "Load Files" in this chapter.

Library dictionary segments allow the linker to quickly scan the library file for needed segments; they are described in the section "Library Files" in this chapter.

Initialization segments are optional parts of load files; if used, they are loaded and executed immediately when they are found by the System Loader; they are used to perform any initialization required by the application during an inital load. Initialization segments are described in the section "Load Files" in this chapter.

Absolute-bank segments are restricted to a specified bank, but can be relocated within that bank.

Direct-page/stack segments are used to preset the zero page and stack registers for an application. See the section "Direct-Page/Stack Segments" in this chapter for more information.

Static segments 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, then all segments in your program are static

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.

A segment can have only one segment type but can have any combination of attributes (static and dynamic are mutually exclusive—together they comprise a single attribute). If more than one object segment is placed in a given load segment by the linker, then the load segment is flagged as code or data according to the *last* segment linked. 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 allow the System Loader to load individual load segments. The format of the segment header is illustrated in Figure 9.2. Following the figure 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. Always use DISPNAME and DISPDATA instead of absolute offsets when referencing LOADNAME, SEGNAME, and the start of the segment body in order to insure that future expansion of the segment header does not affect your program.

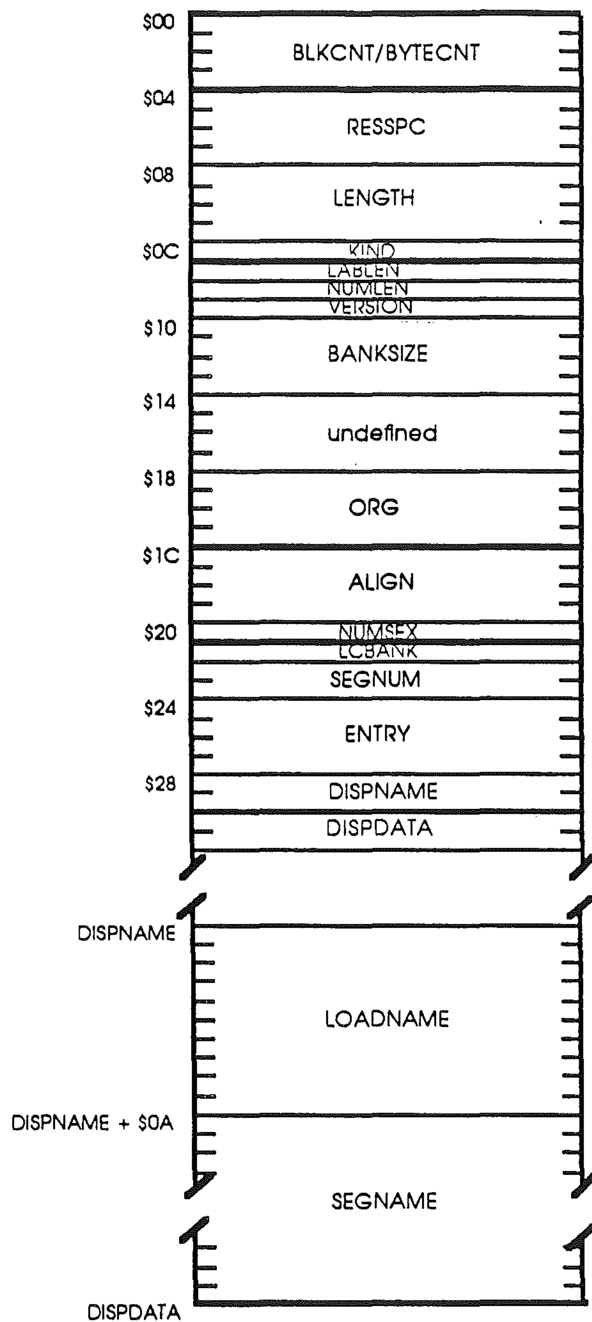


Figure 9.2. Segment Header

BLKCNT/BYTECNT: 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 filetype $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 zero bytes 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` 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`.

KIND: A 1-byte field specifying the type and attributes of the segment. The bits are defined as follows:

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
	$11	absolute-bank segment	Segment Types and Attributes
	$12	direct-page/stack segment	Direct-Page/Stack Segments
5—7	Segment Attribute		
5	1=position independent		Segment Types and Attributes
6	1=private		Segment Types and Attributes
7	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=($A1)`.

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 Cortland.

VERSION: A 1-byte field indicating the version number of the object module format with which the segment is compatible. This field is 1 for the initial 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's too large). If the segment is in a load file, the linker assures that the segment is loaded into a memory block that does not cross this boundary. For Cortland 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. Cortland data segments can use any number from $00 to $00010000 for BANKSIZE.

BANKSIZE is followed by four undefined bytes, reserved for future changes to the segment header specification.

ORG: A 4-byte field indicating the absolute address at which this segment is to be loaded in memory. 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 Cortland.

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 Cortland, this field must be a power of 2, less than or equal to $00010000.

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 Cortland.

LCBANK: 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 ($D000—$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. The language card and bank-switched ROM are described in the *Cortland Hardware Reference* manual.

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 Cortland, the least significant byte of each number field is first, as specified by NUMSEX. Several of the object module format records contain expressions that have to be evaluated by the linker. The operation and syntax of expressions are described in the next section, "Expressions." The operation codes and segment records are described in this section, listed in order of the opcodes. Table 9.1 provides an alphabetical cross reference between segment record types and opcodes.

Table 9.1. Segment-Body Record Types

Record Type	Op Code
ALIGN	$E0
BEXPR	$ED
CONST	$01—$DF
DS	$F1
END	$00
ENTRY	$F4
EQU	$F0
EXPR	$EB
GEQU	$E7
GLOBAL	$E6
INTERSEG	$E3
LCONST	$F2
LEXPR	$F3
LOCAL	$EF
MEM	$E8
ORG	$E1
RELEXPR	$EE
RELOC	$E2
STRONG	$E5
USING	$E4
ZEXPR	$EC

Record Type	Op Code	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 can not 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), then the location counter is decremented and subsequent code overwrites the old code.

RELOC $E2

This is a relocation record, 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, then the number is shifted to the left, filling vacated bit positions with 0's (logical shift left). If the bit-shift operator is (twos complement) negative, then 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
```

which 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; however `LAB|4+4` cannot be used, 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, then the number is shifted to the left, filling vacated bit positions with 0's (logical shift left). If the bit-shift operator is (twos complement) negative, then 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, then 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 one-load-file program, 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, then 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 this:

JSL EXT

where 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, then 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`):

```
E3030020 07000001 000A0045 030000
```

which corresponds to the following values:

$E3	operation code
$03	number of bytes to be relocated
$00	bit-shift operator
$00000720	offset of instruction
$0001	file number
$000A	segment number
$00000345	offset of subroutine referenced

When the loader processes the relocation dictionary, it uses the second 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`):

```
E3030020 07000001 00060045 2A0000
```

which corresponds to the following values:

$E3	operation code
$03	number of bytes to be relocated
$00	bit-shift operator
$00000720	offset of instruction
$0001	file number of jump table segment
$0006	segment number of jump table seg
$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 1-byte attribute fields. The label is assigned the current value of the 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; 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 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" in this chapter for a definition of private segments).

GEQU $E7

This record contains the name of a global 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" 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" in this chapter for a definition of private segments). Note that the linker ignores local labels from code segments, and 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" in this chapter for a definition of private segments).
DS	$F1	This record contains a number indicating how many bytes of 0's 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. LCONST is similar to CONST except that it allows for a much greater number of data bytes. Each relocatable 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 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" 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 $FFFF (65535).

`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, and it does not include the 2-byte file number. The `cINTERSEG` record can be used only if both offsets are less than $FFFF (65535) 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.

Expressions

Several of the object module format 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, 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 twos-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, then the first number is shifted to the left, filling vacated bit positions with 0's (logical shift left). If the second number is negative, then 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 twos-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 twos-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. The location counter is loaded before the bytes from the expression are placed into the code stream, so this 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 acted on as follows:

- $82 Weak reference (see note).
- $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" 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:

```
        LDX     #MSG4-MSG3
```

This line would be assembled into two OMF records:

```
CONST ($01)     A2
EXPR ($EB)      01 : MSG4 MSG3 -
```

In hexadecimal format, these records appear as follows:

```
01A2EB01 83044D53 47348304 4D534733|  "k  MSG4  MSG3
```

```
0200
```

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 op code 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 ($01 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), and `MSG4` is 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 4.

Direct-Page/Stack Segments

The Corltand stack can be located anywhere in the lower 48K bytes of bank $00, and can be any size up to 48K bytes. The direct page is the Corltand equivalent of the zero page of 8-bit Apple II's; the direct page can also be located anywhere in the lower 48K bytes of bank $00, and be up to 48K bytes in length. Since more than one application can be loaded in memory at one time on the Cortland, however, there may be more than one stack and one direct page in bank $00. Furthermore, some applications may place some of their code in bank $00. A given program should therefore probably not use more than about 4K bytes for stack and direct page.

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 Cortland, therefore, is to allocate a single block of memory to the direct page and stack; the direct page occupies the lower part of the allocated space, and 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 /NUMBER=$12 DIRSTACK
  SELECT MYOBJ.A (DEFPAGE)
    •
    •
    •
```

LinkEd is described in Chapter 7.

Library Files

Library files (ProDOS 16 filetype $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 library-file segments are not aligned to 512-byte boundaries, and 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 4). The format of the library dictionary segment is illustrated in Figure 9.3.

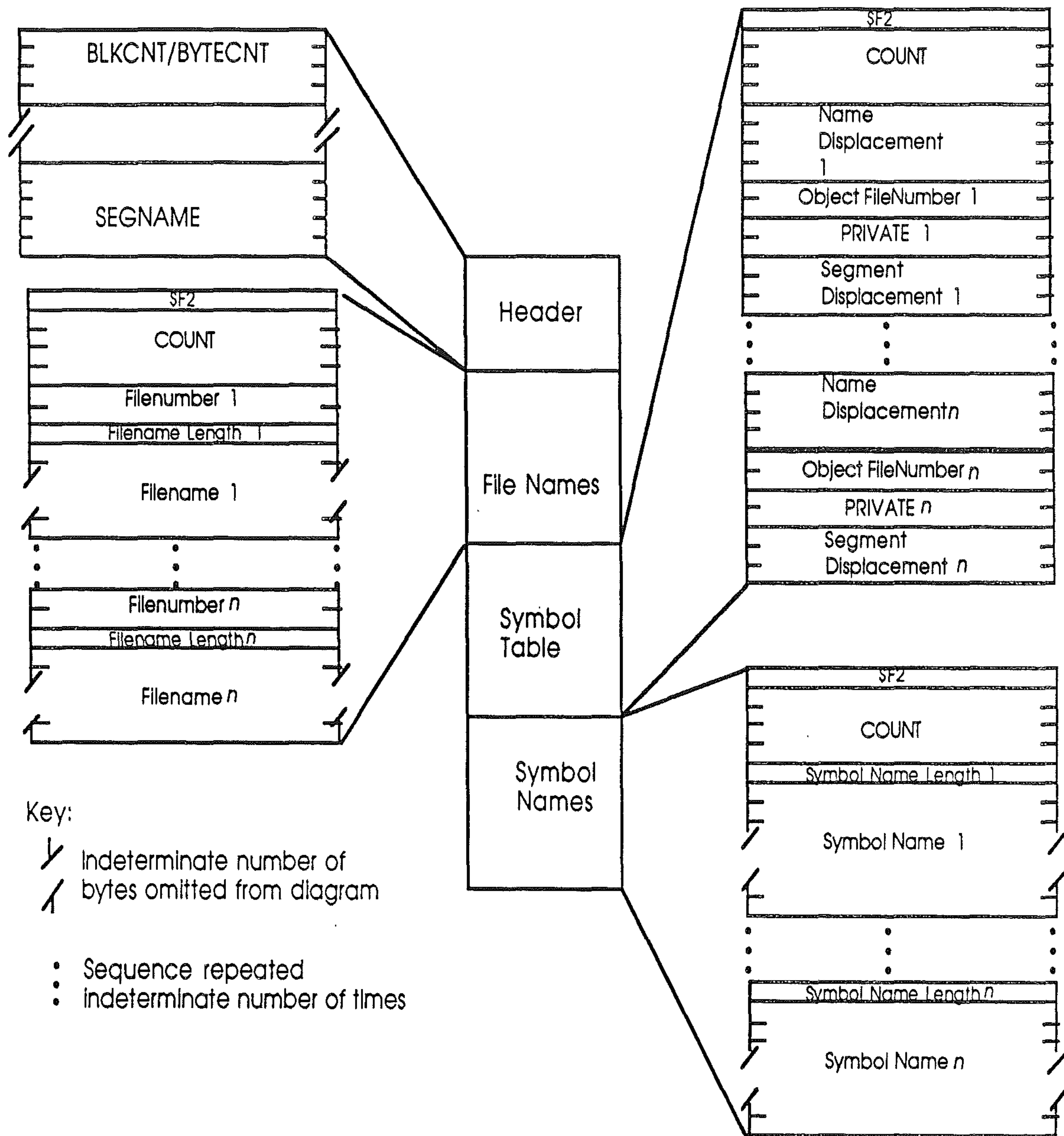


Figure 9.3. Library Dictionary Segment

The library dictionary segment begins with a segment header, which is identical in form to other segment headers except that it contains a BYTECNT field instead of the BLKCNT field. 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 are 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 filenumber 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 =1, then 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 = 0, then 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 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 4 for a discussion of the MAKELIB utility).

Load Files

Load files (ProDOS 16 file types $B3–$BE) contain the load segments that are moved into memory by the System Loader. They 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) records and intersegment (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 jump-table-segment entry contains the file number, segment number, and offset of the reference in the dynamic segment, and 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` and `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" 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 *Cortland ProDOS 16 Technical 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 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 jump table), of the jump table segments in memory.

Each entry in the jump table segment corresponds to a call to an external (inter-segment) 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 System Loader unloads 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, segment number, and 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 *Cortland 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—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)

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 *Cortland 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:

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, then 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 8 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 *Cortland ProDOS 16 Reference* manual.

Currently, run-time library files and multiple load files are not supported by the linker and the System Loader; the pathname segment is created, but 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.

An example of use of the initialization segment is to initialize the graphics environment used by 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 other segments (that is, no `INTERSEG` records can be used).
- It must exit with an `RTL` instruction.

Run-Time Library Files

Run-time library files (ProDOS 16 file type $B4) contain dynamic load segments that the System Loader can load when they are referenced through the jump table. Usually, these 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 like references to dynamic segments in any other load file.

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 or System Loader; 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 CPW 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, so that the program can use shell facilities during execution. The program returns control to the shell with a ProDOS 16 `QUIT` call, which the shell must intercept and act on. Shell load files should use standard Text Toolkit calls for all I/O; the shell program is responsible for initializing the text toolkit routines.

> **Note:** A load file of file type $B5 can be launched by ProDOS via the `QUIT` call if it requires no support other than standard input from the keyboard and output to the screen. ProDOS initializes the text toolkit to use the Pascal I/O drivers (see the *Cortland Toolbox Reference*) for the keyboard and 80-column screen. A program launched in this way does not operate under a shell.

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 CPW 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 exit with a ProDOS `QUIT` call.
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. If the shell load file does not include a direct-page/stack segment, the shell must set up a direct page and stack area for the shell load file. 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 *Cortland 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 have ProDOS 16 filetype $B5. 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 requested any user ID's, it must release all memory buffers with those user ID's.
2. 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 (non-specific) 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.
3. The shell load file should execute a ProDOS 16 `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.

Chapter 10

Shell Calls

The Cortland 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 CPW 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 10-1.

Table 10-1. 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 a Cortland tool call
`Set`	($0106)	Sets the value of a shell variable
`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
`Read`	($010B)	Reads the value of a shell variable
`Execute`	($010D)	Sends a command or list of commands to the shell command interpreter
`Switch_Window`	($010E)	Returns control to the shell when the mouse is clicked in a window owned by another program
`Redirect`	($0110)	Sets device and file for I/O redirection
`Is_Window`	($0112)	Determines whether a file is open as a window on the desktop
`Stop`	($0113)	Detects a request for an early termination of the program
`Read_Indexed`	($0???)	Reads variable table

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; additional information for some calls is passed in hardware registers. In practice, the shell-call block is normally executed by an assembler macro. This section discusses these aspects of shell calls.

> **Important:** Although shell calls are called exactly like ProDOS 16 calls, this section does not provide all of the information relevant to ProDOS 16 calls. ProDOS 16 calls are described in the *Cortland ProDOS 16 Reference* manual.

> **Note:** This chapter assumes that you are using the CPW Assembler to make shell calls. See the *Cortland Programmer's Workshop Assembler Reference* for more information on the CPW Assembler. If you want to access shell calls from a program written in another language, you will probably have to integrate an assembly-language routine into that program. Some languages provide their own assembly-language interface; if the language you are using does not, you can use the techniques illustrated in Chapter 3 of this manual to combine an assembly-language subroutine with routines written in another language.

The Call Block

A shell-call block consists of a `JSR` (jump to subroutine) or a `JSL` (long jump to subroutine) to the ProDOS 16 entry point***is that right?***, followed by a 2-byte system call number and a 4-byte parameter block pointer. The CPW Shell intercepts the call and determines whether it is a CPW 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 must 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`.

A JSL rather than a JSR is required because the JSL uses a 3-byte address, allowing a caller to make the call from anywhere in memory. The JSR instruction uses only a 2-byte address, restricting it to jumps and returns within the current (64K) bank of memory.

Shell-Call Macros

For each call listed in Table 10-1, there is a CPW Assembler macro that you can use to make the call. The macro call consists of the name of the call (as shown in Table 10-1), with the address of the parameter block in the operand field. For example, to call the Get_LInfo function, use the following sequence:

```
            MCOPY       MY.MACROS       ; 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, 1 or more words long, that the caller passes to the shell through the parameter block. It is an input parameter.
- A **result** is a numeric quantity, 1 or more words 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; the data it points to may be either input or output.

A parameter may be both a value and a result. Also, a pointer may designate a location that contains a value, a result, or both.

> **Strings:** 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 CPW Shell call references a parameter block, which may be anywhere in memory. All applications must obtain needed memory from the Memory Manager, and therefore 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:

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.

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 *Cortland ProDOS 16 Reference* manual.

Register Values

There are no register requirements on entry to a shell call. The CPW 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 these values:

A	zero if call successful; if nonzero, 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 CPW 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	unchanged
i	unchanged
z	undefined
c	zero if call successfull, 1 if not
e	undefined

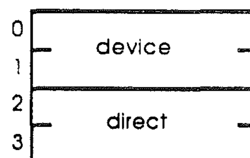
Call Descriptions

This section lists each of the shell calls, describes its use, and describes 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 in alphabetical order. Table 10-1 lists all of 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 function can be used by a program to determine whether to send form feeds to standard output, for example.

Parameter List:



Offset	Label	Parameter Name	Size and Type [range of values]
$00–$01	`device`	Device number	2-byte value [$0000–$0002]
		This parameter indicates which type of input or output has been redirected, as follows: $0000 Standard input $0001 Standard output $0002 Error output	
$02–$03	`direct`	Direction	2-byte value [$0000–$0002]

This parameter indicates the type of redirection that has occurred, as follows:

$0000	Console
$0001	Printer
$0002	Disk file

Possible Errors

$53	Parameter out of range

Error ($0105)

When a Cortland 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 List:

Offset	Label	Parameter Name	Size and Type [range of values]
$00–$01	`error`	Error number	2-byte value [$0000–$FFFF]
		This parameter specifies the error number returned by the tool call.	

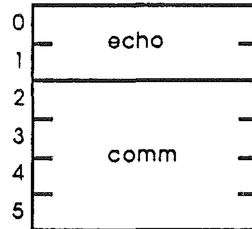
Possible Errors

None

Execute ($010D)

This function sends a command or list of commands to the CPW Shell command interpreter. *****Do I need to say something about the Shell `EXECUTE` command or the local variable table? Has the `echo` parameter been eliminated?*****

Parameter List:



Offset	Label	Parameter Name	Size and Type [range of values]
$00–$01	`echo`	Echo command flag	2-byte value [$0000–$0001]
		*****has this parameter been eliminated?*****If you set this flag to 1 (binary), then the commands being executed are sent to standard output (unless the variable `{echo}` is null). This flag should be set to 0 for interactive-level commands, and to 1 for Exec files. Exec files and variables are described in the section "Exec Files" in Chapter 4.	
$02–$05	`comm`	Address of command string	4-byte pointer [$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). Terminate the command string with a null character ($00). Any output is sent to standard output. If the variable `{exit}` is not null and any command returns a non-zero error code, then any remaining commands are ignored. Error codes and variables are described in the section "Exec Files" in Chapter 4.	

Possible Errors

$??	Memory manager errors for allocate memory, lock, unlock calls
others	Error returned by the last command executed
$30	Error on exit from an Exec file (command returned a non-zero error code) ***???is there such an error?***

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 4, and are listed in Appendix A.

Parameter List:

Offset	**Label**	**Parameter Name**	**Size and Type [range of values]**
$00-$01	`lang`	Language number	2-byte result [$0000–$7FFF]
		The current CPW language number. The current language number is set by the CPW Editor when it opens an existing file, or by the user with a CPW Shell command.	

Possible Errors

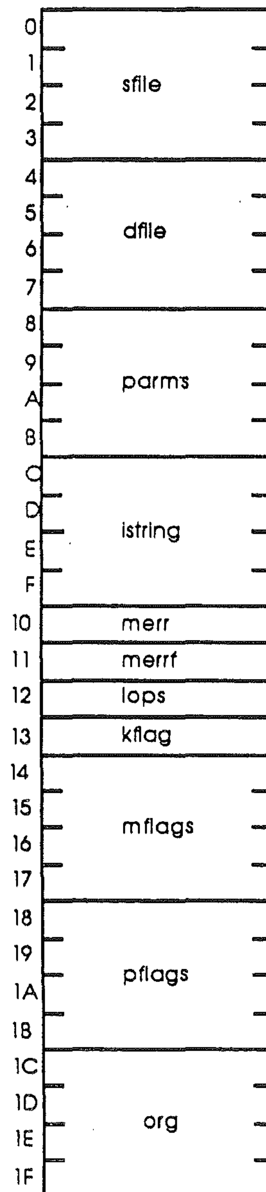
None

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; when the CPW Shell returns control to your program, you can then read the parameter block to obtain the information you need.

Use the `Set_LInfo` call when your program is finished before executing an `RTL` or ProDOS 16 `QUIT` call to return control to the shell.

Parameter List:



Offset	Label	Parameter Name	Size and Type [range of values]
$00–$03	`sfile`	Address of source filename	4-byte pointer [$0000 0000–$00FF FFFF]
		The address of a 65-byte-long buffer containing 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 a partial or full pathname.	
$04–$07	`dfile`	Address of output filename	4-byte pointer [$0000 0000–$00FF FFFF]
		The address of a 65-byte-long buffer containing 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 a partial or full pathname.	
$08–0B	`parms`	Address of parameter list	4-byte pointer [$0000 0000–$00FF FFFF]
		The address of a 256-byte-long buffer containing the list of names from the `NAMES=` parameter list in the CPW Shell command that called the assembler or compiler. If there was no `NAMES` parameter list, the buffer pointed to by `parms` begins with $00.	
$0C–$0F	`istring`	Address of input strings	4-byte result [$0000 0000–$00FF FFFF]
		The address of a 256-byte-long buffer containing the string of commands to be passed on to a specific language compiler. For example, if the `COMPILE` command includes the parameter `C=(-I/CINCLUDES/)`, then the string enclosed in parentheses is found in that buffer when the C compiler is called.	
$10	`merr`	Maximum error level allowed	1-byte result [$00–$10]
		If the maximum error level found by the assembler, compiler, or linker (`merrf`) is greater than `merr`, then the CPW 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`	Maximum error level found	1-byte result [$00–$FF]

This field is used by the `Set_LInfo` call to return the maximum error level found. This field is undefined for the `Get_LInfo` call.

$12	`lops`	Operations flags	1-byte result [$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 as follows:

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 (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 CPW Shell, it clears bit 0 unless a file with another language is appended to the source.

$13	`kflag`	Keep flag	1-byte result [$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` (compilers and assemblers only). For example, if the output filename pointed to by `dfile` is `PROG`, then the first segment to be executed should be put in `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 8 for more information on object-file naming conventions.

$14–$17 `mflags` Flags with a minus sign 4-byte result [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–Z. For each flag set with a minus sign, the corresponding bit is set to 1. See the discussions of the `ALINK` and `ASML` commands in Chapter 4 for descriptions of these option flags.

$18–$1B `pflags` Flags with a plus sign 4-byte result [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–Z. For each flag set with a minus sign, the corresponding bit is set to 1. See the discussions of the `ALINK` and `ASML` commands in Chapter 4 for descriptions of these option flags.

$1C–$1F `org` Origin 4-byte result [$0000 0000–$FFFF FFFF]

The start address of the load file. The origin is used only by the linker. This field is also used on entry to an editor to provide a

displacement into the file. The editor can then place at the top of the screen the line that corresponds to this displacement.

Possible Errors

None

Init_Wildcard ($0109)

This function provides to the CPW 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, so you can call this function every time you want to search for a filename.

Parameter List:

Offset	**Label**	**Parameter Name**	**Size and Type [range of values]**
$00–$03	`file`	Address of filename.	4-byte pointer *****2 bytes??***** [$0000 0000–$00FF FFFF]

The address of a buffer containing a filename that includes a wildcard character. Examples of such filenames are:

```
A=
/CPW/MYPROGS/?.ROOT
```

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 (`?`), then 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 "Wildcards" in Chapter 2.

Possible Errors

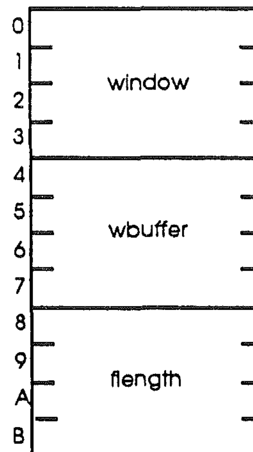
Errors for the following ProDOS 16 and Memory Manager calls. See the *Cortland ProDOS 16 Reference* manual and the *Cortland Toolbox Reference* manual for descriptions of these errors. *****Is this list correct? Complete?*****

Open
Read
Close
Dispose
Get_Info
Get end of file
Lock
Allocate new memory

Is_Window ($0112)

Use this function to find out if a file is open as a window on the desktop, and to get a pointer to the start of the file if it is open as a window. If the file is open on the desktop, this function also returns the length of the file in bytes.

Parameter List:



Offset	Label	Parameter Name	Size and Type [range of values]
$00–$03	`window`	Pointer to file name	4-byte pointer [$0000 0000–$00FF FFFF]
		The address of a buffer containing the name of the file that you want information about.	
$04–$07	`wbuffer`	Pointer to file buffer	4-byte pointer [$0000 0000–$00FF FFFF]
		If the file is open as a window on the desktop, the shell returns the address of the start of the file. If the file is not open, the shell returns a zero in this field. **Note:** Since windows have not yet been implemented in the Cortland Programmer"s Workshop, this call always returns a 0 in the wbuffer field.	
$08–$0B	`flength`	Length of file	4-byte result [$0000 0000–$00FF FFFF]
		If the file is open as a window on the desktop, the shell returns the length of the file in bytes. If the file is not open on the desktop, this field is undefined.	

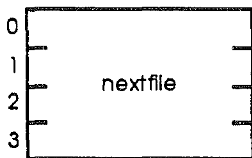
Possible Errors

$40 Invalid pathname syntax

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 that matches the wildcard filename. For example, if the wildcard filename specified in `Init_Wildcard` were `/CPW/UTILITY/XREF.?`, then the first filename returned by the shell in response to a `Next_Wildcard` call might be `/CPW/UTILITY/XREF.ASM65816`.

Parameter List:



Offset	Label	Parameter Name	Size and Type [range of values]
$00–$03	`nextfile`	Address of next filename	4-byte pointer [$0000 0000–$00FF FFFF]

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 length zero). See also the description of `Init_Wildcard`.

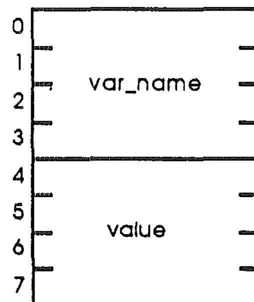
Possible Errors

None

Read ($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 and any Exec files called from that file, or for the interactive command interpreter and all Exec files called from the command interpreter (if that is the command level in use). Variables and Exec files are described in the section "Exec Files" in Chapter 4. Use the `Set` call to set the value of a variable.

Parameter List:



Offset	**Label**	**Parameter Name**	**Size and Type [range of values]**
$00–$03	`var_name`	Pointer to name of variable	4-byte pointer [$0000 0000–$00FF FFFF]
		This is a pointer to a 256-byte 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 ASCII characters.	
$04–$07	`value`	Pointer to value of variable	4-byte pointer [$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. The value consists of a null string (that is, the length byte is $00) for an undefined variable. *****who writes this pointer—you or the shell? is the buffer in the shell's space or your program's?*****	

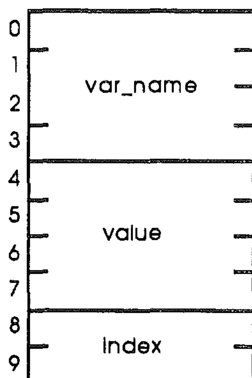
Possible Errors

None

Read_Indexed ($01??)***???***

You can use this function to read the contents of the variable table for the command level at which the call is made. You repeat this call, incrementing the index number by 1 each time, until the entire contents of the variable table have been returned. The CPW Shell's SET command executes this command when it is executed with no parameters, for example.***is that true?***

Parameter List:



Offset	Label	Parameter Name	Size and Type [range of values]
$00–$03	`var_name`	Pointer to name of variable	4-byte pointer [$0000 0000–$00FF FFFF]
		This is a pointer to a 256-byte buffer in which the shell places 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`	Pointer to value of variable	4-byte pointer [$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. The value consists of a null string (that is, the length byte is $00) for an undefined variable.	
$08–$09	`index`	Index number	2-byte value [$0000–$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

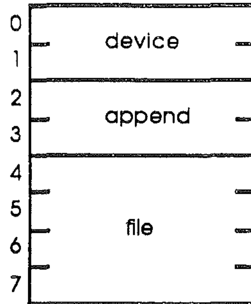
Errors for the following Memory Manager calls. See the *Cortland Toolbox Reference* manual for descriptions of these errors. ***Is this list correct? Complete?*****

Lock
Unlock

Redirect ($0110)

This function instructs the shell to redirect input or output to the printer, console, or a disk file.*****Is there no Redirect Error call?*****

Parameter List:



Offset	Label	Parameter Name	Size and Type [range of values]
$00–$01	`device`	Device number	2-byte value [$0000–$FFFF]
		This parameter indicates the number of the device to or from which I/O is to be redirected.	
$02–$03	`append`	Append flag	2-byte value [$0000–$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`	Address of filename	4-byte pointer [$0000 0000–$00FF FFFF]
		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 *Cortland ProDOS 16 Reference* manual and the *Cortland Toolbox Reference* manual for descriptions of these errors. *****Is this list correct? Complete?*****

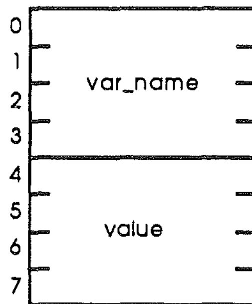
Open
Close

Read
Write
Get end of file

Set ($0106)

This function sets the value of a variable. If the variable has not been previously defined, this function defines it. The value is valid for the currently-executing Exec file and any Exec files called from that file, or for the interactive command interpreter and all Exec files called from the command interpreter (if that is the command level in use). Variables and Exec files are described in the section "Exec Files" in Chapter 4. Use the `Read` call to read the current value of a variable and the `Read_Indexed` call to read a variable table.

Parameter List:



Offset	Label	Parameter Name	Size and Type [range of values]
$00–$03	`var_name`	Pointer to name of variable	4-byte pointer [$0000 0000–$00FF FFFF]
		This is a pointer to a buffer. The buffer contains the name of the variable whose value you wish to change. The name consists of a length byte and a string of ASCII characters.*****how long are the buffers? Does it matter?*****	
$04–$07	`value`	Pointer to value of variable	4-byte pointer [$0000 0000–$00FF FFFF]
		This is a pointer to a buffer. The buffer contains the value to which the variable is to be set. The value is an ASCII string.	

Possible Errors

$54 Out of memory

Errors for the following Memory Manager calls. See the *Cortland Toolbox Reference* manual for descriptions of these errors. *****Is this list correct? Complete?*****

Lock
Unlock
Grow
New

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 4, and are listed in Appendix A.

Parameter List:

Offset	Label	Parameter Name	Size and Type [range of values]
$00-$01	`lang`	Language number	2-byte value [$0000–$7FFF]
		The CPW language number to which the current CPW language should be set. If the language specified is not installed (that is, not listed in the command table), then the "language not available" error is returned..	

Possible Errors

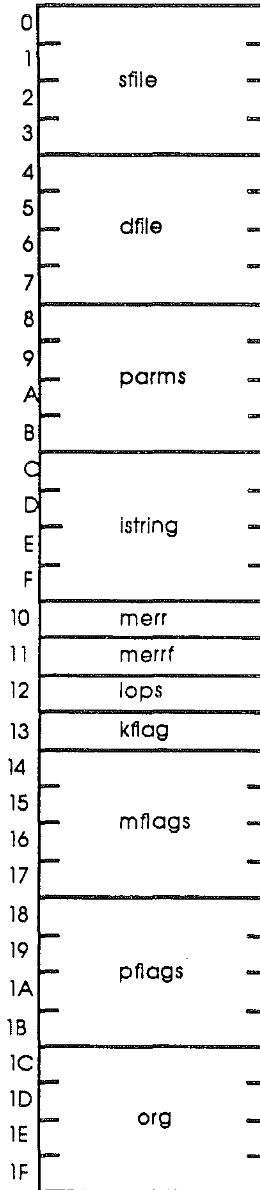
$80 Language not available

Set_LInfo ($0102)

This function is used by an assembler, compiler, linker, or editor to pass parameters to the CPW Shell before executing an `RTL` or ProDOS 16 `QUIT` call to return control to the shell.

Use the `Get_LInfo` call to read parameters passed to your assembler, compiler, linker, or editor.

Parameter List:



Offset	Label	Parameter Name	Size and Type [range of values]

Offset	Name	Description	Size / range
$00–$03	sfile	Address of source filename	4-byte pointer [$0000 0000–$00FF FFFF]

The address of a buffer containing the filename of the source file; that is, the next file that a compiler or assembler is to process. The filename can be any valid ProDOS 16 filename, and can be a partial or full pathname.

Offset	Name	Description	Size / range
$04–$07	dfile	Address of output filename	4-byte pointer [$0000 0000–$00FF FFFF]

The address of a buffer containing 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 a partial or full pathname.

Offset	Name	Description	Size / range
$08–0B	parms	Address of parameter list	4-byte pointer [$0000 0000–$00FF FFFF]

The address of a buffer containing the list of names from the NAMES= parameter list in the CPW Shell command that called the assembler or compiler.

Offset	Name	Description	Size / range
$0C–$0F	istring	Address of input strings	4-byte pointer [$0000 0000–$00FF FFFF]

A placeholder for the address of a buffer containing the string of commands passed to the compiler. This command string is not reused by the shell, so it is not necessary to pass it back to the shell with the Set_LInfo call.

Offset	Name	Description	Size / range
$10	merr	Maximum error level allowed	1-byte value [$00–$10]

If the maximum error level found by the assembler, compiler, or linker is greater than merr, then 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.

Offset	Name	Description	Size / range
$11	merrf	Maximum error level found	1-byte value [$00–$FF]

This field is used by the Set_LInfo call to return the maximum error level found. If merrf is greater than merr, then no further processing is done by the shell. If the high bit of merrf is set, then 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 and control is passed by the shell to the CPW Editor. See also the discussion of the org parameter.

$12	`lops`	Operations flags	1-byte value [$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 as follows:

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 (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 CPW Linker by compiling a LinkEd file, the linker clears bits 0 and 1.

$13	`kflag`	Keep flag	1-byte value [$00–$03]

This flag indicates what files have been created by a compiler or assembler, as follows:

`Kflag` Value	**Meaning**
$00	Do not save output.
$01	No file has been created. This would be the case if a fatal error had been found or `merrf` were greater than `merr`. *****Byte Works: I have a question on this, too long to put in here.*****
$02	Only the `.ROOT` file has been created. This would be the case if only one segment were compiled.
$03	At least one alphabetic suffix has been used.

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 8 for more information on object-file naming conventions.

$14–$17	`mflags`	Flags with a minus sign	4-byte value [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–Z. For each flag set with a minus sign, the corresponding bit is set to 1. See the discussions of the `ALINK` and `ASML` commands in Chapter 4 for descriptions of these option flags.

$18–$1B	`pflags`	Flags with a plus sign	4-byte result [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–Z. For each flag set with a plus sign, the corresponding bit is set to 1. See the discussions of the `ALINK` and `ASML` commands in Chapter 4 for descriptions of these option flags.

$1C–$1F	`org`	Origin	4-byte value [$0000 0000–$FFFF FFFF]

The start address of the load file. The origin is used only by the linker. When a compile or assembly terminates with a fatal error (`merrf`=$FF), the compiler or editor should put the displacement of the line containing the error into the `org` field. The editor can then place that line at the top of the screen.

Possible Errors

None

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 List:

Offset	Label	Parameter Name	Size and Type [range of values]
$00-$01	`stop`	Stop flag	2-byte result [$0000–$0001]

This flag is set ($0001) by the shell when it finds a ♂-. in the keyboard buffer. When a CPW 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 ♂-. 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.

Possible Errors

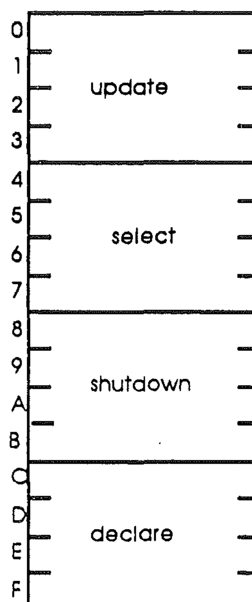
None

Switch_Window ($010E)

When a CPW utility program detects a mouse-down event in a window that it does not own, it uses this function to return control to the CPW Shell. When the shell needs to return to the utility, it executes a `JSL` to one of the vectors that you provide in the `Switch_Window` call. The shell saves the contents of the direct page and stack when you execute the `Switch_Window` call, and restores them when it reenters the utility.

> **Note:** This call has no effect at present, since windows have not yet been added to the CPW interface.

Parameter List:



Offset	Label	Parameter Name	Size and Type [range of values]
$00–$03	`update`	Reentry vector for window updates	4-byte value [$0000 0000–$00FF FFFF]
		If the window must be updated but has not been selected, enter the utility through this vector. This vector should point to a routine that repaints the window on the screen, as when another window is moved from in front of it.	
$04–$07	`select`	Reentry vector when window is selected	4-byte value [$0000 0000–$00FF FFFF]
		If the window has been selected, enter the utility through this vector. This vector should point to a restart routine that reactivates the program without having to reload it. Use the window declaration record (see the `declare` parameter) to determine which window has been selected.	

$08–$0B	`shutdown`	Reentry vector for shut-downs	4-byte value [$0000 0000–$00FF FFFF]
		If a system shutdown is in progress, enter the utility through this vector. This vector should point to a routine that closes all windows owned by the utility, saves files and opens attention boxes as necessary, terminates the utility, and returns control to the shell.	
$0C–$0F	`declare`	Pointer to window declaration record	4-byte pointer [$0000 0000–$00FF FFFF]
		This parameter points to the window manager's window declaration record. When the user clicks the mouse inside a window, this record indicates which window it was. The window manager is described in the *Cortland Toolbox Reference* manual	

Possible Errors

None

FOOLISH NOTIONS

Appendixes

Appendix A

Command Summary

This appendix lists the currently-defined CPW language types, and summarizes the commands used in the CPW Shell, Exec files, CPW Editor, CPW Debugger, and LinkEd files.

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.
italics	Italics indicate a variable, such as a filename or address.
prefix	This parameter indicates any valid directory pathname or partial pathname. It does *not* include a filename. If the volume name is included, *prefix* must start with a slash (/); if *prefix* does not start with a slash, then the current prefix is assumed. For example, if you are copying a file to the subdirectory `SUBDIRECTORY` on the volume `VOLUME`, then the *prefix* parameter would be: `/VOLUME/SUBDIRECTORY/`. If the current prefix were `/VOLUME/`, then you could use SUB`DIRECTORY` for *pathname* . The device numbers `.D1, .D2, . . . .D`*n* can be used for volume names; if you use a device number, do not precede it with a slash. For example, if the volume VOLUME in the above example were in disk drive `.D1`, then you could enter the *prefix* parameter as `.D1/SUBDIRECTORY/`.
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 pathename, in which the current prefix is assumed. For example, if a file is named FILE in the subdirectory `DIRECTORY` on the volume `VOLUME`, then the *pathname* parameter would be: `/VOLUME/DIRECTORY/FILE`. If the current prefix were `/VOLUME/`, then you could use `DIRECTORY/FILE` for *pathname* . 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 unit names `.CONSOLE` and `.PRINTER` can be used as filenames; the device numbers `.D1, .D2, . . . .D`*n* can be used for volume names.
|	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 (B underlined)	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.
. . .	Vertical elipses 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 CPW.

Language	Number	Use
ASM6502	2	6502 Assembler
ASM65816	3	65816 Assembler
BASIC	4	CPW BASIC
BWBASIC	9	Byte Works BASIC
BWC	8	Byte Works C
BWPASCAL	5	Byte Works Pascal
C	10	CPW C
COMMAND	12	CPW command-processor window
EXEC	6	Command file
LINKED	13	CPW Linker command language
PASCAL	11	CPW Pascal
PRODOS	0	ProDOS 16 text file (ProDOS 16 filetype $04)
SMALLC	7	Byte works small C
TEXT	1	CPW text file

Shell

`ALINK [+L|-L] [+S|-S]` *sourcefile* `[KEEP=`*outfile*`]`
Compile a linker command file

`ASM65816`
Change default language to 65816 Assembly Language

`ASML [+L|-L] [+S|-S]`*sourcefile* `[KEEP=`*outfile*`]`
`[NAMES=(`*seg1* `[` *seg2* `[ ...]]) ] [`*language1*`=(`*option* ...`)`
`[`*language2*`=(`*option* ...`) ...]]`
Assemble and link the program

`ASMLG [+L|-L] [+S|-S]`*sourcefile* `[KEEP=`*outfile*`]`
`[NAMES=(`*seg1* `[` *seg2* `[ ...]]) ] [`*language1*`=(`*option* ...`)`
`[`*language2*`=(`*option* ...`) ...]]`
Assemble, link, and go (run the program)

`ASSEMBLE [+L|-L] [+S|-S]`*sourcefile* `[KEEP=`*outfile*`]`
`[NAMES=(`*seg1* `[` *seg2* `[ ...]]) ] [`*language1*`=(`*option* ...`)`
`[`*language2*`=(`*option* ...`) ...]]`
Assemble the program

`C`
Change default language to CPW C

`CATALOG [`*prefix*`]`
`CATALOG [`*pathname*`]`

List the disk directory

CHANGE *pathname language*
Change the language type of an existing source file

CMPL [+L|-L] [+S|-S] *sourcefile* [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2* [...]])] [*language1*=(*option* ...)
[*language2*=(*option* ...) ...]]
Compile and link the program

CMPLG [+L|-L] [+S|-S] *sourcefile* [KEEP=*outfile*]
[NAMES=(*seg1* [,*seg2* [,...]])] [*language1*=(*option* ...)
[*language2*=(*option* ...) ...]]
Compile, link, and go (run the program)

COMMANDS *pathname*
Read the command table

COMPILE [+L|-L] [+S|-S] *sourcefile* [KEEP=*outfile*]
[NAMES=(*seg1* [,*seg2* [,...]])] [*language1*=(*option* ...)
[*language2*=(*option* ...) ...]]
Compile the program

COMPRESS A|C|A C [*prefix*[/]]
Compress and/or alphabetize the disk directory

COPY [-C] *pathname1* [*prefix2*/] [*filename2*]
Copy a file

COPY *prefix1 prefix2*
Copy a directory

COPY *volume1 volume2*
Copy a volume

CREATE *prefix*[/]
Create a new subdirectory

CRUNCH *rootname*
Combine object modules formed by partial compiles or assemblies into a single file

DEBUG
Execute the CPW Debugger program

DELETE *pathname*
Delete a file

DISABLE D|N|B|W|R *pathname*
Disable file attributes

DUMPOBJ [*option* ...] *pathame* [NAMES=(*seg1 seg2* ...)]
List the contents of an OMF file to standard output

EDIT *pathname*
Edit an existing file, or open a new file

ENABLE D|N|B|W|R *pathname*
Enable file attributes

EXEC
Change default language to EXEC command language

EXECUTE *pathname* [*paramlist*]
Execute an Exec file at present command level

FILETYPE *pathname filetype*
Change filetype to type specified

HELP [*commandname*]
Provide on-screen help for commands, or list all available commands

INIT *device* [*name*]
Initialize a disk

LINK [+L|-L] [+S|-S] *objectfile* [KEEP=*outfile*]
LINK [+L|-L] [+S|-S] (*objectfile1 objectfile2* ...) [KEEP=*outfile*]

Link an object module

LINKED
Change default language to the LinkEd command language

MACGEN [±C|-C] *infile outfile macrofile1* [*macrofile2* ...]
Generate a macro library for a specific program

MAKELIB [-F] [-D] *libfile* [+|-|^*objectfile1* +|-|^*objectfile2* . . .]
Generate a library file from an object module

MOVE [-C] *pathname1* [*prefix*/] [*filename2*]
Move a file to another directory or volume

PREFIX [*n*] *prefix*[/]
Change the default prefixes

PRODOS
Change default language to ProDOS 16 text

QUIT
Quit CPW

RENAME *pathname1 pathname2*
Change a filename

RUN [+L|-L] [+S|-S]*sourcefile* [KEEP=*outfile*]
  [NAMES=(*seg1* [,*seg2* [,...]])] [*language1*=(*option* ...)
  [*language2*=(*option* ...) ...]]
Same as ASMLG or CMPLG

SHOW LANGUAGE| LANGUAGES| PREFIX *n*| TIME| UNITS
Show languages, system default language, prefixes, time, volumes on line

SWITCH *pathname1 pathname2*
Change the positions of two files in a directory

TEXT
Change default language to TEXT

TYPE [+N] *pathname1* [*startline1* [*endline1*]]
  [*pathname2* [*startline2* [*endline2*]] . . .]
Type a file to standard output

EXEC Files

BREAK

Terminates the innermost FOR, LOOP, or IF statement currently executing

CONTINUE
Causes control to skip over following statements to the next END statement

ECHO *string*
Writes messages to the screen

EXECUTE *pathname [paramlist]*
Executes an Exec file at present command level

EXIT [*number*]
Terminates execution of the Exec file

EXPORT [*variable* ...]
Makes the listed variables available to Exec files called by the current exec file *****Can more than one variable be listed, as in MPW, or not?*****

FOR *variable* [IN *value1 value2* ...]
•
•
•
END
Creates a loop that is executed once for each parameter-value listed

IF *expression*
•
•
•
[ELSE IF *expression*]
•
•
•
[ELSE]
•
•
•
END
Provides conditional branching in Exec files

LOOP
•
•
•
END
Defines a loop that repeats continuously until a BREAK command is encountered

SET [*variable* [*value*]]
Assigns a value to a variable name

UNSET *variable*

Deletes the definition of a variable

Editor

Beep the Speaker	|CTRL|-G
Bottom of Screen	|CTRL|-B
Bottom of Screen / Page Down	|CTRL|--J -↓
Change	See Search and Replace.
Clear	See Delete.
Copy	|CTRL|-C -C
Cursor Down	|CTRL|-J ↓
Cursor Left	|CTRL|-H ←
Cursor Right	|CTRL|-U →
Cursor Up	|CTRL|-K ↑
Cut	|CTRL|-X -X
Define Macros	-|ESC|
Delete	-|DELETE|
Delete Character	|CTRL|-F -F |ESC| G
Delete Character Left	|DELETE| |CTRL|-D
Delete Line	|ESC| Y
Delete to EOL	|CTRL|-Y -Y
Delete Word	|ESC| |DELETE|

<table>
<tr><td>End of Line</td><td>-.
-></td></tr>
<tr><td>Find</td><td>See Search.</td></tr>
<tr><td>Help</td><td>-?
-/</td></tr>
<tr><td>Insert Line</td><td>|ESC| B</td></tr>
<tr><td>Insert Space</td><td>|ESC| H</td></tr>
<tr><td>Paste</td><td>|CTRL|-V
-V</td></tr>
<tr><td>Quit</td><td>|CTRL|-Q
-Q</td></tr>
<tr><td>Remove Blanks</td><td>|CTRL|-R
-R</td></tr>
<tr><td>Repeat Count</td><td>1 to 32767</td></tr>
<tr><td>Return</td><td>|RETURN|
|CTRL|-M</td></tr>
<tr><td>Screen Moves</td><td>-1 to -9</td></tr>
<tr><td>Scroll Down One Line</td><td>|ESC| C</td></tr>
<tr><td>Scroll Down One Page</td><td>|ESC| X</td></tr>
<tr><td>Scroll Up One Line</td><td>|ESC| E</td></tr>
<tr><td>Scroll Up One Page</td><td>|ESC| W</td></tr>
<tr><td>Search Down</td><td>-L</td></tr>
<tr><td>Search Up</td><td>-K</td></tr>
<tr><td>Search and Replace Down</td><td>-J</td></tr>
<tr><td>Search and Replace Up</td><td>-H</td></tr>
<tr><td>Set and Clear Tabs</td><td>|CTRL|--I</td></tr>
<tr><td>Start of Line</td><td>-,
-<</td></tr>
<tr><td>Tab</td><td>|TAB|
|CTRL|-I</td></tr>
</table>

Command	Keys
Tab Left	|CTRL|-A -A
Toggle Auto Indent Mode	|CTRL|--M
Toggle Escape Mode	|ESC|
Toggle Insert Mode	|CTRL|-E -E
Toggle Select Mode	|CTRL|--X
Toggle Wrap Mode	|CTRL|--W
Top of Screen	|CTRL|-T
Top of Screen / Page Up	|CTRL|--K -↑
Undo Delete	|CTRL|-Z -Z
Word Left	-← |CTRL|--H
Word Right	-→ |CTRL|--U

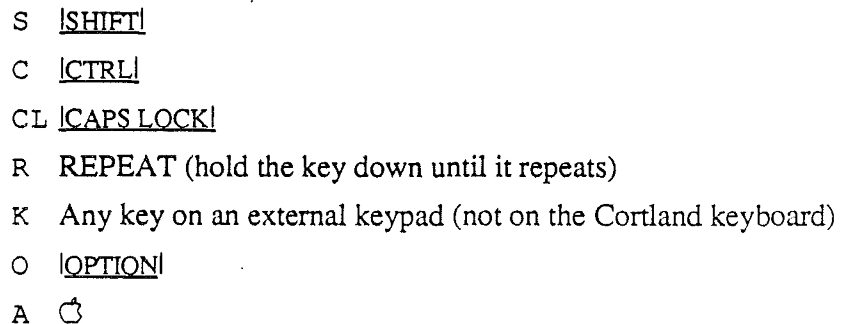
Debugger

This section lists all of the commands that you can use in the CPW Debugger.

Keystroke Modifier

If the command filter is in effect, you must hold down a keystroke-modifier key in order to pass commands on to the debugger. The keystroke modifier setting is shown in the Key field of the register subdisplay. To set the keystoke modifier, use the following command:

KEY=*keynum* — Each bit of the binary number represented by the hexadecimal number *keynum* specifies one key to be used as a keystroke modifier; set that bit to 1 to make that key a keystroke modifier. The bits are assigned as follows:



Bit:	7	6	5	4	3	2	1	0
Key:	A	O		K	R	CL	C	S
Hex Value:	80	40	20	10	08	04	02	01

Where:

- S |SHIFT|
- C |CTRL|
- CL |CAPS LOCK|
- R REPEAT (hold the key down until it repeats)
- K Any key on an external keypad (not on the Cortland keyboard)
- O |OPTION|
- A ⌘

Selecting a Display

Help Screen — ? |RETURN|.

Memory — *address* : |RETURN|.

Direct Page — D |RETURN|.

Application — OFF |RETURN| .

Display Mode — To change the display mode of your application, use the following commands (*these commands work while in single step or trace modes only*):

1 text page 1
2 text page 2

4	40-column screen
8	80-column screen
T	text mode
F	full-screen graphics
M	mixed text and graphics
L	low-resolution graphics
H	Hi-Res graphics
D	Double Hi-Res graphics
B	black-and-white (for Double-Hi-Res graphics)
C	color (for Double-Hi-Res graphics)
S	Super Hi-Res graphics

Monitor — MON |RETURN|

Master Display — In direct-page or memory display, press |ESC|.

If your application is being displayed, type ON |RETURN|.

From the Cortland Monitor, press |CTRL|-Y |RETURN| to return to the master display.

Editing the Master Display

Use the following commands to alter the contents or setup of the master display.

Display Setup

SET — Adjust stack-pointer highlight and number of instructions below highlight in disassembly subscreen.

←	Move the stack pointer up one line.
→	Move the stack pointer down one line.
↑	Move the current instruction up one line.
↓	Move the current instruction down one line.

Disassembly Subdisplay

*address*L — Disassemble the contents of memory starting at *address* and display the next 19 lines of code.

L — Disassemble the contents of memory starting at the current address and display the next 19 lines of code.

address:instruction — Assemble the instruction *instruction* and place the opcode and operand in memory at *address*. Simultaneously, the instruction is placed on the last line of the disassembly subscreen.

ASM	Clear the disassembly subdisplay, such as to prepare for entering a sequence of instructions using the *address:instruction* command.

RAM subdisplay

RETURN	Move to next address down.
↓	Move to next address down.
↑	Move to next address up.
address:	Display the contents of memory starting at *address*.
H	Display the contents of the cell as hex and ASCII.
P	Display the contents of the cell and next cell as a 2-byte address.
L	Display the contents of the cell and next two cells a a long (3-byte) address.
?	Display a help screen. Press any key except ESC to return to the RAM subdisplay.
ESC	Return to the command line.

Breakpoints Subdisplay

RETURN	Move to the next address down.
↓	Move to the next address down.
↑	Move to the next address up.
←	Move left to the address. Type in the starting address of the instruction at which you want the debugger to suspend execution.
→	Move right to the trigger value. Type in a one-byte hexadecimal number to indicate the number of times the debugger should execute this instruction before suspending execution.
DELETE	Delete the currently highlighted breakpoint and increase the number of memory protection lines by one.
?	Display a help screen. Press any key except ESC to return to the breakpoint subdisplay.
ESC	Return to command line.

The following breakpoint commands can be entered from the master-display command line:

`CLR`	Zero all breakpoints to 00/0000-00-00.
`IN`	Insert real breakpoints. **Note:** You cannot edit the breakpoint subdisplay when real breakpoints are in.
`OUT`	Remove real breakpoints.

Memory Protection Subdisplay

|RETURN|	Move to the next address down.
↓	Move to the next address down.
↑	Move to the next address up.
←	Move left to the starting address. Type in the starting address of the code-trace or code-window range.
→	Move right to the ending address. Type in the ending address of the code-trace or code-window range. Do not include a bank value; the bank must be the same as that of the starting address.
`T`	Set this line as a code-trace range.
`W`	Set this line as a code-window range.
|DELETE|	Delete the current memory-protection line and increase the number of breakpoint lines by one.
?	Display a help screen. Press any key except |ESC| to return to the memory protection subdisplay.
|ESC|	Return to the command line.

Command Line Commands

These commands are used on or entered from the debugger command line. Press Return to execute these commands.

Command-Editing Commands

These commands are used for editing commands that you are typing on the command line.

|CTRL|-E	Toggle insert/replace mode.
←	Move the cursor one character to the left.

→	Move the cursor one character to the right.
|CTRL|-D |DELETE|	Delete the character to the left of the cursor.
|CTRL|-F	Delete the character that the cursor is on.
|CTRL|-Y	Delete from the cursor position to the end of line.
|CTRL|-X |ESC|	Delete the entire line.
|CTRL|-Z	Restore the last command typed.
|RETURN|	Execute the command on the line. The entire line is sent to the command interpreter; the line is *not* truncated at the cursor position.

Setting Registers and Memory Values

`e`	Toggle the `e` flag: if it's set to 1, change it to 0; if it's set to 0, change it to 1.
`x`	Toggle the `x` flag: if it's set to 1, change it to 0; if it's set to 0, change it to 1. This command works only if e=0.
`m`	Toggle the `m` flag: if it's set to 1, change it to 0; if it's set to 0, change it to 1. This command works only if e=0.
register=value	Set the register specified by *register* to the value specified by *value*. The values for all registers are given as hexadecimal numbers, except for the processor-status bits, which can be either 1 or 0. Register names are case sensitive.
address:value	Place the hexadecimal value *value* in memory starting at *address*.
address:"string	Place values corresponding to *string* , with the high bit of each byte set, in memory starting at *address*.
address:'string	Place values corresponding to *string* with the high bit of each byte cleared in memory at *address*.
address:!character	Place a value corresponding to *character* with the high bit of the byte cleared in memory at *address*.
address:instruction	Assemble *instruction* and place the opcode and operand in memory starting at *address*.

Breakpoints

`CLR`	Zero all breakpoints to 00/0000-00-00.

`IN`	Insert real breakpoints. **Note:** You cannot edit the breakpoint subdisplay when real breakpoints are in.
`OUT`	Remove real breakpoints.

Hexdecimal—Decimal Conversion

value=	Convert *value* from hexadecimal to decimal. This command is identical to the $*value* command.
$*value*=	Convert *value* from hexadecimal to decimal. This command is identical to the *value* command.
+*value*=	Convert *value* from decimal to hexadecimal.
–*value*=	Convert *value* from decimal to hexadecimal. A negative decimal value is converted to a two-byte twos complement hexadecimal equivalent. For example, –10 = $FFF6. (Note that, if you put in $FFF6, you get 65526, not -10.)

Saving Display Configurations

`CSAVE` *pathname*	Saves the current display configuration on disk to the file specified by *pathname*.
`CLOAD` *pathname*	Restore a previously-saved display configuration from the disk file specified by *pathname*.

Printing

`P`*num*	Print the current text screen; *num* is the slot number for the printer. If you omit *num*, slot 1 is used.

Loading and Running Your Program

`LOAD` *pathname*	Load the program to debug.
`S`	Enter single-step mode at the current instruction.
address`S`	Enter single-step mode at address *address*.
`T`	Enter trace mode at the current instruction
address`T`	Enter trace mode at address *address*.

*address*G	JSL directly to code at address *address* . If you omit *address*, then the current K/PC address is used. The debugger automatically turns off the master display before executing this command.
*address*J	JML directly to code at address *address* . If you omit *address*, then the current K/PC address is used. The debugger automatically turns off the master display before executing this command.

Other Command-Line Commands

KEY=*keynum*	Each bit of the binary number represented by the hexadecimal number *keynum* specifies one key to be used as a keystroke modifier; set that bit to 1 to make that key a keystroke modifier. The bits assignments are described in the section "Keystroke Modifier" in this appendix.
PREFIX *n pathname*	Change ProDOS 16 prefix Prefix *n* to *pathname.*
V	Display the current version number and copyright of the CPW Debugger.
MON	Enter the Cortland Monitor. From the Monitor, type CTRL-Y RETURN to return to the debugger.
Q	Exit debugger.

Trace and Single-Step Mode Commands

ESC	Terminate trace or single-step mode and return to the command-line.
SPACE	Single-step one instruction.
RETURN	Start continuous tracing.
R	Trace until the next RTS, RTI, or RTL.
J	If the current instruction (the next to be executed) is a JSL, execute in real time until an RTL or RTI. If the next instruction is not a JSL, the command is ignored.
↓	Skip the next instruction.
Q	Toggle the sound on or off. If the sound is on, the speaker beeps each time an instruction is executed.
1	Change the display to text page 1. Use this command when in 40-column text mode or mixed text and graphics mode.
2	Change the display to text page 2. Use this command when in 40-column text mode or mixed text and graphics mode.

4	Change the display to a 40-column screen. Use this command when in text mode.
8	Change the display to a 80-column screen. Use this command when in text mode.
T	Change the display to text mode.
F	Change the display to full-screen graphics mode.
M	Change the display to mixed text and graphics mode.
L	Change the display to low-resolution graphics mode.
H	Change the display to high-resolution graphics mode.
D	Change the display to double-high-resolution graphics mode.
S	Change the display to super-high-resolution graphics. This is the normal Cortland display mode.
B	Change the display to black and white double-high-resolution graphics.
C	Change the display to color double high-resolution graphics.
←	Change to the slow trace rate.
→	Change to the fast trace rate.
⌘	Pause the trace until the ⌘ key is released.

LinkEd

APPEND *pathname*
Append a LinkEd source file

COPY *pathname*
Copy a LinkEd source file

EJECT
Skip to a new page if printer is on

KEEP *pathname*
Open a file for output

LIBRARY *pathname*
Search a library

LINK[/ALL] *pathname*
Link an object file

LIST ON|OFF
Control subroutine listing

OBJ *val*
Set phantom program counter

OBJEND
Turn off previous OBJ

ORG *val*
Set program counter

PRINTER ON|OFF
Control printed output

SEGMENT[/DYNAMIC] [/NUMBER=*kind*] *segname*
Start load segment

SELECT[/SCAN] *pathname* (*seg1* [,*seg2* [,...]])
Choose specific object segments

SOURCE ON|OFF
Control LinkEd source program listing

SYMBOL ON|OFF
Control symbol table output

Appendix B

Error Messages

*****This appendix will contain information about all error messages you can get when running CPW and the Debugger, when that information is available. Meanwhile, I have included the error-messages section from the Linker Preliminary Notes.*****

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 appendix lists and describes the error messages returned by the CPW Linker. They are divided into two groups: *recoverable* (the linker continues processing), and *fatal* (the linker stops). For recoverable errors, the linker also returns an error-level number as an indication of the severity of the problem that caused the error.

Recoverable Errors

When the linker detects a recoverable error, it prints

1. The name of the segment that contained the error
2. How far into the segment (in bytes) the error point lies
3. A text error message, with the error level number in brackets immediately to the right of the message

The following error level numbers 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. Reassembly will be required when the problem is corrected.

The following errors are recoverable. 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 2 as it was on pass 1.

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 [4]
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).

Address is not Zero-Page [4]
The most significant bytes of the evaluated expression were not zero, but 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).

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 [4]
A label was defined twice in the program.

Remove one of the definitions.

Evaluation Stack Overflow [2]
(a) There may be a syntax error in the expression being evaluated. (b) The expression may be too complex for the linker to evaluate.

(a) Check to see if a syntax error has also occurred; if so, correct the problem that caused that error. (b) 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 [8]
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.

Linker Version Mismatch [2]
An older version of the linker, which cannot properly link the segments, is being used.

Update the linker.

MEM Location has been Passed [4]
The linker encountered a MEM directive that tried to reserve a memory area that the linker had already passed. The linker must find a memory definition before it places code in the defined locations.

Move the MEM directive to an earlier segment, preferably the first. This error applies only to absolute code, and should therefore be rarely encountered when writing for the Cortland.

Only JSL Can Reference Dynamic Segment [8]

ORG Location has been Passed [4]
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 Cortland.

Relative Address out of Range [4]
The given destination address is too far from the current location.

Change the addressing mode or move the destination code closer.

Relocation Expression Syntax Error
Some expressions are legal in relocatable code, if they are supported by the OMF. Some expressions, such as `LAB|4+2` are not legal because the linker cannot express them in a way that would allow the loader to perform the relocation.

Segment Header MEM Directive Not Allowed [8]

Undefined Op Code [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 errors.

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

When the linker finds a fatal error, it cannot continue processing. It prints the error message, waits for a keypress, and then quits.

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.

Could not Read Sublib Directory
A ProDOS error occurred while the linker was trying to read the directory of the library disk.

This error is usually the result of a bad disk or disk drive. Put the library disk in a diferent drive, or use another disk.

Illegal Sublib Directory
The library directory pointed to by the sublib prefix does not exist, or is not a directory.

Use the `SYSGEN` directive to correct the directory name.

Input File not Found
The `.ROOT` file could not be found.

The linker expects file naming conventions to be followed. That is, when it is asked to link the file `MYPROG`, it actually looks for the file named `MYPROG.ROOT`, because `MYPROG.ROOT` is the name of the first object file created by the assembler or compiler when it is asked to assemble the source file named `MYPROG`. If the proper `.ROOT` file name is not found, this error is returned.

Check the spelling of the file name in both the `KEEP` directive and the `LINK` directive. Make sure the `.ROOT` file has the same prefix as the file specified in those commands.

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.

Out of Memory
All free memory has been used; the memory needed by the linker is not available.

This error should not occur. If it does, it is either a bug in the linker program or a Memory Manager problem.

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.

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, 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.

SLANDEROUS ACCUSATIONS

Glossary

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.**

assembler: A program that produces object modules from source files written in assembly language.

binary file format: The ProDOS 8 loadable file format, consisting of one absolute memory image along with its destination address.

binary output file: A file of absolute code that is ProDOS 16 file type $06. The system loader will not load binary files.

code segment: An object segment that contains program code. Code segments are provided for programs that differentiate between code and data segments; see the "Segment Types" section in Chapter 8.

command table

compiler: A program that produces object modules from source files written in a high-level language such as C.

CPW Linker: The linker supplied with CPW.

current language

current prefix

data code: Code that consists primarily of data.

data segment: An object segment that contains data code. Data segments are provided for programs that differentiate between code and data segments; see the "Segment Types" section in Chapter 8.

dynamic segment: A segment that can be loaded and unloaded during execution as needed. Compare with **static segment.**

emulation mode

external command

field delimiter

file number

file number cross-reference: The part of the pathname table that contains load-file numbers and pointers to their corresponding pathnames.

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:

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.

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 table) needed to load dynamic segments.

initialization segment: A segment in an initial load file that is loaded and executed first, to perform any initialization that the program may require

internal command

INTERSEG Record: A part of a relocation dictionary. It contains relocation information for external (intersegment) references.

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 a program.

language card: Memory from $D000 to $FFFF with two RAM banks in the $Dxxx space, corresponding to expansion memory on a card in a 48K Apple II or Apple II Plus.

language.command

LinkEd: The language (set of commands) recognized by the CPW Linker.

library dictionary segment: The first segment of a library file; it contains the names and locations of all the other segments in the file. The linker uses the library dictionary segment to find the segments it needs.

library file: An object file containing program 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 file may contain a **library-segment dictionary.**

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; each memory image is followed by a **relocation dictionary.**

load segment: A segment in a load file.

local symbol: A label defined only within an individual segment. Other segments cannot access the label.Compare with **global symbol.**

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.

makelib utility: A program that creates library files from object files.

memory image: A portion of a disk file or segment that can be read directly into memory.

Memory Manager: The part of the operating system that allocates blocks of memory as needed, and keeps track of which blocks of memory are available. All applications must request blocks of memory from the memory manager rather than loading data directly into a preselected memory location.

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.

moveable segment: A segment in memory that can be moved by the memory manager whenever necessary. The memory-resident version of a position-independent segment.

native mode

object file: The output from an assembler or compiler, and the input to the linker.

object module format: The general format used in object files, library files, and load files; described in the Object Module Format section of Chapter 8.

Object Segment: A segment in an object file.

OMF: Object module format.

OMF file: Any file in object module format.

operand

operator

parameter block

pathname table: 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 table is created by the linker.

pathname list: The part of the pathname table that contains the file pathnames.

pathname segment

pipline

pointer

position-independent segment: A load segment that is moveable when loaded in memory.

program code: Code that consists primarily of instructions.

Programmer's Workshop: On Cortland, a set of programs whose purpose is to facilitate the writing, translation, execution, and debugging of system programs and applications for the Cortland. Components of the Programmer's Workshop include the shell, editor, assembler, linker, debugger, and various compilers.

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.

RELOC record: A part of a relocation dictionary that contains relocation information for local (within-segment) references.

relocate: The process of modifying a file or segment at load time so that it will execute correctly at the location in memory at which it is loaded. See also **relocatable segment.**

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.**

root filename

segment body

segment header

segment number

shell call

shell-call block

static segment: A segment that is loaded at program boot time, and is not unloaded or moved during execution.

string

symbol table

relocation dictionary: A portion of a load segment that contains relocation information necessary to modify the memory image immediately preceding it. When the segment is loaded into memory, the relocation dictionary is used to patch location-dependent addresses into the code. Relocation dictionaries also contain the information necessary to transfer control to external references.

resolve

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: An individual component of an OMF file. Each file contains one or more segments.

segment jump table: A segment in a load file that contains all references to dynamic segments that will be called during execution of the program. The segment jump table is created by the linker.

shell: A program in the Cortland Programmer's Workshop that provides a command processor interface between the user and the other components of the Programmer's Workshop.

Shell Load File:

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.

Startup Load File:

static segment: A segment that is loaded only at program boot time, and is not unloaded during execution. Compare with **dynamic segment**.

symbolic reference

system loader: The part of the operating system that reads the files generated by the linker, loads them into memory, and relocates them if necessary.

text file format

token

utility

wildcard

THIS PAGE INTENTIONALLY BLANK EXCEPT FOR THIS NOTICE WHICH IS NOT BLANK BUT TAKES UP SPACE ON THE PAGE THAT YOU COULD OTHERWISE USE FOR SOMETHING ELSE