Cortland ProDOS 16 Reference

Includes System Loader

(Version 1.0)

Beta Draft

Engineering Part Number: 030-3126
8/11/86

Writer: David C. Bice
Apple Technical Publications

Changes Since Previous Draft

- The title is changed from *Cortland Operating System Reference* to *Cortland ProDOS 16 Reference*, with the subtitle *Includes System Loader*.
- All reference to the Cortland Finder has been removed from the manual.
- There are extensive changes to the System Loader functioning, data structures, and calls.
- Most changes to the ProDOS 16 parts of the manual are minor. The most significant change is in the manner of allocating direct-page/stack space.
- The glossary is now complete.
- The preface is more complete and includes space for a roadmap figure.

There are hundreds of minor changes in spelling, usage, terminology and technical interpretation throughout the manual. It is not practical or even possible to list those changes or put change bars in the margins to note them.

Schedule

12/20/85	Release of Document Design
2/21/86:	Release of Preliminary Notes
5/22/86:	Release of ProDOS/16 (version 1.0) Alpha Draft
8/11/86:	Release of Cortland ProDOS 16 (version 1.0) Beta Draft

Contents

195 Appendixes

List of Figures

List of Tables

Preface

The *Cortland ProDOS 16 Reference* is a manual for software developers, advanced programmers, and others who wish to understand the technical aspects of the Cortland operating system. In particular, this manual will be useful to you if you want to write

- a stand-alone program that will automatically run when the computer is started up
- a routine that catalogs disks, manipulates sparse files, or otherwise interacts with the Cortland file system at a basic level
- an interrupt handler
- a program that loads and runs other programs
- any program using segmented, dynamic code

The functions and calls in this manual are in assembly language format. If you are programming in assembly language, you will use the same format to access operating system features. If you are programming in a higher-level language, you will use library interface routines specific to your language. Those library routines are not described here; consult your language manual.

Road map to Cortland technical manuals

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 technical manuals are listed in Table P-1. Figure P-1 is a diagram showing the relationships among the different manuals.

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: Volume 1	How the tools work
Cortland Toolbox Reference: Volume 2	More Toolbox specifications
Cortland Programmer's Workshop	The development environment
Cortland Workshop Assembler Reference*	Using the CPW assembler
Cortland Workshop C Reference*	Using C on the Cortland
ProDOS 8 Technical Reference	ProDOS for Apple II programs
Cortland ProDOS 16 Reference	ProDOS and Loader for Cortland
Human Interface Guidelines	For all Apple computers
Apple Numerics Manual	Numerics for all Apple computers

*There is a Pocket Reference for each of these.

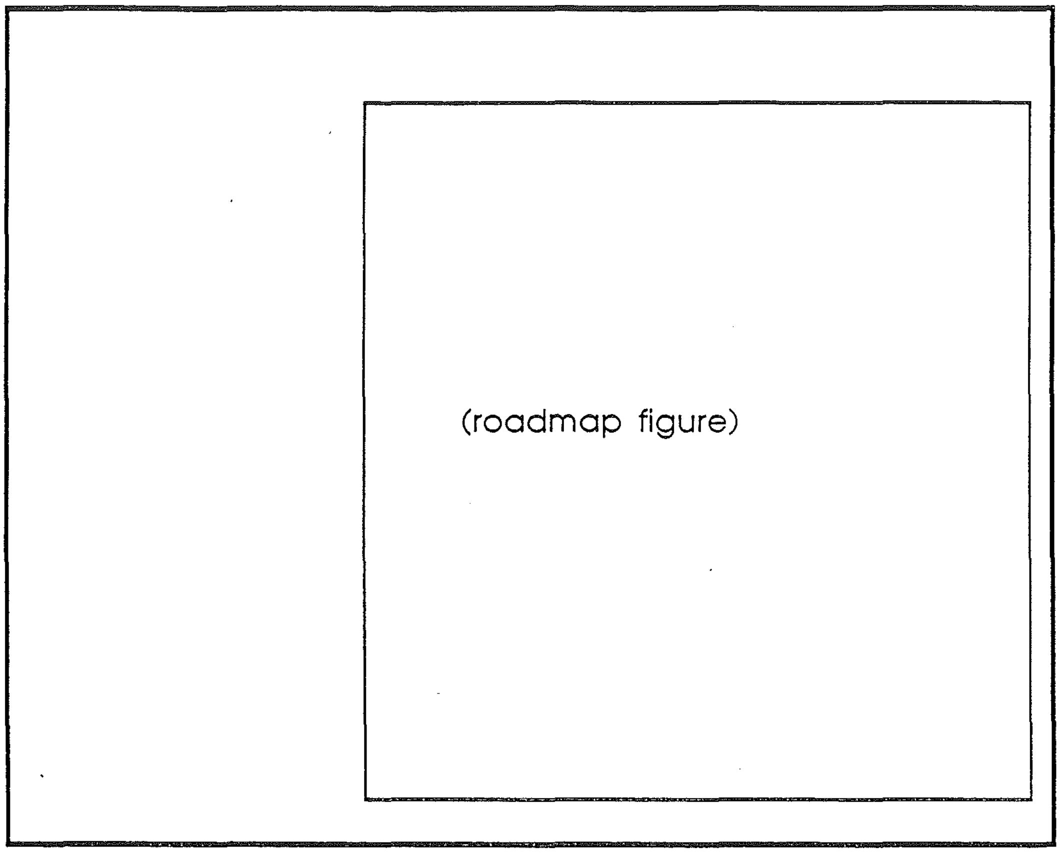


Figure P-1. Roadmap to the technical manuals

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 the technical reader understand the features of the Cortland, particularly the features that are different from other Apple computers. Having read the introductory manuals, the reader will refer to specific reference 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 point of view of the user, the *Technical Introduction* 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.

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, only a starting point for programmers writing applications for the Cortland. It introduces the routines in the Cortland Toolbox and the program environment they run under. It includes a sample **event-driven program** that demonstrates how a program uses the Toolbox and the operating system.

An **event-driven program** waits in a loop until it detects an event such as a click of the mouse button.

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.

The hardware reference manual

The *Cortland Hardware Reference* is required reading for hardware developers, and it will also be of interest to anyone else who wants to know how the machine works. Information for developers includes the mechanical and electrical specifications of all connectors, both internal and external. Information of general interest includes descriptions of the internal hardware, which provide a better understanding of the machine's features.

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 *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 the mouse. The *Firmware Reference* also describes the Monitor, a low-level programming and debugging aid for assembly-language programs.

The Toolbox manuals

Like the Macintosh, the Cortland has a built-in Toolbox. 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.

Of course, you don't have to use the Toolbox at all. If you only want to write simple programs that don't use the mouse, or windows, or menus, or other parts of the **desktop user interface**, then you can get along without the Toolbox. However, if you are developing an application that uses the desktop interface, or if you want to use the Super Hi-Res graphics display, you'll find the Toolbox to be indispensable.

In applications that use the **desktop user interface,** commands appear as options in pull-down menus, and material being worked on appears in rectangular areas of the screen called windows. The user selects commands or other material by using the mouse to move a pointer around on the screen.

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 *Cortland Programmer's Workshop Reference* includes information about the parts of the workshop that all developers will use, regardless which programming language they use: the shell, the editor, the linker, the debugger, and the utilities. The manual also tells how to write other programs, such as custom utilities and compilers, to run under the CPW Shell.

The *Programmer'sWorkshop Reference* describes the way you use the workshop to create an application and includes a sample program to show how this is done.

Programming-language manuals

Apple is currently providing a 65C816 assembler and a C compiler. Other compilers can be used with the workshop, provided that they follow the standards defined in the *Cortland Programmer's Workshop Reference.*

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 Workshop Assembler Reference* and the *Cortland Workshop C Reference.*

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 8 Reference* and *ProDOS 16 Reference.* ProDOS 16 uses the full power of the Cortland and is not compatible with earlier Apple IIs. The ProDOS 16 manual includes information about the System Loader, which works closely with ProDOS 16, and about the Finder, which is the user interface to the operating system. If you are writing programs for the Cortland, whether as an application programmer or a system programmer, you are almost certain to need the *ProDOS 16 Reference.*

ProDOS 8, previously just called *ProDOS*, is compatible with the models of Apple II that use 8-bit CPUs. As a developer of Cortland programs, you need to use ProDOS 8 only if you are developing programs to run on 8-bit Apple II's as well as on the Cortland.

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.* If you develop programs for any Apple computer, you should know about those manuals.

The *Human Interface Guidlines* describes Apple's standards for the human interface of programs that run on Apple computers. If you are writing an application for the Cortland, you should be familiar with the contents of this manual.

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's 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 is 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 65816 is available through the Apple Programmer's and Developer's Association, administered by the A.P.P.L.E. cooperative in Renton, Washington.

How to use this manual

The *Cortland ProDOS 16 Reference* is both a reference manual and a learning tool. It is divided into several parts, to help you quickly find what you need.

- Part I describes ProDOS 16, the central part of Cortland's operating system
- Part II lists and explains the ProDOS 16 operating system calls
- Part III describes the System Loader and lists all loader calls
- The final part consists of appendixes, a glossary, and an index

The first chapter in each part is introductory; read it first if you are not already familiar with the subject. The remaining chapters are primarily for reference, and need not be read in any particular order. A quick reference card tabulates calls, formats, and features for both

ProDOS 16 and the System Loader. The ProDOS 16 *Exerciser*, on a diskette included with the manual, provides a way to practice making ProDOS 16 calls before actually coding them.

This manual does not explain 65816 assembly language. Refer to *Cortland Programmer's Workshop Assembler Reference* for information on Cortland assembly language programming.

This manual does not give a detailed description of ProDOS 8, the Apple II operating system from which ProDOS 16 was derived. For synopses of the differences between ProDOS 8 and ProDOS 16, see Chapter 1 and Appendix F of this manual. For more detailed information, see *ProDOS 8 Reference* or *ProDOS Technical Reference Manual..*

Other materials you'll need

Hardware and software

To use the products described in this manual, you will need a Cortland with at least one external disk drive. ProDOS 16 and the System Loader require only the minimum memory configuration (256K RAM), although Cortland Programmer's Workshop and many application programs may require more memory.

You will also need a Cortland system disk. A system disk contains ProDOS 16, ProDOS 8, the System Loader, and other system software necessary for proper functioning of the computer. A system disk may also contain application programs.

If you wish to practice making ProDOS 16 operating system calls you will need the ProDOS 16 *Exerciser*, a program on the diskette included with this manual.

Publications

This manual is the only reference for ProDOS 16 and the System Loader. Other useful references are described under "Roadmap to Cortland Technical Manuals," in this preface.

Notations and conventions

To help make the manual more understandable, the following conventions and definitions are used throughout.

Terminology:

This manual may define certain terms, such as Apple II and ProDOS, slightly differently than what you are used to. Please note:

Apple II: A general reference to the Apple II family of computers, especially those that may use ProDOS 8 or ProDOS 16 as an operating system. It includes the 64k Apple II Plus, the Apple IIc, the Apple IIe, and the Cortland.

8-bit Apple II: Any Apple II computer that is *not* a Cortland. Since previous members of the Apple II family share many characteristics, it is useful to distinguish them as a group from the Cortland.

ProDOS: A general term describing the family of operating systems developed for Apple II computers. It includes both ProDOS 8 and ProDOS 16; it does not include DOS 3.3 or SOS.

ProDOS 8: The existing, 8-bit ProDOS operating system (through version 1.2). In previous Apple II documentation, ProDOS 8 is called simply *ProDOS*.

ProDOS 16: A 16-bit operating system developed for the Cortland computer. It is the system described in this manual.

Typographic conventions

Each new term introduced in this manual is printed first in **boldface.** That lets you know that the term has not been defined earlier, and also indicates that there is an entry for it in the glossary.

Throughout the manual, assembly language labels and entry points that appear in text passages are printed in a special typeface (for example, `ref_num` and `GET_ENTRY`). Function names that are English language terms are printed with initial caps (for example, Load Segment By Number). When the name of a label or variable is used to mean the *value of* that variable, the word is printed in italics (for example, "if *ref_num* is nonzero...").

Watch for these

The following words mark special messages to you:

Note: Text set off in this manner—with a word or phrase such as **Note**—presents sidelights or interesting points of information.

Important: Text set off in this manner—with the word **Important:**—presents important information or instructions.

Warning: Text set off in this manner—with the word **Warning:**—indicates potential problems or disaster.

Part I

How ProDOS 16 Works

This part of the manual describes version 1.0 of ProDOS 16. ProDOS 16 is the core of the Cortland operating system; it provides file management and input/output capabilities, and controls certain other aspects of the Cortland operating environment.

Chapter 1

About ProDOS 16

This chapter introduces ProDOS 16. It gives background information on the development of ProDOS 16, followed by an overview of ProDOS 16 in relation to the Cortland. A brief comparison of ProDOS 16 with ProDOS 8, its closest relative in the Apple II world, is followed by a reference list of the most pertinent ProDOS 16 features.

The chapter's organization roughly parallels that of Part I as a whole. Each section refers you to the appropriate chapter for more information on each aspect of ProDOS 16.

Background

The Cortland is the latest Apple II computer; its microprocessor, the 65816, is a successor to the Apple IIs' 6502 and functions in both 8-bit (6502 **emulation**) mode and 16-bit (**native**) mode (see *Technical Introduction to the Cortland*). In accordance with the design philosophy governing all Apple II family products, the Cortland is compatible with existing Apple II software; most presently available Apple II, Apple IIc, and Apple IIe applications will run unchanged on the Cortland.

To retain this compatibility while adding new features, the Cortland requires two separate operating systems: ProDOS 8 and ProDOS 16. ProDOS 8 is the current operating system for the Apple II family; it functions in Cortland emulation mode. ProDOS 16 is a newly developed system, written specifically for the Cortland; it functions in both emulation and native mode. The two operating systems automatically switch to match the system file that is loaded, so the user need not worry about which operating system is active at any one time.

ProDOS 8 on the Cortland functions identically to ProDOS 8 on other Apple II computers, except for some minor differences noted in Appendix B. For a complete description of ProDOS 8, see the *ProDOS 8 Reference.*

What Is ProDOS 16?

ProDOS 16 is the central part, or kernel, of the Cortland operating system. Although other software components (such as the System Loader described in this manual) may be thought of as parts of the overall operating system, ProDOS 16 is the key component. It manages the creation and modification of **files**. It accesses the **disk devices** through which the files are stored and retrieved. It dispatches interrupt signals to **interrupt handlers**. It also controls certain aspects of the Cortland **operating environment**, such as pathname prefixes and procedures for quitting programs and starting new ones.

Programming levels in the Cortland

Figure 1-1 is a simplified logical diagram of the Cortland, from a programmer's point of view. Boxes representing parts of the system form a vertical hierarchy; arrows between the boxes show the flow of control or execution from one level to the next. At the highest level is the programmer or user; he directly manipulates the execution of the application program that runs on the machine. Theapplication, in turn, interacts directly with the next lower level of software—the operating system. The operating system interacts with the very lowest level of software in the machine: the built-in firmware and toolbox routines. Those routines directly manipulate the switches, registers, and input/output devices that constitute the computer's hardware.

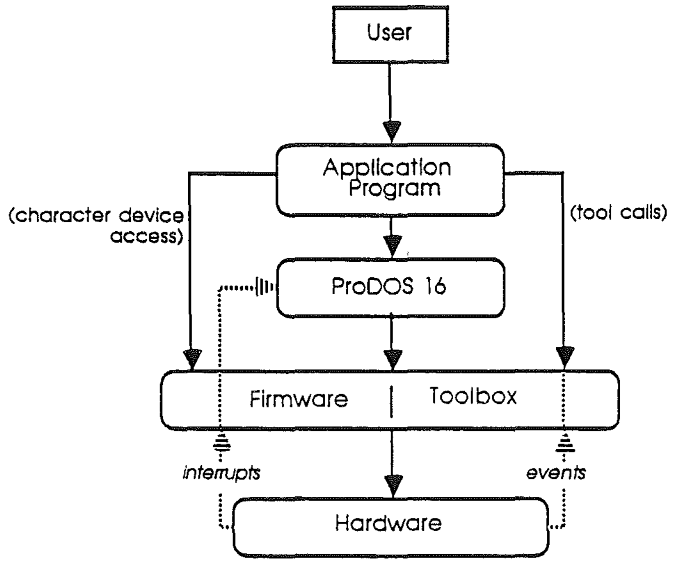


Figure 1-1. Programming levels in the Cortland.

This hierarchical view shows that the operating system is an intermediary between the application program and the computer hardware. A program need not know the details of individual hardware devices it accesses; instead, it makes operating system **calls**. The operating system then translates those calls into the proper instructions for whatever devices are connected to the system.

The lowest software level, between the operating system and hardware, is extensively developed in the Cortland. It consists of two parts: the **firmware**, a collection of traditional ROM-based routines for performing such tasks as character I/O, interrupt handling, and memory manipulation; and the **toolbox**, a large set of assembly-language routines and macros useful to all levels of software. As the arrows on Figure 1-1 show, ProDOS 16 accesses the firmware/tools level of the Cortland directly, but so do application programs. In other words, for tool calls and certain types of I/O, applications *bypass* ProDOS 16 and interact directly with low-level system software.

The arrows pointing *upward* along the diagram show a counterflow of information, in which lower levels in the machine notify higher levels of important hardware conditions. **Interrupts** from hardware devices are handled both by firmware and by ProDOS 16; **events** are similar to interrupts but are handled by applications through tool calls.

Disks, volumes, and files

ProDOS 16 communicates with several different types of disk drives, but the type of drive and its physical location (slot or port number) need not be known to a program that wants to access that drive. Instead, a program makes calls to ProDOS 16, identifying the disk it wants to access by its *volume name* or *device name.*

Information on a volume is divided into files. A **file** is an ordered collection of bytes that has several attributes, including a name and a file type. Files are either **standard files** (containing any type of code or data) or **directory files** (containing the names and disk locations of other files). When a disk is initially formatted, its **volume directory** file is created; the volume directory has the same name as the volume itself.

ProDOS 16 supports a **hierarchical file system**, meaning that volume directories can contain the names of either files or other directories, called **subdirectories**; subdirectories in turn can contain the names of files or other subdirectories. In a hierarchical file system, a file is identified by its **pathname**, a sequence of file names starting with the volume directory and ending with the name of the file. Figure 1-2 shows the relationships among files in a hierarchical file system.

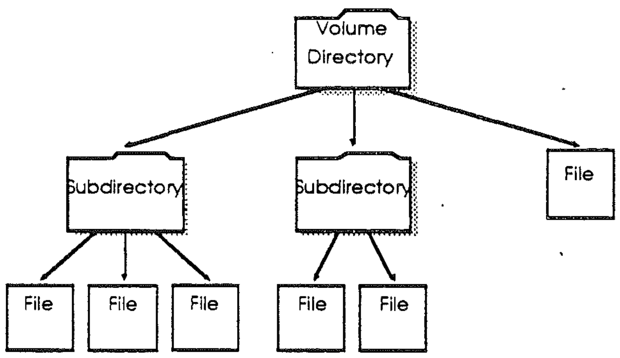


Figure 1-2. Example of a hierarchical file structure.

See Chapter 2 and Appendix A for detailed information on ProDOS 16's file structure, organization and formats.

Memory use

ProDOS 16 and application programs on the Cortland are relieved of most memory management tasks. The Memory Manager, a Cortland Tool, allocates all memory space, keeps track of available memory, and frees memory no longer needed by programs. If a program needs to allocate some memory space, it requests the space through a call to the

Memory Manager. If a program makes a ProDOS 16 call that results in memory allocation, ProDOS 16 requests the space from the Memory Manager and allocates it to the program.

The Memory Manager is described further in Chapter 3 of this manual, and in *Cortland Toolbox Reference.*

External devices

ProDOS 16 communicates only with **block devices**, such as disks. Programs that wish to access **character devices** such as printers and communication ports must do so directly, either through the device firmware or through Cortland Toolbox routines written for those devices. See *Cortland Firmware Reference* and *Cortland Toolbox Reference*.

Certain devices generate **interrupts** to tell the computer that the device needs attention. ProDOS 16 is able to handle up to 16 interrupting devices. You may place an interrupt handling routine into service through a ProDOS 16 call; your routine will then be called each time an interrupt occurs. If you install more than one routine, the routines will be polled in the order in which they were installed.

You may also remove an interrupt routine with a ProDOS 16 call. In writing, installing, and removing interrupt handling routines, be sure to follow the conventions and requirements given in Chapter 8, "Adding Routines to ProDOS 16."

ProDOS 16 and ProDOS 8

ProDOS 16, although derived from ProDOS 8, adds several capabilities to support the new features and operating modes of the Cortland. They include the following:

- The 65816 microprocessor functions in both 8-bit (emulation) and 16-bit (native) execution modes. ProDOS 16 is designed to accept system calls from applications running in either 8-bit or 16-bit mode; ProDOS 8 accepts system calls from applications running in 8-bit mode only.
- The Cortland has 256 kilobytes (256K) of RAM memory, presently expandable to 4 megabytes (4 Mb); its maximum potential memory space is as much as 16Mb. ProDOS 16 has the ability to accept system calls from anywhere in that 16Mb memory space (addresses up to $FFFFFF), and those calls can manipulate data anywhere in memory. Under ProDOS 8, system calls can be made from memory addresses below $FFFF only—the lowest 64K of memory.
- ProDOS 16 relies on a sophisticated memory management system (see Chapter 3), instead of the simple global page bit map used by ProDOS 8.
- Applications under ProDOS 16 must make calls to allocate memory or to access system global variables, such as date and time, system level, and I/O buffer addresses. ProDOS 8 maintains that information in the system global page in memory bank $00, but under ProDOS 16 the global page cannot be supported.
- ProDOS 16 also provides several programming conveniences not available under ProDOS 8, including named devices and multiple, user-definable file prefixes.

Upward compatibility

In a strict sense, ProDOS 16 is not upwardly compatible from ProDOS 8. Programs written to function under ProDOS 8 on an Apple II will not run on the Cortland, *under ProDOS 16*, without some modification. Conceptually, however, ProDOS 16 *is* upwardly compatible from ProDOS 8, in at least two ways:

1. The two operating systems are themselves similar in structure:
   - The set of ProDOS 16 system calls is a superset of the ProDOS 8 calls; for (almost) every ProDOS 8 system call, there is a functionally equivalent ProDOS 16 call, usually with the same name.
   - The calls are made in a nearly identical ways in both ProDOS systems, and the parameter blocks for passing values to functions are laid out similarly.
   - ProDOS 16 uses exactly the same file system as ProDOS 8. It can read from and write to any disk volume produced by ProDOS 8. This file system similarity applies to the disk-resident file and volume structure as well as the logical file and volume structure.
2. Both operating systems are included with the Cortland. Most applications written for ProDOS 8 on the Apple II family of computers will run unchanged on the Cortland—not under ProDOS 16, but under ProDOS 8.

Thus, even though the operating systems are not completely compatible, the Cortland computer is completely upwardly compatible from other Apple II computers.You never need be concerned with which operating system is functioning—if you run an Apple II application, ProDOS 8 is automatically loaded; if you run a Cortland application, ProDOS 16 is automatically loaded. Chapter 5 explains the details of how this is accomplished.

Downward compatibility

ProDOS 16 is not downwardly compatible to ProDOS 8. Applications written for ProDOS 16 will not run on the Apple II, IIc, or IIe. The extra memory needed by Cortland applications and the additional instructions recognized by the 65816 microprocessor make applications written for ProDOS 16 incompatible with the other Apple II computers.

Eliminated ProDOS 8 system calls

As mentioned under "Upward Compatibility," most ProDOS 8 calls have functionally exact equivalents in ProDOS 16. However, some ProDOS 8 calls do not appear in ProDOS 16 because they are unnecessary. The eliminated calls are

`RENAME`	The ProDOS 16 call `CHANGE_PATH` performs the same function.
`GET_TIME`	Under ProDOS 16, the time and date are obtained through a call to the Miscellaneous Toolset (see *Cortland Toolbox Reference*).
`SET_BUF`	Under ProDOS 16, the Memory Manager, rather than the application, allocates file I/O buffers.

`GET_BUF`	This call is unnecessary under ProDOS 16 because the `OPEN` call returns a handle to the file's I/O buffer.
`ONLINE`	This call is replaced in ProDOS 16 by the `VOLUME` call.

Other features

Like ProDOS 8, ProDOS 16 supports block devices only. It does not support I/O operations for the built-in serial ports, mouse, Apple DeskTop Bus, sound generation system, or any other nonblock device. Applications must access these devices through the device firmware or the Cortland Toolbox.

ProDOS 8 and ProDOS 16 have identical file structures. Each can read and execute the other's files, with the following exceptions:

- ProDOS 8 does not recognize ProDOS 16 load files (types $B0 - $BF)
- ProDOS 16 does not recognize ProDOS 8 system files (type $FF) or binary files (type $06)

The default operating system on the Cortland (after a cold or warm restart) can be either ProDOS 8 or ProDOS 16, depending on the organization of files on the startup disk. See "System Startup" in Chapter 5.

Running under ProDOS 8 does not disable memory beyond the addresses ProDOS 8 can reach, nor does it disable any other advanced Cortland features; all system resources are available, even though the application itself may make use of only the "ProDOS 8—Apple II" portion.

Summary of ProDOS 16 features

The following lists summarize the principal features that distinguish ProDOS 16 from other microcomputer operating systems. Refer to the glossary and to appropriate chapters for definitions and explanations of terms that may be unfamiliar to you.

In general, ProDOS 16...

- is a single-task operating system
- supports a heirarchical, tree-structured file system
- allows device-independent I/O for block devices

ProDOS 16 system calls...

- use the JSL instruction and a parameter block
- return error status in the A and P registers
- preserve all other CPU registers
- can be made from 65816 native mode or 6502 emulation mode

- can be made from anywhere in memory
- can access parameter blocks anywhere in memory
- can use pointers that point anywhere in memory
- can transfer data anywhere in memory

The ProDOS 16 file management system...

- uses a hierarchical file structure
- supports pathname prefixes (8 allowed)
- allows byte-oriented access to both directory files and data files
- allocates files dynamically and noncontiguously on block devices
- supports sparse files
- provides buffers automatically
- supports access attributes that enable/disable
  - reading
  - writing
  - renaming
  - destroying
  - backup
- assigns a system level to open files
- automatically marks files with date and time
- allows up to 14 volumes on line
- allows up to 8 open files
- uses a 512-byte block size
- allows volume sizes up to 32 megabytes
- allows data file sizes up to 16 megabytes
- allows 64 characters per pathname
- allows 64-character prefixes
- allows 15 characters per volume name
- allows 15 characters per file name

The ProDOS 16 device management system...

- supports the ProDOS 8 block device protocol
- names each block device
- allows 15 characters per device name
- allows 14 devices on line simulataneously

The ProDOS 16 interrupt management system...

- receives hardware interrupts not handled by firmware
- dispatches interrupts to user-provided interrupt handlers
- allows installation of up to 16 interrupt handlers

For memory management, ProDOS 16...

- dynamically allocates and releases system buffers (through the Memory Manager)
- can directly access up to 2^{24} bytes (16 megabytes) of memory
- can run with a minimum of 256K memory

In addition, ProDOS 16...

- provides a `QUIT` call to cleanly exit one program and start another, with the option of returning later to the quitting program

Chapter 2

ProDOS 16 Files

The largest part of ProDOS 16 is its file management system. This chapter explains how files are named, how they are created and used, and a little about how they are organized on disks. It discusses ProDOS 16 *file access* and *file housekeeping* calls.

Using files

Filenames

A ProDOS 16 **filename** or **volume name** is up to 15 characters long. It may contain uppercase letters (A-Z), digits (0-9), and periods (.), and it must begin with a letter. Lowercase letters are automatically converted to uppercase. A filename must be unique within its directory. Some examples are

```
MEMOS
CHAP11
MY.PROGRAM
```

Pathnames

A ProDOS 16 **pathname** is a series of filenames, each preceded by a slash (/). The first filename in a pathname is the name of a volume directory. Successive filenames indicate the path, from the volume directory to the file, that ProDOS 16 must follow to find a particular file. The maximum length for a pathname is 64 characters, including slashes. Examples are

```
/DISK86/CHARTS/SALES.JUN
/DISK86/MY.PROGRAM
/DISK86/MEMOS/CHAP11
```

All calls that require you to name a file will accept either a full pathname or a **partial pathname.** A partial pathname is a portion of a pathname; you can tell that it is not a full pathname because it doesn't begin with a slash and a volume name. The maximum length for a partial pathname is 64 characters, including slashes.

These partial pathnames are all derived from the sample pathnames above:

```
SALES.JUN
MY.PROGRAM
MEMOS/CHAP11
CHAP11
```

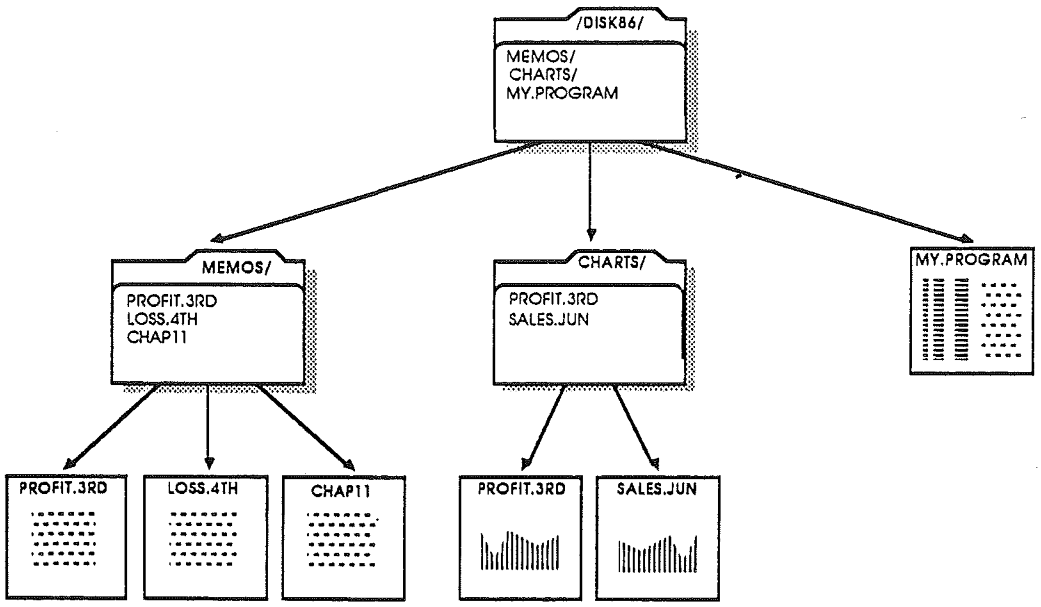


Figure 2-1. Example of a ProDOS 16 file structure

ProDOS 16 automatically adds a **prefix** to the front of partial pathnames to form full pathnames. A prefix is a pathname that indicates a directory; several prefixes are stored internally by ProDOS 16.

For the partial pathnames listed above to indicate the proper files, their prefixes should be set to

```
/DISK86/CHARTS/
/DISK86/
/DISK86/
/DISK86/MEMOS/
```

respectively. The slashes at the end of these prefixes are optional; however, they are convenient reminders that prefixes indicate directory files.

The maximum length for a prefix is 64 characters. The minimum length for a prefix is zero characters, known as a **null prefix**. You set and read prefixes using the calls `SET_PREFIX` and `GET_PREFIX`. The 64-character limits for the prefix and partial pathname combine to create a maximum pathname of 128 characters.

ProDOS 16 allows you to set more than one prefix, and then refer to each prefix by code numbers. When, as in the above examples, no particular prefix number is specified, ProDOS 16 adds the default **system prefix** to the partial pathname you provide. See Chapter 5 for a more complete explanation and examples.

Figure 2-1 illustrates a hypothetical directory structure; it contains all the files mentioned above. Note that, even though there are two files named `PROFIT.3RD` in the volume directory `/DISK.86/`, they are easily distinguished because they are in different subdirectories (`MEMOS/` and `CHARTS/`). That is why a full pathname is necessary to completely specify a file.

Creating files

A file is placed on a disk by the `CREATE` call. When you create a file, you assign it the following properties:

- A **pathname.** This pathname is a unique path by which the file can be identified and accessed. This pathname must place the file within an existing directory.
- An **access byte.** The value of this byte determines whether or not the file can be written to, read from, destroyed, or renamed.
- A **file type.** This byte indicates to other applications the type of information to be stored in the file. It does not affect, in any way, the contents of the file.
- A **storage type.** This byte determines the physical format of the file on the disk. There are only two different formats: one is used for directory files, the other for non-directory files.

When you create a file, the properties listed above are placed on the disk, in a format as shown in Appendix A. Once a file has been created, it remains on the disk until it is deleted (using the `DESTROY` call).

To check what the properties for a given file are, use the `GET_FILE_INFO` call. To change the file's name, use the `CHANGE_PATH` call. To alter the other properties, use the `SET_FILE_INFO` call.

Opening files

Before you can read information from or write information to a file, you must use the `OPEN` call to open the file for access. When you open a file you specify it by pathname. The pathname you give must indicate a previously created file; the file must be on a disk mounted in a disk drive.

The `OPEN` call returns a reference number (*ref_num*) and the location of a buffer (*io_buffer*) to be used for transferring data to and from the file. All subsequent references to the open file must use its reference number. The file remains open until you use the `CLOSE` call.

Each open file's I/O buffer is used by the system the entire time the file is open. Thus it is wise to keep as few files open as possible. ProDOS 16 allows a maximum of 8 open files at a time.

When you open a file, some of the file's characteristics are placed into a region of memory called a **file control block**. Several of these characteristics—the location in memory of the file's buffer, a pointer to the end of the file (the EOF), and a pointer to the current position in the file (the file's MARK)—are accessible to applications via ProDOS 16 calls, and may be changed while the file is open.

It is important to be aware of the differences between a file on the disk and an open file in memory. Although some of the file's characteristics and some of its data may be in memory at any given time, the file itself still resides on the disk. This allows ProDOS 16 to manipulate files that are much larger than the computer's memory capacity. As an application writes to the file and changes its characteristics, new data and characteristics are written to the disk.

The EOF and MARK

To aid reading from and writing to files, each open file has one pointer indicating the end of the file (the EOF), and another defining the current position in the file (the MARK). ProDOS 16 moves both EOF and MARK automatically when necessary, but an application program can also move them independently of ProDOS 16.

The EOF is the number of readable bytes in the file. Since the first byte in a file has number 0, the EOF, when treated as a pointer, points one position past the last character in the file.

When a file is opened, the MARK is set to indicate the first byte in the file. It is automatically moved forward one byte for each byte written to or read from the file. The MARK, then, always indicates the next byte to be read from the file, or the next byte position in which to write new data. It cannot exceed the EOF.

If during a write operation the MARK meets the EOF, both the MARK and the EOF are moved forward one position for every additional byte written to the file. Thus, adding bytes to the end of the file automatically advances the EOF to accommodate the new information. Figure 2-2 illustrates the relationship between the MARK and the EOF.

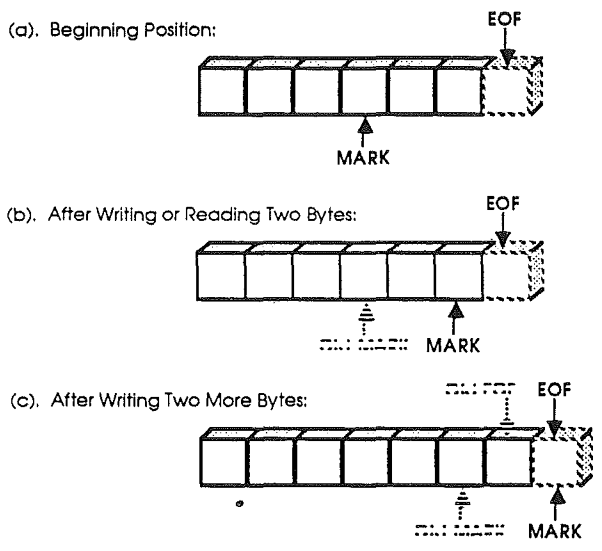

(a). Beginning Position:
EOF
MARK
(b). After Writing or Reading Two Bytes:
EOF
MARK
(c). After Writing Two More Bytes:
EOF
MARK

Figure 2-2. Automatic movement of EOF and MARK

An application can place the EOF anywhere, from the current MARK position to the maximum possible byte position. The MARK can be placed anywhere from the first byte in the file to the EOF. These two functions can be accomplished using the `SET_EOF` and `SET_MARK` calls. The current values of the EOF and the MARK can be determined using the `GET_EOF` and `GET_MARK` calls.

Reading and writing files

`READ` and `WRITE` calls to ProDOS 16 transfer data between memory and a file. For both calls, the application must specify three things:

- The reference number of the file (assigned when the file was opened).
- The location in memory of a buffer (`data_buffer`) that contains, or is to contain, the transferred data. Note that this cannot be the same buffer (`io_buffer`) whose location was returned when the file was opened.
- The number of bytes to be transferred.

When the request has been carried out, ProDOS 16 passes back to the application the number of bytes that it actually transferred.

A read or write request starts at the current MARK, and continues until the requested number of bytes has been transferred (or, on a read, until the end of file has been reached). Read requests can also terminate when a specified character is read. To turn on this feature and set the character(s) on which reads terminate, use the `NEWLINE` call. It is typically used for reading lines of text that are terminated by carriage returns.

By the Way: Neither a READ nor a WRITE call necessarily causes a disk access. ProDOS's I/O buffer for each open file can hold one block (512 bytes) of data; it is only when a read or write crosses a block boundary that a disk access occurs.

Closing and flushing files

When you finish reading from or writing to a file, you must use the CLOSE call to close the file. When you use this call, you specify only the reference number of the file (assigned when the file was opened).

CLOSE writes any unwritten data from the file's I/O buffer to the file, and it updates the file's size in the directory, if necessary. Then it frees the 1024-byte buffer space for other uses and releases the file's reference number and file control block. To access the file once again, you have to reopen it.

Information in the file's directory, such as the file's size, is normally updated only when the file is closed. If the user were to press Control-Reset (typically halting the current program) while a file is open, data written to the file since it was opened could be lost, and the integrity of the disk could be damaged. This can be prevented by using the FLUSH call.

FLUSH, like CLOSE, writes any unwritten data from the file's I/O buffer to the file, and updates the file's size in the directory. However, it keeps the file's buffer space and reference number active, and allows continued access to the file. In other words, the file stays open. If the user presses Control-Reset while an open but flushed file is in memory, there is no loss of data and no damage to the disk.

Both the CLOSE and FLUSH calls, when used with a reference number of 0, normally cause all open files to be closed or flushed. Specific groups of files can be closed or flushed using the *system file level* (see next).

File levels

When a file is opened, it is assigned a level, according to the value of a specific byte in memory (the **system file level**). If the file level is never changed, the CLOSE and FLUSH calls, when used with a reference number of 0, cause all open files to be closed or flushed. But if the level has been changed since the first file was opened, only the files having a file level greater than or equal to the current system level are closed or flushed.

The system file level feature may be used, for example, by a controlling program such as a BASIC interpreter to implement an EXEC command:

1. The interpreter opens an EXEC program file when the level is $0.

2. The interpreter then sets the level to, say, $80.

3. The EXEC program opens whatever files it needs.

4. The EXEC program executes a BASIC CLOSE command, to close all the files it has opened. All files at or above level 7 are closed, but the EXEC file itself remains open.

You assign a value to the system file level with a SET_LEVEL call; you obtain the current value by making a GET_LEVEL call.

File format and organization

This portion of the chapter describes in general terms the organization of files on a disk. For more detailed information, see Appendix A.

In general, *structure* refers in this manual to the hierarchical relationships among files—directories, subdirectories, and files. *Format* refers to the arrangement of information (such as headers, pointers and data) within a file. *Organization* refers to the manner in which a single file is stored on disk, in terms of individual 512-byte **blocks**. The three concepts are separate but interrelated. For example, because of ProDOS 16's hierarchical file *structure*, part of the *format* of a directory file includes pointers to the files within that directory. Also, because files are *organized* as noncontiguous blocks on disk, part of the *format* of every file larger than one block includes pointers to other blocks.

Directory files and standard files

Every ProDOS 16 file is a named, ordered sequence of bytes that can be read from, and to which the rules of MARK and EOF apply. However, there are two types of files: **directory files** and **standard files**. Directory files are special files that describe and point to other files on the disk. They may be read from, but not written to (except by ProDOS 16). All nondirectory files are standard files. They may be read from and written to.

A directory file contains a number of similar elements, called **entries**. The first entry in a directory file is the header entry: it holds the name and other properties (such as the number of files stored in that directory) of the directory file. Each subsequent entry in the file describes and points to some other file on the disk. Figure 2-3 shows the format of a directory file.

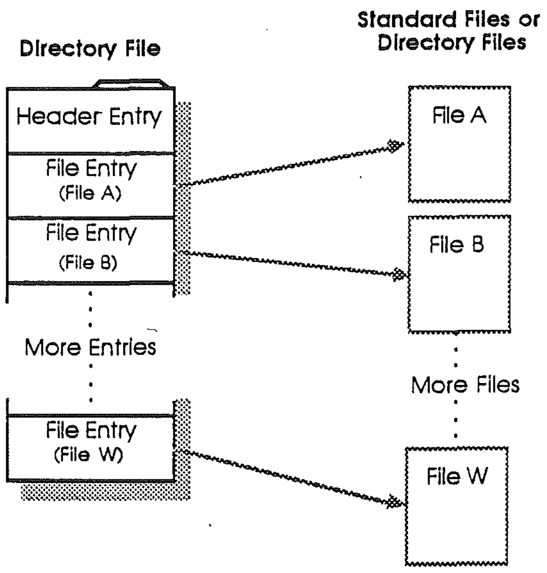


Figure 2-3. Directory file format

The files described and pointed to by the entries in a directory file can be standard files or other directory files.

An application does not need to know the details of directory format to access files with known names. Only operations on unknown files (such as listing the files in a directory) require the application to examine a directory's entries. For such tasks, refer to Appendix A.

Standard files have no such predefined internal format: the arrangement of the data depends on the specific file type.

File organization

Because directory files are generally smaller than standard files, and because they are sequentially accessed, ProDOS 16 uses a simpler form of storage for directory files than it does for standard files. Both types of files are stored as a set of 512-byte blocks, but the way in which the blocks are arranged on the disk differs.

A directory file is a linked list of blocks: each block in a directory file contains a pointer to the next block in the directory file as well as a pointer to the previous block in the directory. Figure 2-4 illustrates this organization.

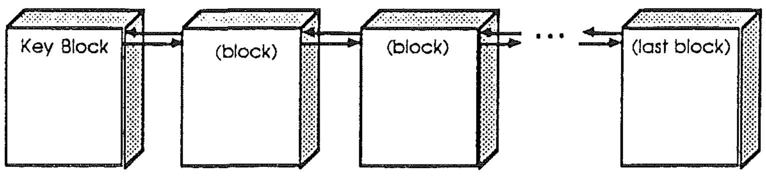


Figure 2-4. Block organization of a directory file

Data files, on the other hand, are often quite large, and their contents may be randomly accessed. It would be very slow to access such large files if they were organized sequentially. Instead, ProDOS 16 stores standard files using a **tree organization**. The largest possible standard file has a **master index block** that points to 128 **index blocks**. Each index block points to 256 **data blocks** and each data block can hold 512 bytes of data. The block organization of the largest possible standard file is shown in Figure 2-5.

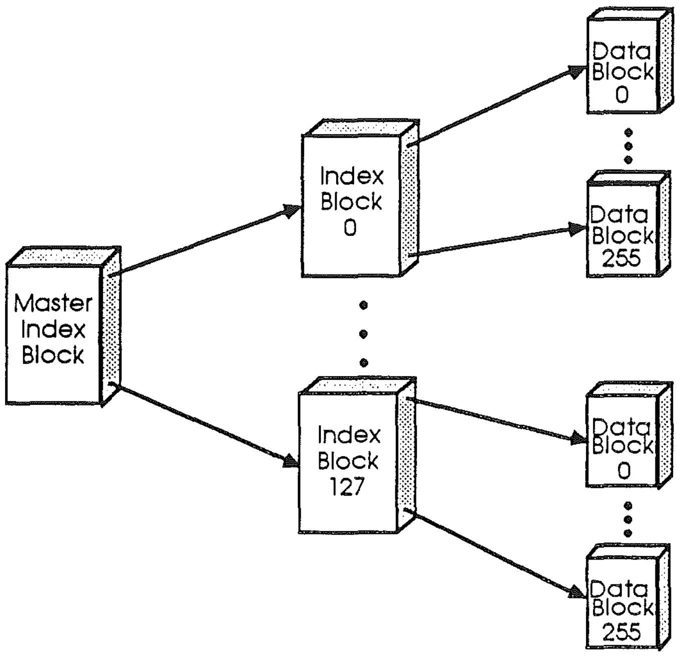


Figure 2-5. Block organization of a standard file

Most standard files do not have this exact organization. ProDOS 16 only writes a subset of this form to the file, depending on the amount of data written. This technique produces three distinct forms of standard file: seedling, sapling, and tree files. All three are explained in Appendix A.

Sparse files

In most instances a program writes data sequentially into a file. But by writing data, moving the EOF and MARK, and then writing more data, a program can also write nonsequential data to a file. For example, a program can open a file, write a few characters of data, and then move the EOF and MARK (thereby making the file bigger) by an arbitrary amount before writing a few more bytes of data. Only those blocks that contain nonzero information are actually allocated for the file, so it may take up as few as three blocks on the disk (a total of 1536 bytes). However, as many bytes as are specified by the value of

EOF (up to 16 megabytes) can potentially be read from it. Such files are known as **sparse files.** Sparse files are explained in more detail in Appendix A.

> **Important:** In transferring sparse files, the fact that more data can be read from the file than actually resides on the disk can cause a problem. Suppose that you were trying to copy a sparse file from one disk to another. If you were to read data from one file and write it to another, the new file would be much larger than the original because data that is not actually on the disk can be read from the file. Thus if your application is going to transfer sparse files, you must use the information in Appendix A to determine which blocks should be copied, and which should not.

The file utility programs supplied with the Cortland automatically preserve the structure of sparse files on a copy.

Chapter 3

ProDOS 16 and Cortland Memory

Strictly speaking, memory management is separate from the operating system in the Cortland. This chapter shows how ProDOS 16 uses memory and how it interacts with the Memory Manager.

Cortland memory configurations

The Cortland microprocessor is capable of directly addressing 16megabytes (16Mb) of memory. As shipped, the basic memory configuration for Cortland is 256 kilobytes (256K) of RAM and 128K of ROM, arranged within the 16Mb memory space as shown in Figure 3-1.

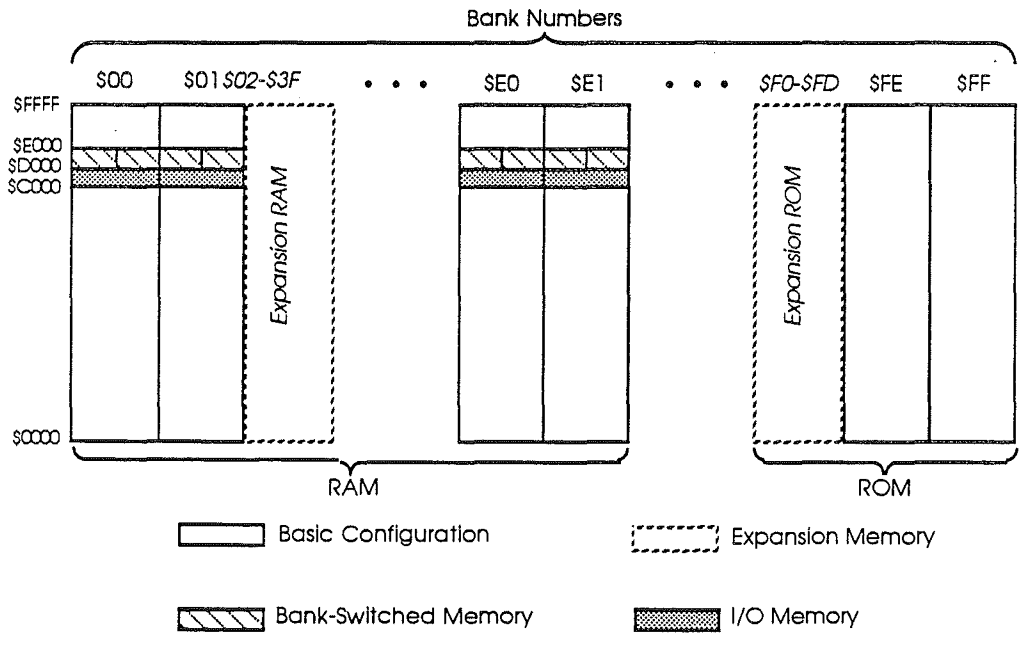


Figure 3-1. Cortland memory map

The total memory space is divided into 256 **banks** of 64K bytes each (see Table 3-1). Banks $00 and $01 are used for system software, applications, and are the memory space used for Apple IIe/IIc **emulation mode**. Banks $E0 and $E1 are used principally for high-resolution video display and additional system software and RAM-based tools. Specialized areas of RAM include I/O space, bank-switched memory, and display buffers in locations consistent with Apple IIe/IIc memory configurations. Banks $FF and $FE are ROM, which contains firmware and ROM-based tools. For a more detailed picture of Cortland Memory, see *Technical Introduction to the Cortland* and *Cortland Hardware Reference.*

Table 3-1. Cortland memory units

Unit	**Size**
bank	65,536 bytes (256 pages)
block	512 bytes (for disk storage)
page	256 bytes
long word	4 bytes
word	2 bytes
byte	8 bits
nibble	4 bits (one-half byte)

With a 1-megabyte Cortland Memory Expansion Card, 16 additional banks of memory are made available; they are numbered sequentially, from $02 to $11. Expansion banks have none of the specialized memory areas shown for banks $00-$01 and $E0-$E1; all 64K bytes in each bank is available for applications.

For Apple IIe/IIc emulation, the Cortland Memory Map is modified so that banks $00 and $01 are identical to the Main and Auxiliary RAM on an Apple IIc or an Apple IIe with extended 80-column card. See *Apple IIc Technical Reference Manual* or *Apple IIe Technical Reference Manual* for details. Because it is used for emulation, the lower 48K of both banks $00 and $01 and the display pages in banks $E0 and $E1 is also called **special memory**; there are restrictions on the placement of certain types of code in special memory. For example, any system software that must remain active in emulation mode cannot be put in special memory. See "Memory Manager" in *Cortland Toolbox Reference* for more details.

ProDOS 16 and System Loader memory map

ProDOS 16 and the System Loader together occupy nearly all addresses from $D000 through $FFFF in both banks $00 and $01. This is the same memory space that ProDOS 8 occupies in the Apple IIe/IIc: all of the Language Card area (addresses above $D000), including most of bank-switched memory.

In addition, ProDOS 16 reserves (through the Memory Manager) approximately 10.7K bytes just below $C000 in bank $00 (in the region normally occupied by `BASIC.SYSTEM` in an Apple IIe/IIc), for I/O buffers, ProDOS 8 interface tables, and other code.

The part of ProDOS 16 that controls loading of both ProDOS 16 and ProDOS 8 programs is located in parts of bank-switched memory in banks $E0 and $E1. Other system softweare occupies most of the rest of the Language Card areas of banks $E0 and $E1.

None of these reserved memory areas is available for use by applications.

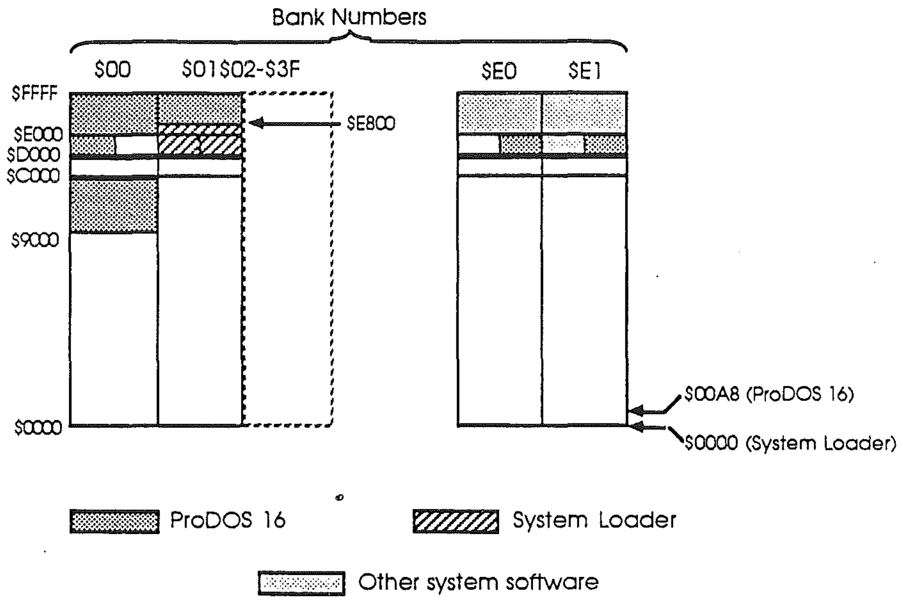


Figure 3-2. ProDOS 16 and System Loader memory map

Entry points and fixed addresses

Because most Cortland memory blocks are movable and under the control of the Memory Manager (see next section), there are very few fixed entry points available to applications programmers. References to fixed entry points in RAM are strongly discouraged, since they are inconsistent with flexible memory management and are sure to cause compatibility problems in future versions of Cortland. Informational system calls and referencing by handles should take the place of access to fixed entry points.

The single supported ProDOS 16 entry point is $E100A8. That location is the entry point for all ProDOS 16 calls.

> **Note:** ProDOS 16 does *not* support the ProDOS 8 Global Page or any other fixed locations used by ProDOS 8.

The single supported System Loader entry point is $E1 00 00. That location is the entry point for all Cortland Tool calls.

Memory management

ProDOS 16 itself does no memory management. All allocation and deallocation of memory in the Cortland is performed by the **Memory Manager**. The Memory Manager is a Cortland Tool; for a complete description of its functions, see *Cortland Toolbox Reference*.

The Memory Manager

The Memory Manager is a ROM-resident Cortland Tool that controls the allocation, deallocation, and repositioning of memory blocks in the Cortland. It works closely with ProDOS 16 and the System Loader to provide the needed memory spaces for loading programs and data and for providing buffers for input/output. All Cortland software, including the System Loader and ProDOS 16, must obtain needed memory space by making requests (calls) to the Memory Manager.

The Memory Manager keeps track of how much memory is free and what parts are allocated to whom. Memory is allocated in **blocks** of arbitrary length; each block possesses several attributes that describe how the Memory Manager may modify it (such as moving it or deleting it), how it must be aligned in memory (for example, on a page boundary), and what program owns it. Table 3-2 lists the Memory Manager attributes that a memory block has.

Table 3-2. Memory block attributes

Attribute	Explanation
movable (yes/no)	Can the block be moved while in memory?
fixed address (yes/no)	Must it be loaded at a specific adress?
fixed bank (yes/no)	Must it be in a particular memory bank?
bank-boundary limited (yes/no)	It is prohibited from extending across a bank boundary?
special-memory usable (yes/no)	May it be in special memory (banks $00 and $01)?
page-aligned (yes/no)	Must it be aligned to a page boundary?
purge level (0 to 3)	Can it be purged? If so, with what priority?
locked (yes/no)	Is the block locked (temporarily nonmovable and unpurgeable)?

Besides creating and deleting memory blocks, the Memory Manager moves blocks when necessary to consolidate free memory. When it **compacts** memory in this way, it of course can move only those blocks that needn't be fixed in location. This implies that as many memory blocks as possible should be movable, if the Memory Manager is to be efficient in compaction.

Pointers and handles

To access an entry point in a movable block, an application cannot use a simple pointer, since the Memory Manager may move the block and change the entry point's address. Instead, each time the Memory Manager allocates a memory block, it returns to the requesting application a **handle** referencing that block.

A handle is a pointer to a pointer; it is the address of a fixed (nonmovable) location that contains the address of the block. If the Memory Manager changes the location of the block, it updates the address in the fixed location; the value of the handle itself is not changed. Thus the application may continue to access the block using the handle, no matter how often the block itself is moved in memory. If a block will always be fixed in memory (**locked** or **nonmovable**), it may be referenced by a pointer instead of by its handle.

ProDOS 16 and the System Loader use both pointers and handles to reference memory locations. Pointers and handles must be at least three bytes long to access the full range of Cortland memory. However, all pointers and handles used by ProDOS 16 are four bytes long, for ease of manipulation by the 16-bit registers in the 65816 microprocessor.

How an application obtains memory

When an application makes a ProDOS 16 call that requires allocation of memory (such as opening a file or writing from a file to a memory location), ProDOS 16 first obtains any needed memory blocks from the Memory Manager and then performs its tasks. Likewise, the System Loader requests any needed memory either directly or indirectly (through ProDOS 16 calls) from the Memory Manager. Conversely, when an application informs the operating system that it no longer needs memory, that information is passed on to the Memory Manager which in turn frees that application's allocated memory. In all of these cases the memory allocation and deallocation is completely automatic, as far as the application is concerned.

Any other memory that an application needs for its own purposes must be requested directly from the Memory Manager. Figure 3-3 shows which parts of the Cortland memory can be allocated through requests to the Memory Manager. Applications for Cortland should avoid requesting absolute (fixed-address) blocks. Chapters 7 and 17 of this manual discuss program memory management further; see also *Programmer's Intorduction to the Cortland.*

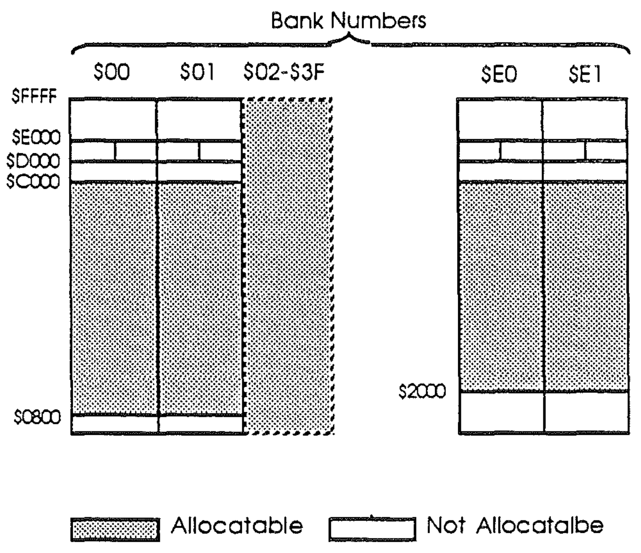


Figure 3-3. Memory allocatable through the Memory Manager

Chapter 4

ProDOS 16 and External Devices

An **external device** is a piece of equipment (hardware) that transfers information to or from the Cortland. Disk drives, printers, mice, and joysticks are external devices. The keyboard and screen are also considered external devices. An **input** device transfers information *to* the computer, an **output** device transfers information *from* the computer, and an **input/output** device transfers information both ways.

This chapter discusses how ProDOS 16 provides an interface between applications and certain external devices.

Block devices

A block device reads and writes information in multiples of one **block** of characters (512 bytes) at a time. Furthermore, it is a **random-access** device: it can access any block on demand, without having to scan through the preceding or succeeding blocks. Block devices are usually used for storage and retrieval of information, and are always input/output devices. Disk drives are block devices.

ProDOS 16 supports access to block devices; that is, you may read from or write to a block device by making ProDOS 16 calls. In addition to READ, WRITE, and the other file calls described in Chapter 2, ProDOS 16 provides three "lower-level" device-access calls. These calls allow you to access information on a block device without considering what files the information is in. The calls are

GET_DEV_NUM	returns the device number associated with a particular named device
READ_BLOCK	reads one block (512 bytes) of data from a specified device
WRITE_BLOCK	writes one block (512 bytes) of data to a specified device

The device number of a device is a required input for the other ProDOS 16 device calls. The block read and write calls are powerful but are not needed by most applications; the file access calls described in Chapter 2 are sufficient for normal disk I/O.

A block device generally requires a *device driver* to translate ProDOS 16's **logical** block device model into the **tracks** and **sectors** by which information is actually stored on the **physical** device. The device driver may be circuitry within the disk drive itself (Unidisk 3.5), it may be included as part of ProDOS 16 (Disk II), or it may be on a separate card in an expansion slot.

Chapter 7, "Adding Routines to ProDOS 16," describes the device drivers recognized by ProDOS 16 and presents the **protocol** that any drivers used on the Cortland must follow.

Note on RAM disks: RAM disks are internal software constructs that the operating system treats like external devices. Although ProDOS 16 provides no particular support for RAM disks, any RAM disk that behaves like a block device in all respects will be supported just as if it were an external device.

Character devices

A character device reads or writes a stream of characters in order, one at a time. It is a **sequential-access** device: it cannot access any position in a stream without first accessing all previous positions. It can neither skip ahead nor go back to a previous character. Character devices are usually used to pass information to and from a user or another computer; some are input devices, some are output devices, and some are input/output devices. The keyboard, screen, printer and communications port are character devices.

ProDOS 16 does not provide direct support for character devices; that is, you cannot access them through ProDOS 16 calls. Consult the appropriate firmware or tools documentation, such as *Cortland Firmware Reference* or *Cortland Toolbox Reference*, for instructions on how to make calls to the particular device you wish to use.

Named devices

ProDOS 16 permits block devices to have assigned names. This ability is a convenience for users, because they will no longer have to know thwe volume name to access a disk.

However, ProDOS 16's support for named devices is limited; device names may be used only in the `VOLUME` and `GET_DEV_NUM` calls. Other calls require either a volume name or the device number returned by the `GET_DEV_NUM` call.

Devices are named according to a built-in convention; assigned names may not be changed. The naming convention is as follows:

Device	**Name**
Any block device	*.Dn*

where *n*= a 1-digit or 2-digit number (assigned consecutively)

Note on RAM-based drivers: The capacity for named devices lends itself to future incorporation of RAM-based drivers; however, RAM-based drivers arel not supported in the present release of ProDOS 16. If RAM-based drivers are developed later, they may have their own names that could supersede the ProDOS 16-assigned names.

Number of online devices

ProDOS 16 (v.1) supports up to 14 active devices at a time. The Cortland normally accepts up to 4 devices connected to its disk port (Smartport) and two devices per expansion slot (slots 1 through 7). It is possible, however, to have up to 4 devices on a single slot (slot 5 with a Smartport card). Nevertheless, the total number of devices on line still cannot exceed 14.

Device search at startup

When ProDOS 16 boots, it performs a device search to identify all built-in pseudo-slot ROMs (internal ROMs) and all real physical slot ROMs (card ROMs). Every block device found is incorporated into ProDOS 16's list of devices, and assigned a device number (`dev_num`) and device name (`dev_name`).

> **Note:** Control Panel settings determine whether internal ROM or card ROM is active for each slot. ProDOS 16 cannot support both internal and external devices with the same slot number.

Volume control blocks

For each device with nonremovable media (such as a hard disk) found at boot time, a volume control block (VCB) is created in memory. The VCB keeps track of the characteristics of that online volume. For other devices (such as floppy disk drives) found at boot time, VCB's are created as files are opened on the volumes in those devices. A maximum of eight VCB's may exist at any one time; if you try to open a file on a device whose volume presently has no open files, and if there are already eight VCB entries, error $55 (VCB table full) is returned. Thus, even though there may be up to 14 devices connected to your system, only eight (at most) can be active (open) at any one moment.

Interrupt handling

On the Cortland, interrupts may be handled at either the firmware or the software level. The built-in interrupt handers are in firmware (see *Cortland Firmware Reference*); user-installed interrupt handlers are software and may be installed through ProDOS 16.

When the Cortland detects an interrupt that is to be handled through ProDOS 16, it dispatches through the interrupt vector at $00 03 FE (page 3 in bank zero). At this point the machine is executing in emulation mode, using the standard clock speed and 8-bit registers. The vector at $00 03 FE has only two address bytes; in order to allow access to all of Cortland memory, it points to another bank zero location. The vector in that location then passes control to the ProDOS 16 interrupt dispatcher. The interrupt dispatcher switches to full native mode (including higher clock speed) and then polls the user-installed interrupt handlers.

Figure 4-1 is a simplified picture of what happens when a device generates an interrupt that is handled through a ProDOS 16 interrupt handler.

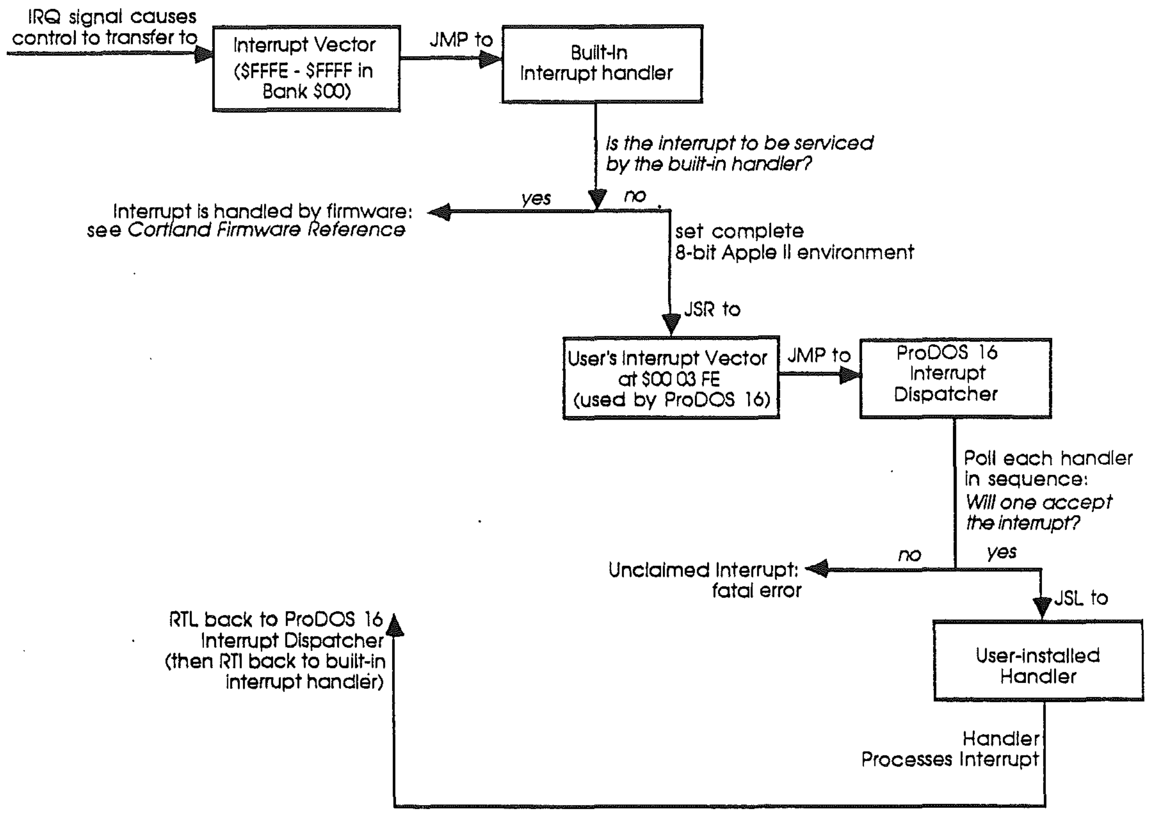

IRQ signal causes control to transfer to
Interrupt Vector ($FFFE - $FFFF in Bank $00)
JMP to
Built-in Interrupt handler
Is the interrupt to be serviced by the built-in handler?
yes
no
Interrupt is handled by firmware: see Cortland Firmware Reference
set complete 8-bit Apple II environment
JSR to
User's Interrupt Vector at $00 03 FE (used by ProDOS 16)
JMP to
ProDOS 16 Interrupt Dispatcher
Poll each handler in sequence: Will one accept the interrupt?
no
yes
Unclaimed Interrupt: fatal error
JSL to
User-installed Handler
Handler Processes Interrupt
RTL back to ProDOS 16 Interrupt Dispatcher (then RTI back to built-in interrupt handler)

Figure 4-1. Interrupt handling through ProDOS 16

ProDOS 16 supports up to 16 user-installed interrupt handlers. When an interrupt occurs that is not handled by firmware, ProDOS 16 transfers control to each handler successively until one of them claims it. There is no grouping of interrupts into classes; their priority rankings are reflected only by the order in which they are polled.

If you write an interrupt-handling routine, to make it active you must install it with the `ALLOC_INTERRUPT` call; to remove it, you must use the `DEALLOC_INTERRUPT` call. Be sure to enable the hardware generating the interrupt only *after* the routine to handle it is allocated; likewise, disable the hardware *before* the routine is deallocated. See Chapter 8 for further details on writing and installing interrupt handlers.

Unclaimed interrupts

An **unclaimed interrupt** is defined as the condition in which the hardware Interrupt Request Line (IRQ) is active (being pulled low), indicating that an interrupt-producing device needs attention, yet none of the installed interrupt handlers claims responsibility for the interrupt. When an interrupt signal occurs and ProDOS 16 can find no handler to claim it, it assumes that a serious hardware error has occurred. It issues a system failure error message to the System Death Manager (see *Cortland Toolbox Reference*), and stops processing the current application. Processing cannot resume until the user reboots the system.

Chapter 5

ProDOS 16 and the Operating Environment

ProDOS 16 is one of the many components that make up the Cortland **operating environment**, the overall hardware and software setting within which Cortland application programs run. This chapter describes how ProDOS 16 functions in that environment and how it relates to the other components.

Cortland system disks

A Cortland **system disk** is a disk containing the system software needed to run any application you wish to execute. Most system disks contain one or more operating systems (ProDOS 16 and ProDOS 8), the System Loader, RAM-based Tool Sets, RAM patches to ROM-based Tool Sets, fonts, desk accessories, boot-time initialization programs, and possibly one or more applications.

There are two basic types of system disks: *complete* system disks and *application* system disks. A complete system disk has a full set of Cortland system software, as listed in table 5-1. It is a resource pool from which application system disks can be constructed. An application system disk has one or more application programs and only the specific system software it needs to run the application(s). For example, a word processor system disk may include a large selection of fonts, whereas a spreadsheet system disk may have only a few fonts.

Software developers may create application system disks for their programs. Users may also create application system disks, perhaps by combining several individual application disks into a multiapplication system disk. Apart from the essential files listed in table 5-2, there is no single set of required contents for application system disks.

Complete System Disk

Every Cortland user (and developer) needs at least one complete system disk. It is a pool of system software resources, and may contain files missing from any of the available application system disks. Table 5-1 lists the contents of a complete system disk.

Table 5-1. Contents of a complete Cortland system disk

Directory/File	**Description**
`PRODOS`	a routine that loads the proper operating system and selects an application, both at boot time and whenever an application quits
`SYSTEM/`	a subdirectory containing the following files:
`P8`	ProDOS 8 operating system
`P16`	ProDOS 16 operating system
`LOADER`	the Cortland System Loader
`START`	typically a program selector
`LIBS/`	a subdirectory containing the standard system libraries
`TOOLS/`	a subdirectory containing all RAM-based tools
`FONTS/`	a subdirectory containing all fonts
`DESK.ACCS/`	a subdirectory containing all desk accessories
`SYSTEM.SETUP/`	a subdirectory containing system initialization programs
`TOOL.SETUP`	a load file containing patches to ROM and a program to install them. This is the only *required* file in the `SYSTEM.SETUP/` subdirectory; it is executed before any others in the subdirectory.
`BASIC.SYSTEM`	The Applesoft BASIC system interface program

The complete system disk is an 800K byte, double-sided 3.5-inch diskette; the required files will not fit on a 140K, single-sided 5.25-inch diskette.

When you boot a complete system disk, it executes the file `SYSTEM/START`. From the `START` file, you may choose to call Applesoft BASIC; no other application programs are available on the disk.

The SYSTEM.SETUP/ subdirectory

The `SYSTEM.SETUP/` subdirectory may contain several different types of files, all of which need to be loaded and initialized at boot time. They include the following:

- **The file TOOL.SETUP:** This file must always be present; it is executed before any others in `SYSTEM.SETUP/`. `TOOL.SETUP` installs and initializes any RAM patches to ROM-based Tool Sets. After `TOOL.SETUP` is finished, ProDOS 16 loads and executes the remaining files in the `SYSTEM.SETUP/` subdirectory, which may belong to any of the categories listed below.
- **Permanent Init Files (filetype $B6):** These files are loaded and executed just like standard applications (type $B3), but they are not shut down when finished. They also must have certain characteristics:
  1. They must be loaded in non-special memory.
  2. They cannot permanently allocate any stack/direct-page space.
  3. They must terminate with an `RTL` rather than a `QUIT`.
- **Temporary Init Files (type $B7):** These files are loaded and executed just like standard applications (type $B3), and they are shut down when finished. They must terminate with an `RTL` rather than a `QUIT`.

- **New Desk Accessories (type $B8):** These files are loaded but not executed. They must be in non-special memory.
- **Classic Desk Accessories (type $B9):** These files are loaded but not executed. They must be in non-special memory.

Application system disks

Each application program or group of related programs comes on its own application system disk. The disk has all of the system files needed to run that application, but it may not have all the files present on a complete system disk. Different applications may have different system files on their application system disks.

For example, the *ProDOS Programmer's Disk*, included with this manual, is an application system disk. It contains all the system files listed above, plus the files `EXER16` and `EXER8`. Those two files are the *ProDOS 16 Exerciser* and the *ProDOS 8 Exerciser*, respectively. The *ProDOS 16 Exerciser* is described in Chapter 8 and Appendix C of this manual; he *ProDOS 8 Exerciser* is described in the *ProDOS 8 Reference.****It is not clear that the disks will actually be shipped in this format.***

Table 5-2 shows which files must be present on all application system disks, and which files are needed only for particular applications. In some very restricted instances, it may be possible to fit an application and its required system files onto a 5.25-inch (140K) diskette; most applications, however, require an 800K diskette.

Table 5-2. Contents of a Cortland application system disk

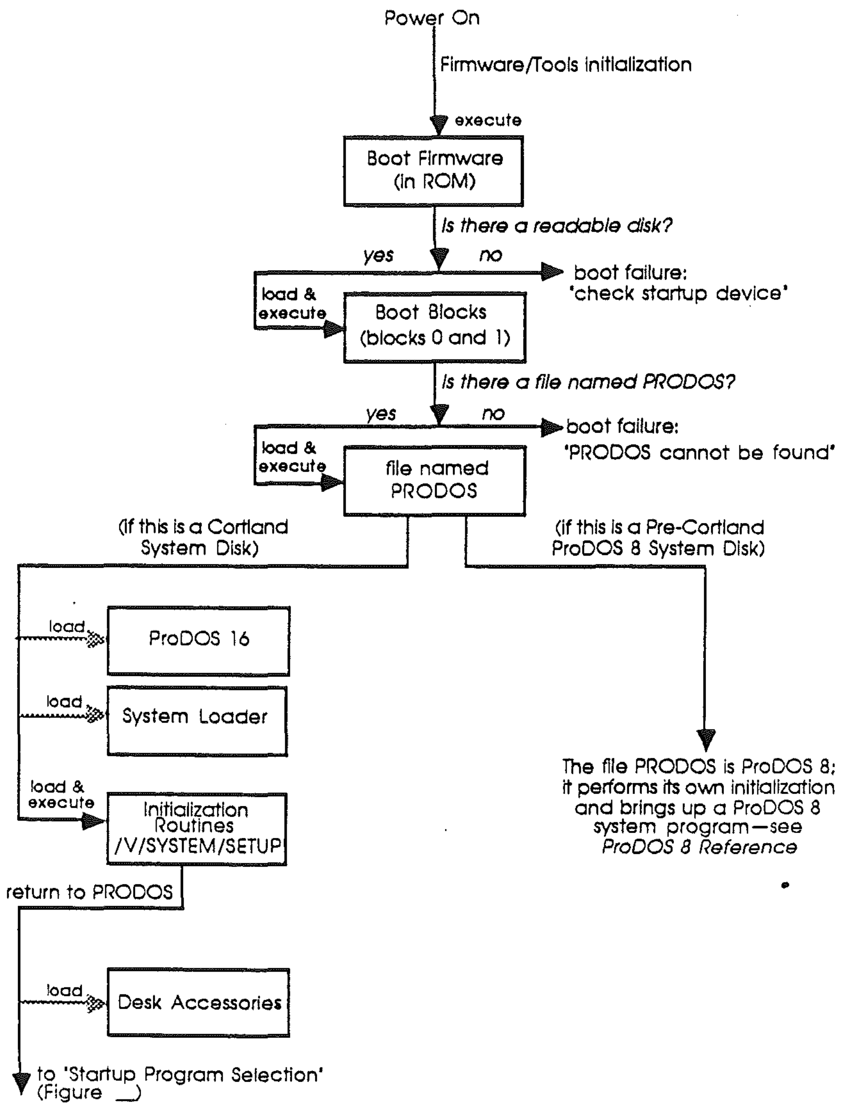
Directory/File	Required/(Required If...)
PRODOS	**required**
SYSTEM/	**required**
P8	(required if the application is ProDOS 8-based)
P16	**required**
LOADER	**required**
START	(required if the program selector is to be used)
LIBS/	(required if system library routines are needed)
TOOLS/	(required if the application needs RAM-based tools)
FONTS/	(required if the application needs fonts)
DESK.ACCS/	(required if desk accessories are to be provided)
SYSTEM.SETUP/	**required**
TOOL.SETUP	**required**
BASIC.SYSTEM	(required if the application is written in Applesoft BASIC)

System startup

Disk blocks 0 and 1 on a Cortland system disk contain the startup (boot) code. They are identical to the boot blocks on Apple IIe/IIc system disks (ProDOS 8 system disks). This allows ProDOS 8 system disks to boot on a Cortland, and it also means that the initial part of the ProDOS 16 bootstrap procedure is identical to that for ProDOS 8.

Figure 5-1 shows the boot initialization procedure. First, the boot firmware in ROM reads the boot code (blocks 0 and 1) into memory and executes it. For a system disk with a volume name /V,

1. The boot code searches the disk's volume directory for the first file named /V/PRODOS with the file type $FF.
2. If the file is found, it is loaded and executed at location $2000 of bank $00.



Power On
Firmware/Tools initialization
execute
Boot Firmware (in ROM)
Is there a readable disk?
yes
no
boot failure: 'check startup device'
load & execute
Boot Blocks (blocks 0 and 1)
Is there a file named PRODOS?
yes
no
boot failure: 'PRODOS cannot be found'
load & execute
file named PRODOS
(if this is a Cortland System Disk)
(if this is a Pre-Cortland ProDOS 8 System Disk)
load
ProDOS 16
load
System Loader
load & execute
Initialization Routines /V/SYSTEM/SETUP
return to PRODOS
load
Desk Accessories
to 'Startup Program Selection' (Figure __)
The file PRODOS is ProDOS 8; it performs its own initialization and brings up a ProDOS 8 system program—see ProDOS 8 Reference

Figure 5-1. Boot initialization sequence

From this point on, a Cortland system disk behaves differently from an Apple IIe/IIc system disk. On an Apple IIe/IIc system disk, the file named PRODOS is the ProDOS 8 operating system. On a Cortland system disk, however, this PRODOS file is not the operating system itself; it is an operating system loader and application selector. When it receives control from the boot code, /V/PRODOS performs the following tasks (see also Figure 5-1):

3. It relocates the part of itself named PQUIT to an area in memory where PQUIT will reside permanently. PQUIT contains the code required to terminate one program and start another (either ProDOS 8 or ProDOS 16 application).
4. /V/PRODOS loads the ProDOS 16 operating system (file /V/SYSTEM/P16).
5. Using ProDOS 16 calls, /V/PRODOS loads the Cortland System Loader (file /V/SYSTEM/LOADER).
6. /V/PRODOS performs any necessary boot initialization of the system, by executing the files in the subdirectory /V/SYSTEM/SYSTEM.SETUP. If there is a file named TOOL.SETUP in that subdirectory, it is executed first—it loads RAM-based tools and RAM patches to ROM-based tools.

   Every file in the subdirectory /V/SYSTEM/SYSTEM.SETUP must be a Cortland load file of type $B6, $B7, $B8, or $B9. These file types are described under "The SYSTEM.SETUP Subdirectory," in this chapter. After executing TOOL.SETUP, /V/PRODOS loads and executes, in turn, every other file that it finds in the subdirectory.
7. Now /V/PRODOS selects (= determines the pathname of) the system program or application to run. Figure 5-2 shows this procedure.
   a. It first searches for a type $B3 file named /V/SYSTEM/START. Typically, that file is a program selector, but it could be any Cortland application. If START is found, it is selected.
   b. If there is no START file, /V/PRODOS searches the boot volume directory for the first file that is either 1.) a ProDOS 8 system program (type $FF with the filename extension .SYSTEM), or 2.) a ProDOS 16 system program (type $B3 with the filename extension .SYS16). Whichever is found first is selected.

   Note: If a ProDOS 8 system program is found first, but the ProDOS 8 operating system (file /V/SYSTEM/P8) is not on the system disk, /V/PRODOS will then search for and select the first ProDOS 16 system program (ProDOS 16 is *always* on the system disk).

   c. If /V/PRODOS cannot find a file to execute (for example, if there is no START file and there are no ProDOS 8 or ProDOS 16 system programs), it will bring up an interactive routine that prompts the user for the filename of an application to load.
8. Finally, /V/PRODOS passes control to an entry point in PQUIT. It is PQUIT, not /V/PRODOS, that actually loads the selected program. The next section describes that procedure.

Note: PRODOS will write an error messsage to the screen if you try to boot it on an Apple II computer other than a Cortland. This is because ProDOS 8 on a Cortland disk is in the file V/SYSTEM/P8, not in the file PRODOS.

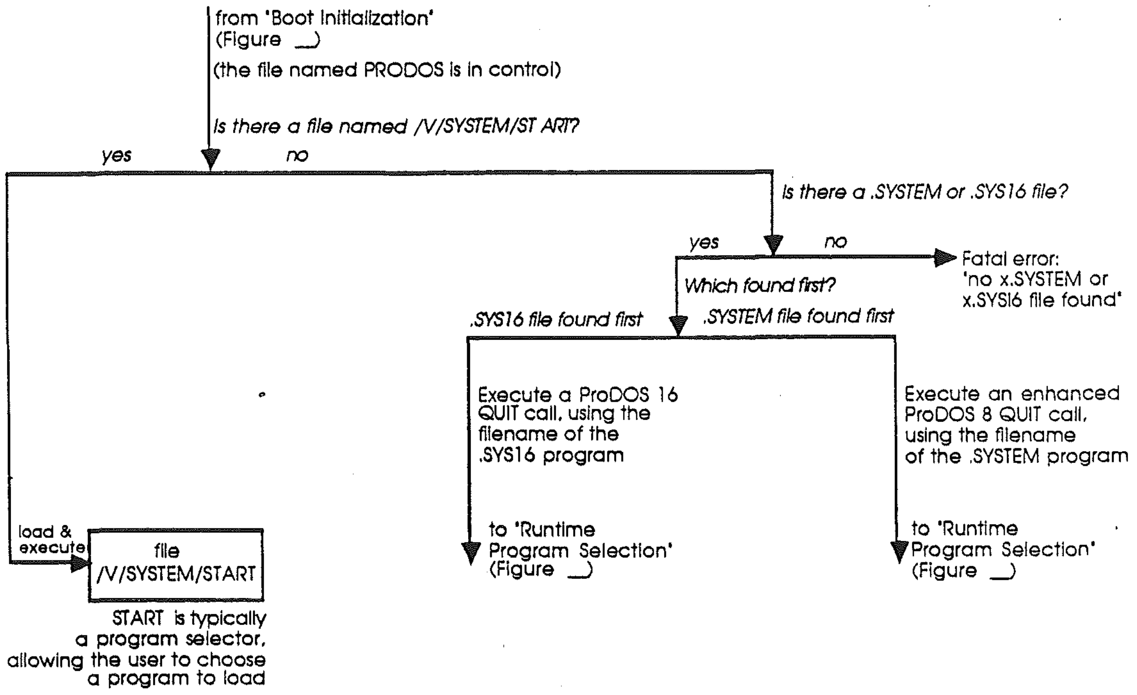

from 'Boot Initialization' (Figure _)
(the file named PRODOS is in control)
Is there a file named /V/SYSTEM/START?
yes
no
Is there a .SYSTEM or .SYS16 file?
yes
no
Fatal error: 'no x.SYSTEM or x.SYS16 file found'
Which found first?
.SYS16 file found first
.SYSTEM file found first
Execute a ProDOS 16 QUIT call, using the filename of the .SYS16 program
Execute an enhanced ProDOS 8 QUIT call, using the filename of the .SYSTEM program
to 'Runtime Program Selection' (Figure _)
to 'Runtime Program Selection' (Figure _)
load & execute
file /V/SYSTEM/START
START is typically a program selector, allowing the user to choose a program to load

Figure 5-2. Startup program selection

Starting and Quitting Applications

The Cortland startup sequence ends when control is passed to the program selection routine (`PQUIT`). This routine is entered both at boot time and whenever an application terminates.

PQUIT

`PQUIT` is the ProDOS program dispatcher. It determines which ProDOS 8 or ProDOS 16 program is to be run next, and runs it. After startup, both `PQUIT` and the System Loader are permanently resident in memory; `PQUIT` loads ProDOS 16 programs through calls to the System Loader.

`PQUIT` has two entry points: `P8PQUIT` and `P16PQUIT`. Whenever a ProDOS 8 application executes a `QUIT` call, control passes through the `P8PQUIT` entry point. Whenever a ProDOS 16 application executes a `QUIT` call, control passes through the `P16PQUIT` entry point. To launch the first program at system startup, `/V/PRODOS` makes a ProDOS 8 or ProDOS 16 `QUIT` call and passes control to the proper `PQUIT` entry point.

`PQUIT` supports three types of quit call: the standard ProDOS 8 `QUIT` call, an enhanced ProDOS 8 `QUIT` call, and the ProDOS 16 `QUIT` call.

Standard ProDOS 8 QUIT call

The standard ProDOS 8 `QUIT` call's parameter block consists of a one-byte parameter count field, followed by four null fields in this order: byte, word, byte, word. As ProDOS 8 is currently defined, all fields must be present and all must be set to zero. There is thus no way for a program to use the standard `QUIT` call to specify the pathname of the next program to run.

Enhanced ProDOS 8 QUIT call

The enhanced ProDOS 8 `QUIT` call differs from the standard call only in the permissible values of the first two parameters. In the enhanced `QUIT` call, the first (byte) parameter is defined as the *quit type*. If it is zero, the call is identical to a standard `QUIT` call; if it is $EE, `PQUIT` interprets the following (word) parameter as a pointer to a string which is the pathname of the next program to run.

The enhanced ProDOS 8 `QUIT` call is meaningful only on the Cortland, and only when `PQUIT` is present to interpret it. It behaves like the standard `QUIT` call in any other situation.

ProDOS 16 QUIT call

The ProDOS 16 `QUIT` call has two parameters: a pointer to the pathname of the next program to execute, and a boolean *return flag* to notify `PQUIT` whether or not control should eventually return to the program making the `QUIT` call.

If the value of the return flag is true, `PQUIT` pushes the UserID of the calling (=quitting) program onto an internal stack. As subsequent programs run and quit, several UserID's may be pushed onto the stack. With this mechanism, multiple levels of shells may execute subprograms and subshells, while ensuring that they eventually regain control when their subprograms quit.

For example, the program selector (`START` file) might pass control to a software development system shell, using the `QUIT` call to specify the shell and placing its own ID on the stack. The shell in turn could hand control to a debugger, likewise puting its own ID on the stack. If the debugger quits without specifying a pathname, control would pass automatically back to the shell; when the shell quit, control would pass automatically back to the `START` file.

This automatic return mechanism is specific to the ProDOS 16 `QUIT` call, and therefore is not available to ProDOS 8 programs. When a ProDOS 8 application quits, it cannot put its ID on the internal stack.

QUIT procedure

This is a brief description of how `PQUIT` handles all three types of `QUIT` call. Refer also to Figure 5-3.

1. If a ProDOS 16 or enhanced ProDOS 8 `QUIT` call specifies a pathname, `PQUIT` attempts to execute the specified file. Under certain conditions this may not be possible: the file may not be on line, there may be insufficient memory , and so on. In that case the `QUIT` call fails and returns an error.
2. If the `QUIT` call specifies no pathname, `PQUIT` pulls a UserID off its internal ID stack and attempts to execute that program. Typically, programs with UserID's on the stack are in the System Loader's **dormant** state (see "User Shutdown" in Chapter 18), and it may be possible to restart them without reloading them from disk. Under certain conditions it may not be possible to execute the program: the file may not be on line, there may be insufficient memory , and so on. In that case the `QUIT` call fails and returns an error.
3. If the `QUIT` call specifies no pathname and the ID stack is empty, `PQUIT` executes an interactive routine that allows the user to do any of these:
   - reboot the system
   - execute the file `/V/SYSTEM/START`
   - enter the pathname of a program to execute
4. If the quitting program is a ProDOS 16 program, `PQUIT` calls the loader's User Shutdown routine to place that program in a dormant state.
5. Once it has determined which program to load, `PQUIT` knows which operating system is required. If it is not the *current* system,
   a. `PQUIT` shuts down the current operating system and loads the required one.
   b. `PQUIT` then makes Memory Manager calls to free memory used by the former operating system and allocate memory needed by the new system. If the new operating system is ProDOS 8, `PQUIT` allocates all special memory for the program.
6. The new program is loaded. `PQUIT` calls the System Loader to load ProDOS 16 programs; for ProDOS 8 programs, `PQUIT` passes control to ProDOS 8, which then loads and executes its own program directly.
7. Finally (if it is a ProDOS 16 program), `PQUIT` sets up various aspects of the program's environment, including the direct-register and stack-pointer values, and passes control to the program.

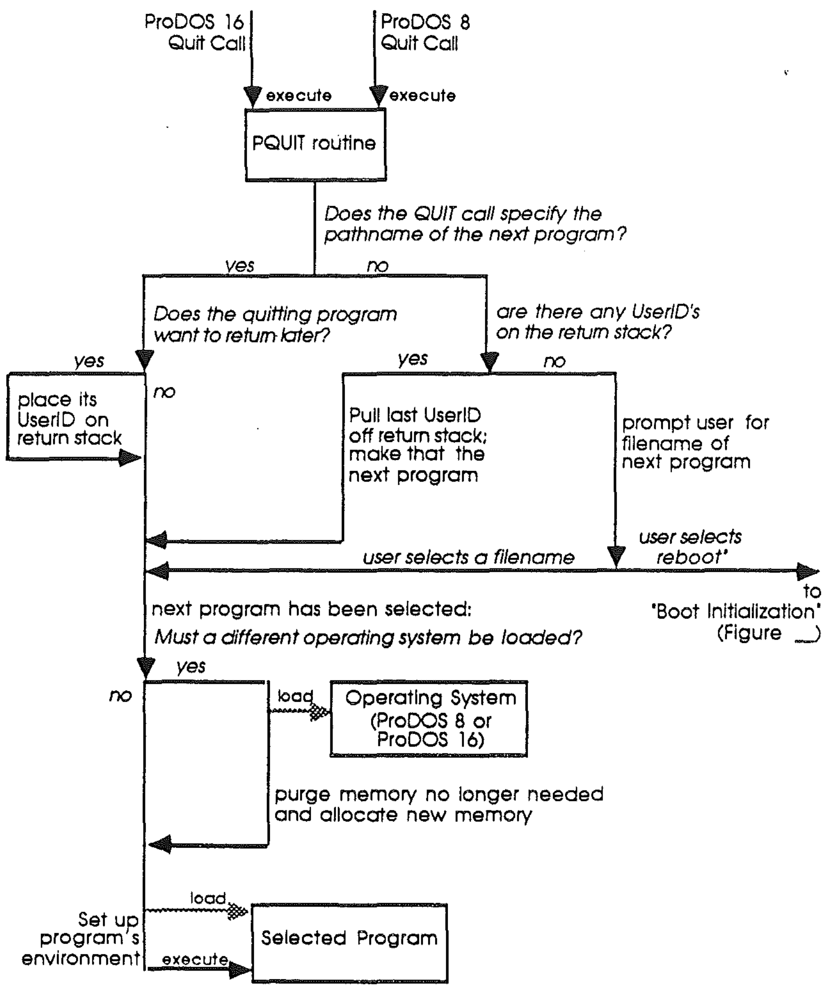

ProDOS 16 Quit Call
ProDOS 8 Quit Call
execute
execute
PQUIT routine
Does the QUIT call specify the pathname of the next program?
yes
no
Does the quitting program want to return later?
are there any UserID's on the return stack?
yes
no
yes
no
place its UserID on return stack
Pull last UserID off return stack; make that the next program
prompt user for filename of next program
user selects a filename
user selects reboot*
to 'Boot Initialization' (Figure __)
next program has been selected:
Must a different operating system be loaded?
yes
no
load
Operating System (ProDOS 8 or ProDOS 16)
purge memory no longer needed and allocate new memory
load
Set up program's environment
execute
Selected Program

Figure 5-3. Run-time program selection (QUIT call)

When the just-launched program receives control from PQUIT, the machine is in the following state:***information not yet available***

Pathname prefixes

A user-assigned prefix is convenient when many files in the same subdirectory are accessed, because it shortens the pathname references. A *set* of prefixes is convenient when files in several different subdirectories must be repeatedly accessed. The System Loader, for example, makes use of multiple prefixes. Once the pathnames are assigned to prefixes, you can refer to the prefixes instead of remembering all the different pathnames.

ProDOS 16 will support 8 prefixes, referred to by the **prefix designators** `0/`, `1/`, `2/`,...,`7/`. Each prefix designator must include a terminating slash to separate it from the rest of the pathname. Three of the prefix designators have default values:

`0/` is the system prefix—the name of the volume from which the presently running ProDOS 16 was booted.

`1/` is the application subdirectory prefix—the pathname of the subdirectory that contains the currently running application.

`2/` is the system library subdirectory prefix—the pathname of the subdirectory that contains the library modules used by the currently running application. On a typical Cortland startup disk, prefix `2/` is `/V/SYSTEM/LIBS` (where `/V/` is the volume name).

Your application may assign the rest of the prefixes; they have no default or startup values. In fact, once an application is running, it may also change the values of prefixes `0/`, `1/`, or `2/`.

The prefix designators are set (assigned to specific pathnames) and retrieved through the `SET_PREFIX` and `GET_PREFIX` calls. Although a partial prefix may be used in the `SET_PREFIX` call, prefixes are always stored in memory as complete pathnames. Once set, a prefix designator may be placed at the beginning of a partial pathname, replacing the actual prefix. If no prefix designator is specified in a partial pathname, ProDOS 16 assigns the designator `0/` and attaches the system prefix.

Note: The default value of the ProDOS 16 system prefix (designated by `0/`) is equivalent to the "system prefix" recognized by ProDOS 8.

The following are some examples of prefix use. They assume that the system prefix (`0/`) is set to `/a/b` and that prefix `5/` is set to `/m/n`. The pathname provided by the caller is on the left; the full pathname, constructed by ProDOS 16, is on the right.

`/x/y/z`	`/x/y/z`	(full pathname provided)
`p/q/r`	`/a/b/p/q/r`	(partial pathname—implicit use of system prefix)
`0/p/q/r`	`/a/b/p/q/r`	(explicit use of system prefix)
`5/p/q/r`	`/m/n/p/q/r`	(use of prefix `5/`)

Tools, firmware, and system software

Although ProDOS 16 is the principal part of the Cortland operating system, several "operating system-like" functions are actually carried out by other software components. This section briefly describes some of those components; for detailed information see the references listed with each one.

The Memory Manager

As explained in Chapter 3, the Memory Manager takes care of all memory allocation, deallocation, and housekeeping chores. Applications obtain needed memory space either directly, through requests to the Memory Manager, or indirectly through ProDOS 16 or System Loader calls (which in turn obtain the memory through requests to the Memory Manager).

The Memory Manager is a ROM-resident Cortland Toolset; for more detailed information on its functions and how to call them, see *Cortland Toolbox Reference*.

The System Loader

The System Loader is a Cortland Toolset that works very closely with ProDOS 16 and the Memory Manager. It resides on the system disk, along with ProDOS 16 and other system software (see "Cortland System Disks" in this chapter). All programs and data are loaded into memory by the System Loader.

The System Loader supports both static and dynamic loading of segmented programs and subroutine libraries. It loads files that conform to a specific format (**object module format**); such files are produced by the **Cortland Programmer's Workshop Linker** and other components of the Cortland Programmer's Workshop (see *Cortland Programmer's Workshop Reference*).

The System Loader is described in Part III of this manual.

The Scheduler

The Scheduler is a **Heartbeat Task**, a routine that functions in conjunction with the Cortland Heartbeat Interrupt signal (see "Heartbeat Tools" in *Cortland Toolbox Reference*). Its purpose is to coordinate the execution of interrupt handlers and other interrupt-based routines such as desk accessories.

The Scheduler is required only when an interrupt routine needs to call a piece of system software, such as ProDOS 16, that is not **reentrant**. If ProDOS 16 is in the middle of a call when an interrupt occurs, the interrupting routine cannot itself call ProDOS 16, because that would disrupt the first (not yet completed) call. The system needs a way of telling an interrupt routine to hold off until the system software it needs is no longer busy.

The Scheduler accomplishes this by periodically checking a firmware flag called the **Busy word** and maintaining a queue of processes that may be activated when the busy flag is cleared. Interrupt routines that make operating system calls must go through the Scheduler (see Chapter 8).

The UserID Manager

The UserID Manager is a Miscellaneous Tool that provides a way for programs to obtain unique identification numbers. Every memory block allocated by the Memory Manager is marked with a UserID that shows what system software, application, or desk accessory it belongs to.

Part of each block's 2-byte UserID is a **Type ID field**, describing the category of load segment that occupies it. All ProDOS 8 and ProDOS 16 blocks are type 3; System Loader blocks are type 7; blocks of controlling programs (such as a shell or switcher) are type 2; and blocks containing application segments are type 1. Appendix D diagrams the format for the UserID word. See "Miscellaneous Tools" in *Cortland Toolbox Reference* for further details.

The System Death Manager

All fatal errors, including fatal ProDOS 16 errors, are routed through the System Death Manager, a Miscellaneous Tool. It displays a default message on the screen, or, if passed a pointer when it is called, displays an ASCII string with a user-chosen message. Program execution halts when the System Death Manager is called.

The System Death Manager is described under "Miscellaneous Tools" in *Cortland Toolbox Reference* .

Chapter 6

Programming With ProDOS 16

This chapter presents requirements and suggestions for writing Cortland programs that use ProDOS 16.

Programming suggestions for the System Loader are in Chapter 17 of this manual. More general information on how to program for the Cortland is available in *Programmer's Introduction to the Cortland*. For language-specific programming instructions, consult the appropriate language manual in the Cortland Programmer's Workshop (see "Cortland Programmer's Wokshop" in this chapter).

Application requirements

As used in this manual, an **application** is a complete program, typically called by a user (rather than another program), that can communicate directly with ProDOS 16 and any other system software or firmware it needs. For example, word processors, spreadsheet programs, and language interpreters are examples of applications. Data files and source-code files, as well as subroutines, libraries, and utilities that must be called from other programs are not applications.

A **system program** may be defined as a stand-alone application. It is a program that can run as a start-up program (booted from the system disk).

To be an application, a Cortland program must

- consist of executable machine language code
- be in Cortland Object Module Format (see Appendix D)
- be file type $B3 - $BF (see Appendix A)
- have a filename extension of `.SYS16` (if you want it to be a *system program*)
- make ProDOS 16 calls as described in this manual (see Chapter 8)
- observe the ProDOS 16 `QUIT` conventions (see Chapter 5)
- observe all other applicable ProDOS 16 conventions, such as the conventions for interrupt handlers (see Chapter 7)
- get all needed memory from the Memory Manager (see Chapter 2)

All other aspects of the program are up to you. The rest of this chapter presents conventions and suggestions to help you create an efficient and useful application or system program, consistent with Cortland programming concepts and practices.

Stack and direct page

In the Cortland, the 65816 microprocessor's stack-pointer register is 16 bits wide; that means that the hardware stack may be located anywhere in bank $00 of memory. Also, the stack may be as much as 64K bytes deep. In theory, then, the stack may occupy any unused space of any size in bank $00. In practice, however, the stack is limited to about 32K bytes because of reserved memory areas in bank $00 (see Chapter 3).

The **direct page** is the Cortland equivalent to a **zero page**. The difference is that it need not be page zero in memory. Like the stack, the direct page may be placed in any unused area of bank $00. The microprocessor's *direct register* is 16 bits wide, and all zero-page (direct-page) addresses are added as offsets to the contents of that register.

In practice, however, less space is available. First, only the lower 48K bytes of bank $00 can be allocated; the rest is reserved for I/O and system software. Also, because more than one program can be active at a time, there may be more than one stack and more than one direct page in bank $00. Furthermore, many applications may want to have parts of their code as well as their stacks and direct pages in bank $00.

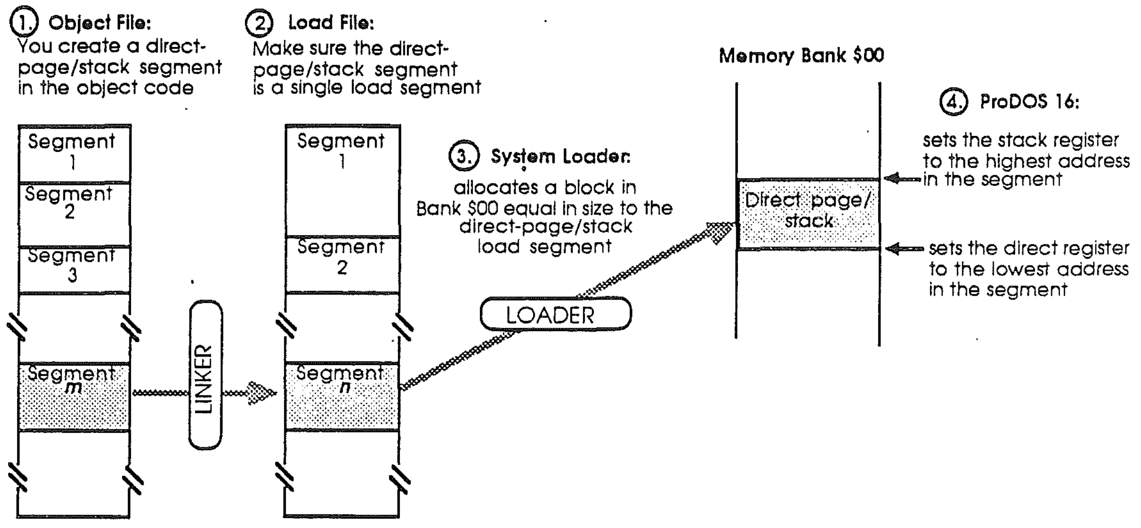
Your program should therefore be as efficient as possible in its use of stack and direct-page space. The total size of both should probably not exceed about 4K bytes in most cases. Still, that gives you the opportunity to write programs that require stacks and direct pages much larger than the 256 bytes available for each on other Apple II's.

Automatic allocation of stack and direct page

Only you can decide how much stack and direct-page space your program will need when it is running. The best time to make that decision is during program development, when you create your source file(s). If you specify at that time the total amount of space needed, ProDOS 16 and the System Loader will automatically allocate it and set the stack and direct registers each time your program runs.

Definition during program development

You define your program's stack and direct-page needs by specifying a "direct-page/stack" object segment (KIND = $92) when you assemble or compile your program (Figure 6-1). The size of the segment is the total amount of stack and direct-page space your program needs. It is not *necessary* to create this segment; if you need no such space or if the ProDOS 16 default (see below) is sufficient,you may leave it out.



1. Object File:
You create a direct-page/stack segment in the object code
2. Load File:
Make sure the direct-page/stack segment is a single load segment
Segment 1
Segment 2
Segment 3
Segment m
LINKER
Segment 1
Segment 2
Segment n
3. System Loader:
allocates a block in Bank $00 equal in size to the direct-page/stack load segment
LOADER
Memory Bank $00
Direct page/ stack
4. ProDOS 16:
sets the stack register to the highest address in the segment
sets the direct register to the lowest address in the segment

Figure 6-1. Automatic direct-page/stack allocation

When the program is linked, it is important that the direct-page/stack segment not be combined with any other object segments into a load segment—the linker must create a single load segment corresponding to the direct-page/stack object segment. If there is no direct-page/stack object segment, the linker will not create a corresponding load segment.

Allocation at run time

Each time the program is started (either through Initial Load or Restart), the System Loader looks for a direct-page/stack load segment. If it finds one, it determines the segment's size from from the `LENGTH` field in the segment's header. The System Loader then calls the Memory Manager to allocate a page-aligned, locked, dynamic memory block of that size in bank $00. The loader passes the base address and size of that space, along with the program's UserID and starting address, to ProDOS 16. ProDOS 16 sets the A (accumulator), D (direct) , and S(stack) registers as shown, then passes control to the program:

A = UserID assigned to the program
D = address of the first (lowest) byte in the direct-page/stack space
S = address of the last (highest) byte in the direct-page/stack space

By this convention, direct-page addresses are offsets from the base 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 and tested to make sure this cannot occur.

When your program terminates with a `QUIT` call, the System Loader's Application Shutdown function purges the direct-page/stack segment along with the program's other dynamic segments. The stack and direct page are therefore *not* preserved between program starts; they must be reallocated each time the program is run.

Note: There is no provision for extending or moving the direct-page/stack space after its initial allocation. Because bank $00 is so heavily used, the space you request may be unavailable—the memory adjoining your stack is likely to be occupied by a locked memory block. Make sure that the amount of space you specify at link time fills all your program's needs.

ProDOS 16 default stack and direct page

If the loader finds no direct-page/stack segment in a file at load time, it still returns the program's UserID and starting address to ProDOS 16, but it does not call the Memory Manager to allocate a direct-page/stack space and it returns zeros as the base address and size of the space. ProDOS 16 then calls the Memory Manager itself, and allocates a 1K direct-page/stack segment with the following attributes:

size:	1,024 bytes
owner:	program with the UserID returned by the loader
fixed/movable:	fixed
locked/unlocked:	locked
purge level:	1
may cross bank boundary?	no
may use special memory?	yes
alignment:	page-aligned
absolute starting address?	no
fixed bank?	yes—bank $00

See *Cortland Toolbox Reference* for a general description of memory block attributes assigned by the Memory Manager.

Once allocated, the default direct-page/stack is treated just it would be if it had been specified by the program: ProDOS 16 sets the A, D, and S registers before handing control to the program, and at shutdown time the System Loader purges the segment.

Manual allocation of stack and direct page

You (your program, that is) may allocate your own stack and direct-page space at run time, if you prefer. When ProDOS 16 transfers control to you, be sure to save the UserID value left in the accumulator before doing the following:

1. Using the starting or ending address left in the D or S register by ProDOS 16, make a `FindHandle` call to the Memory Manager, to get the memory handle of the automatically-provided direct-page/stack space. Then, using that handle, get rid of the space with a `DisposeHandle` call.
2. You can now allocate your own direct-page/stack space through the Memory Manager `NewHandle` call. Make sure that the allocated block is *purgeable*, *unmovable*, and *locked.*
3. Place the appropriate values (beginning and end addresses of the segment) in the D and S registers.

Managing system resources

Various hardware and software features of the Cortland provide information to an application or increase its flexibility. This section suggests ways to use those features.

Global variables

Under ProDOS 8, a fixed-address **Global Page** maintains the values of important variables and addresses for system programs and applications. The Global Page is at the same address in any machine or machine configuration that supports ProDOS 8, so an application can always access those variables at the same addresses.

ProDOS 16 does not maintain a Global Page. Such a set of fixed locations is inconsistent with the flexible and dynamic memory management system of the Cortland. Instead, calls to ProDOS 16, tools or firmware will give you the information formerly provided by the Global Page. Table 6-1 shows the Cortland calls used to obtain information equivalent to ProDOS 8 Global Page values.

Table 6-1. Cortland equivalents to ProDOS 8 global page information

Global Page Information	Cortland Equivalent
Global page entry points	(not supported)
Device driver vectors	(not supported)
List of active devices	returned by `VOLUME` call (ProDOS 16)
Memory Map	(responsibility of the Memory Manager)
Pointers to I/O buffers	returned by `OPEN` call (ProDOS 16)
Interrupt vectors	returned by `ALLOC_INTERRUPT` call (ProDOS 16)
Date/Time	returned by `ReadTime` call (Misc. Toolset)
System Level	returned by `GET_LEVEL` call (ProDOS 16)
MACHID	(not supported)
System Program version	(not supported)
ProDOS 16 Version	returned by `GET_VERSION` call (ProDOS 16)

Prefixes

The three predefined prefixes and five user-definable prefixes offer not only convenience in coding pathnames, but flexibility in writing for different system and application disk volumes. For example, any files on the boot disk can always be accessed through the prefix `0/`, regardless of the boot volume name. Any library routine in the system library subdirectory (normally the `SYSTEM/LIBS` subdirectory of the boot volume) will have the prefix `2/`, regardless of which system disk is on line. If you put libraries specific to your application in the same subdirectory as your application, they can always be called with the prefix `1/`, regardless of what subdirectory or disk your program inhabits.

Of course, your application can always change the values of any of the prefixes. For example, it may change prefix `2/` if it wishes to access libraries on a system volume other than the boot volume.

Native mode and emulation mode

As noted in Chapter 6, you can make ProDOS 16 calls from either emulation or native mode. Thus if part of your program requires the machine to be in emulation mode, you needn't reset it to native mode before calling ProDOS 16. Emulation-mode programs in the Cortland must be located in bank $00.

ProDOS 8 programs run entirely in emulation mode. If you wish to modify a ProDOS 8 program to run under ProDOS 16, or if you wish to use Cortland features available only in native mode, see "Revising a ProDOS 8 Application for ProDOS 16" in this chapter. See also *Programmer's Introduction to the Cortland.*

Setting initial machine configuration

When a Cortland application(type $B3) is first launched, the Cortland is in full native mode with all shadowing off (see Chapter 5). If your program needs a different machine configuration, it must make the proper settings. . .***information not yet available***

Allocating memory

All memory allocation is done through calls to the Memory Manager, described in *Cortland Toolbox Reference*. Memory space you request may be either movable or nonmovable. If it is movable, you access it through a memory handle; if it is nonmovable, you may access it through a handle or through a pointer. Since the Memory Manager does not return a pointer to an allocated block, you obtain the pointer by dereferencing the handle (see Chapter 3).

ProDOS 16 parameter blocks are referenced by pointers; if you do not code them into your program segments and reference them with labels, you must put them in nonmovable memory blocks. To allocate space for a parameter block, request a memory block of sufficient size from the Memory Manager, and specify that it be nonmovable, or else movable but **locked**. Obtain a pointer to the space by dereferencing its memory handle, and use that pointer in your ProDOS 16 call block. Since the memory block is locked, the pointer will always be valid.

Loading another program

If your program does not wish to load another program, it should use a ProDOS 16 QUIT call with no parameters. That normally brings up a program selector which allows the user to choose the next program to load. Most applications function this way.

However, if you want your application to load and execute another application, there are several ways to do it. If you wish to pass control to permanently to another application, use the ProDOS 16 QUIT call with a pathname pointer, as described in Chapter 5. By using the return flag parameter in the ProDOS 16 QUIT call, your program can function similarly to a shell—whenever it quits to another specified program, control will eventually return to it.

If you wish to load but not necessarily pass control to another program, or if you want your program to remain in memory after it passes control to another program, use the System Loader's Initial Load function (described in Chapter 18). When your program actively loads other program files, it is called a **controlling program**; the CPW Shell (see next section) is a controlling program. Chapter 17 presents suggestions for writing controlling programs.

You may load a ProDOS 8 application (type $FF) through the ProDOS 16 `QUIT` call, but you cannot do so with the System Loader's Initial Load call; the System Loader will load only ProDOS 16 load files (types $B3-$BF).

> **Note:** Because ProDOS 8 will not load type $B3 files, ProDOS 8-based applications that load and run other applications cannot run any newer ProDOS 16 applications. This restriction is a natural consequence of the lack of downward compatibility. If you wish to modify an older application to be able to use it with ProDOS 16, see "Revising a ProDOS 8 Application for ProDOS 16," later in this chapter.

Using interrupts

ProDOS 16 provides conventions (see Chapter 7) to ensure that interrupt-handling routines will function correctly. If you are writing a print spooler, game, communications program or other routine that uses interrupts, please follow those conventions.

As explained in Chapter 4, an *unclaimed interrupt* causes a system failure: control is passed to the System Death Manager and execution halts Your program may pass a message to the System Death manager to display on the screen when that happens. In addition, because the System Death Manager is a tool, and because all tools may be replaced by user-written routines, you may substitute your own error handler for unclaimed interrupts. See *Cortland Toolbox Reference* for information on the System Death Manager and for instructions on writing your own Tool Set.

If ProDOS 16 is called while it is in the midst of another call, it issues a "ProDOS is busy" error. This situation normally arises only when an interrupt handler makes ProDOS 16 calls; a typical application will nearly always find ProDOS 16 free to accept a call. Chapter 7 provides instructions on how to avoid this error when writing interrupt handlers; nevertheless, *all programs* should be able to handle the "ProDOS is busy" error code in case it occurs.

File creation/modification date and time

The information in this section is important to you if you are writing a file or disk utility program, or any routine that copies files.

All ProDOS 16 files are marked with the date and time of their creation. When a file is first created, ProDOS 16 stamps the file's directory entry with the current date and time on the system clock. If the file is later modified , ProDOS 16 then stamps it with a modification date and time (its creation date and time remain unchanged).

The creation and modification fields in a file entry refer to the *contents* of the file. The values in these fields should be changed only if the contents of the file change. Since data in the file's directory entry itself are not part of the file's contents, the modification field should not be updated when another field in the file entry is changed, *unless* that change is due to an alteration in the file's contents. For example, a change in the file's name is not a modification; on the other hand, a change in the file's EOF always reflects a change in its contents and therefore is a modification.

Remember also that a file's entry is a part of the contents of the directory or subdirectory that contains that *entry*. Thus, whenever a file entry is changed in any way (whether or not its modification field is changed), the modification fields in the entries for all its enclosing subdirectories—including the volume directory—must be updated.

Finally, when a file is *copied*, a utility program must be sure to give the copy the same creation and modification date and time as the original file, and *not* the date and time at which the copy was created.

To implement these concepts, file utility programs should note the following procedures:

1. **To create a new file:**
   a. Set the creation and modification fields of the file's entry to the current system date and time.
   b. Set the modification fields in the entries of all subdirectories in the path containing the file to the current system date and time.
2. **To rename a file:**
   a. Do not change the file's modification field.
   b. Set the modification fields of all subdirectories in the path containing the file to the current system date and time.
3. **To alter the contents of a file:**
   a. ProDOS 16 considers a file's contents to have been modified if any `WRITE` or `SET_EOF` operation has been performed on the file while it is open. If that condition has been met, set the file's modification field to the current system date and time when the file is closed.
   b. Also set the modification fields of all subdirectories in the path containing the file to the current system date and time.
4. **To delete a file:**
   a. Delete the file's entry from the directory or subdirectory that contains it.
   b. Set the modification fields of all subdirectories in the path containing the deleted file to the current system date and time.
5. **To copy a file:**
   a. Make a `GET_FILE-INFO` call on the source file (the file to be copied), to get its creation and modification dates and times.
   b. Make a `CREATE` call to create the destination file (the file to be copied to). Give it the creation date and time values obtained in step (a).
   c. Open both the source and destination files. Use `READ`s and `WRITE`s to copy the source to the destination. Close both files.

Note: The procedure for copying sparse files is more complicated than this. See Appendix A.

d. Make a `SET_FILE-INFO` call on the destination file, using all the information returned from `GET_FILE INFO` in step (a). This sets the modification date and time values to those of the source file.

ProDOS 16 automatically carries out all steps in procedures (1) through (4). Procedure (5) is the responsibility of the file copying utility.

Revising a ProDOS 8 application for ProDOS 16

If you have written a ProDOS 8-based program for the Apple II Plus, IIe, or IIc, it will run unchanged on the Cortland. The only noticeable difference will be its faster execution because of the greater clock speed of the Cortland. However, the program will not be able to take advantage of any advanced Cortland features such as large memory, the Toolbox, the mouse-based interface, and new graphics and sound abilities. This section discusses some of the basic alterations necessary to upgrade a ProDOS 8 application for native mode execution under ProDOS 16 on the Cortland.

Because ProDOS 16 closely parallels ProDOS 8 in function names and calling structure, it is not difficult to change system calls from one ProDOS to the other. But several other aspects of your program also must be redesigned if it is to run in native mode under ProDOS 16. Depending on the program's size and structure and the new features you wish to install, those changes may range from minor to drastic.

Memory management

Because the Cortland supports segmented load files, one of the first decisions to make is whether and how to segment the program (both the original program and any added parts), and where in memory to put the segments.

To help decide where in memory to place pieces of your program, consider that execution speed is related to memory loacation: banks $E0 and $E1 are slow memory, and all the other banks are fast memory (see Figure 3-1). Those parts of your program that are executed most often should probably go into fast memory, while less-used parts and data segments may be appropriate in slow memory. Program segments that make heavy use of I/O instructions might work best in slow memory; however,performance testing of the completed program is the only way to accurately determine where segments should go.

Bank zero is available for stack, direct page, and whatever other code or data your program wants in that space. Bank zero space is limited, however, and other software will need parts of it also. Still, bank zero's nearly 32K of total available direct-page/stack space is much larger than the 512 bytes available for stack and zero page under ProDOS 8.

Memory management methods are completely different under ProDOS 16 than under proDOS 8. If your ProDOS 8 program manages memory by allocating its own memory space and marking it off in the global page bit map, the ProDOS 16 version must be altered so that it requests all needed space from the Memory Manager. Whereas ProDOS 8 does not check to see if you are using only your marked-off space, the Memory Manager under

ProDOS 16 will not assign to your program any part of memory that has already been allocated.

Hardware configuration

ProDOS 8 applications run only in 6502 emulation mode on the Cortland. That does not mean that applications converted to run under ProDOS 16 must necessarily run in native mode. There are at least three configurations possible:

- The program may run in emulation mode, but make ProDOS 16 calls.
- The program may run in native mode with the m- and x-bits set. The accumulator and index registers will remain 8 bits wide.
- The program may run in full native mode (m- and x-bits cleared).

Modifying a program for the first configuration probably involves the least effort, but returns the least benefit.

Modifying a program to run in full native mode is the most difficult, but it makes best use of all Cortland features.

Converting system calls

For most ProDOS 8 calls, there is an equivalent ProDOS 16 call with the same name. Each call block must be modified for ProDOS 16: the `JSR` replaced with a `JSL`, the call number field made 2 bytes long, and the parameter list pointer made 4 bytes long. The only other conversion required is the reconstruction of the parameter block to the ProDOS 16 format.

For other ProDOS 8 calls, the ProDOS 16 equivalent performs a slightly different task, and the original code will have to be changed to account for that. For example, in ProDOS 8 an `ON_LINE` call can be used directly to determine the names of all online volumes; in ProDOS 16 a succession of `VOLUME` calls is required. Refer to the detailed descriptions in Chapters 9 through 13 to see which ProDOS 16 calls are different from their ProDOS 8 counterparts.

Still other ProDOS 8 calls have no equivalent in ProDOS 16. They are listed and described under "Eliminated ProDOS 8 System Calls," in Chapter 1. If your program uses any of these calls, they will have to be replaced as shown.

Modifying interrupt handlers

If you have written an interrupt handling routine, it needs to be updated to conform with the ProDOS 16 interrupt handling conventions (Chapter 7). There are very few changes necessary: making it return with an `RTL` rather than an `RTS` may be the only modification you need.

Compilation/assembly

Once your source code has been modified and augmented as desired, you need to recompile/reassemble it. You must use an assembler or compiler that produces object files in Cortland Object Module Format (OMF); otherwise the program cannot be properly linked and loaded for execution. Using a different compiler or assembler may mean that, in addition to modifying your program code, you might have to change some assembler directives to follow the syntax of the new assembler.

If you have been using the EDASM assembler, you will not be able to use it to write Cortland programs. The Cortland Programmer's Workshop is a set of development programs that allow you to produce and edit source files, assemble/compile object files, and link them into proper OMF load files. See "Cortland Programmer's Workshop" in this chapter.

After your revised program is linked, assign it the proper Cortland application file type ($B3-$BE).

Stack and direct page

The fixed stack and zero-page locations provided for your program by ProDOS 8 are not available under ProDOS 16. You may either let ProDOS 16 assign you a default 1,024-byte space, or you may define a direct-page/stack segment in your object code. In either case, the segment may be anywhere in bank $00. See "Stack and Direct Page," in this Chapter.

Cortland Programmer's Workshop

The Cortland Programmer's Workshop (CPW) is a powerful set of development programs designed to facilitate the creation of Cortland applications. If you are planning to write programs for the Cortland, CPW will make your job much easier. The Workshop includes the folowing components:

- Shell
- Editor
- Linker
- Debugger
- Assembler
- C Compiler

All these components work together (under the Shell) to speed the writing, compiling or assembling, and debugging of programs. The Shell acts as a command interpreter and an interface to ProDOS 16, providing several operating system functions and file utilities that can be called by users and by programs running under the Shell.

See the following manuals for more information on the Cortland Programmer's Workshop:

- *Cortland Programmer's Workshop Reference* (describes the Shell, Editor, Linker, and Debugger)
- *Cortland Programmer's Workshop Assembler Reference*
- *Cortland Programmer's Workshop C Reference*
- *Cortland Programmer's Workshop Pascal Reference*

Human Interface Guidelines

All people who develop application programs for Apple computers are strongly encouraged to follow the principles presented in the Apple *Human Interface Guidelines*. That manual describes the **Desktop Interface** through which the computer user communicates with his computer and the applications running on it. This section briefly outlines a few of the human interface concepts; please refer to the manual for specific design information.

The Apple Desktop Interface, first introduced with the Macintosh™ computer, is designed to appeal to an audience of nonprogrammers. Whatever the purpose or structure of your application, it will comunicate with the user in a consistent, standard, and non-threatening manner if it adheres to the Desktop Interface standards. These are some of the basic principles:

Human control: Users should feel that they are controlling the program, rather than the reverse. Give them clear alternatives to select from, and act on their selections consistently.

Dialog: There should be a clear and friendly dialog between human and computer. Make messages and requests to the user in plain English.

Direct Manipulation and Feedback: The user's physical actions should produce physical results. When a key is pressed, place the corresponding letter on the screen. Use highlighting, animation, and dialog boxes to show users the possible actions and their consequences.

Exploration: Give the user permission to test out the possibilities of the program without worrying about negative consequences. Keep error messages infrequent. Warn the user when risky situations are approached.

Graphic design: Good graphic design is a key feature of the guidelines. Objects on the screen should be *simple* and *clear*, and they should have *visual fidelity* (that is, they should look like what they represent). *Icons* and *palettes* are common graphic elements that need careful design.

Avoiding modes: a **mode** is a portion of an application that the user has to formally enter and leave, and that restricts the operations that can be performed while it's in effect. By restricting the user's options, modes reinforce the idea that computers are unnatural and unfriendly. Use modes sparingly.

Device-independence: Make your program as hardware-independent as possible. Don't bypass the tools and resources in ROM—your program may become incompatible with future products and features.

Consistency: As much as possible, all applications sould use the same interface. Don't confuse the user with a different interface for each program.

Evolution: Consistency does not mean that you are restricted to using existing desktop features. New ideas are essential for the evolution of the Human Interface concept. If your application has a feature that is described in *Human Interface Guidelines*, you should implement it exactly as described; if it is something new, make sure it cannot be confused with an existing feature. It is better to do something completely different than to half agree with the guidelines.

Chapter 7

Adding Routines to ProDOS 16

This chapter discusses device-handling routines that may be used with ProDOS 16. Because such routines are directly connected to ProDOS 16 and interact with it at a low level, they are essentially transparent to applications and can be considered "part of" ProDOS 16. Two types of routines are discussed: interrupt handlers and block device (disk) drivers.

Interrupt handlers

The Cortland has extensive firmware interrupt support (see *Cortland Firmware Reference*). In addition, ProDOS 16 supports up to 16 user-installed interrupt handlers (see Chapter 5). If you write an interrupt handler, it should follow the conventions described here. Note also the precautions you must take if your handler makes operating system calls.

Interupt handler conventions

Interrupt handling routines written for the Cortland must follow certain conventions. The interrupt dispatcher will set the following machine state before passing control to an interrupt handler:

e = 0
m = 0
x = 0
i = 1
c = 1
speed = high

Before returning to ProDOS 16, the interrupt handler must restore the machine to the following state:

e = 0
m = 0
x = 0
i = 1
speed = high

In addition the c flag must be cleared (= 0) if the handler serviced the interrupt, and set (= 1) if the handler did not service the interrupt. The handler must return with an `RTL` instruction.

When an interrupt occurs, ProDOS 16 successively polls the installed interrupt handlers until one of them services it, as indicated by the value of the c flag when the routine returns to ProDOS 16. ProDOS 16 then switches back to emulation mode and performs an `RTI` to the Cortland firmware interrupt handling system. If *no* handler services the interrupt, it is an unclaimed interrupt (see Chapter 4).

Installing interrupt handlers

Interrupt handlers are installed with the `ALLOC_INTERRUPT` call and removed with the `DEALLOC_INTERRUPT` call. The ProDOS 16 interrupt dispatcher maintains an **interrupt vector table**, an array of up to 16 vectors to interrupt handlers. As each successive `ALLOC_INTERRUPT` call is made, the dispatcher adds another entry to the end of the table. Each time a `DEALLOC_INTERRRUPT` call is made to delete a vector from the table, the remaining vectors are moved toward the beginning of the array, filling in the gap. Interrupt handling routines are polled by ProDOS 16 in the order in which their vectors occur in the interrupt vector table.

There is no way to reorder interrupt vectors except by allocating and deallocating interrupts. Interrupts that occur often or require fast service should be allocated first, so their vectors will be near the beginning of the interrupt vector table. If you need *extremely* fast interrupt service, install your interupt handler directly in the Cortland firmware interrupt dispatcher, rather than through ProDOS 16. See *Cortland Firmware Reference* for further information.

Be sure to enable the hardware generating the interrupt only *after* the routine to handle it is allocated; likewise, disable the hardware *before* the routine is deallocated. Otherwise, a fatal unclaimed interrupt error may occur (see "Unclaimed Interrupts" in Chapter 4).

Making operating system calls during interrupts

ProDOS 16 is not reentrant. That is, it does not save its own state when interrupted. It therefore is illegal to make an operating system call while another operating system call is in progress; if a call is attempted, ProDOS 16 will return an error (number $06, "ProDOS is busy").

For applications this is not a problem; the operating system is almost always free to accept a call from them. Only routines that are started through interrupts (such as interrupt handlers and desk accessories) need to be careful not to call ProDOS 16 while it is busy.

If an interrupt handler needs to make an operating system call, it must

1. find out if the operating system is busy
2. if so, defer itself temporarily
3. return control to the operating system so that it may complete the current call
4. regain control and make its own system call

The **System Scheduler**, part of a ROM-based tool, allows interrupt handlers to follow these procedures. See *Cortland Toolbox Reference* for detailed information.

Device drivers

If a disk drive supplied by another manufacturer is to work with ProDOS 16, it must look and act (to the operating system) just like a drive supplied by Apple Computer. Its boot ROM must have certain values in certain places, and its driver routine must use certain direct-page locations for its call parameters.

> **Note:** Because ProDOS 16 does not support character devices, only block device drivers are considered here.

Block device protocols

Like ProDOS 8, ProDOS 16 supports only one block I/O convention (the ProDOS protocol). Two other protocols, the Smartport and Extended Smartport protocols, are supported only to the extent that ProDOS 16 maintains their entry point; ProDOS 16 must do this because the entry point is part of the ProDOS protocol.

The current ProDOS block device I/O protocol is described in the following section. For compatibility purposes, that protocol is supported in the Cortland; when an application uses the protocol, ProDOS 16 makes the extra block moves necessary for multibank operation.

> **Note:** In following text, *ProDOS* refers to both ProDOS 16 and ProDOS 8. Where a certain feature applies only to one or the other, *ProDOS 16* or *ProDOS 8* is specified explicitly.

ProDOS block device protocol

ROM code conventions

During startup, ProDOS searches for all active block storage devices. For slot *n*, if it finds the following three bytes in that slot's ROM, ProDOS assumes it has found a disk drive:

$C*n*01	=	$20
$C*n*03	=	$00
$C*n*05	=	$03

ProDOS then checks the value of the byte at location $C*n*FF:

- If C*n*FF = $00, ProDOS assumes it has found a Disk II with 16-sector ROMs and marks its internal device driver table (ProDOS 16) or the device driver table in the ProDOS global page (ProDOS 8) with the address of the Disk II driver routines. The Disk II driver routines are part of ProDOS and support any drive that emulates Apple's 16-sector Disk II (280 blocks, single volume, and so on; see ***reference?***).
- If C*n*FF = $FF, ProDOS assumes it has found a Disk II with 13-sector ROMs. Because ProDOS does not support that format, it skips over the drive and continues its search.

- If ProDOS finds any value other than $00 or $FF at $C*n*FF, it assumes it has found an intelligent disk controller. If, in addition, the status byte at $C*n*FE indicates that the device supports `READ` and `STATUS` requests, ProDOS marks the device table (ProDOS 16) or global page (ProDOS 8) with a device-driver address whose high byte is equal to $C*n* and whose low byte is equal to the value found at $C*n*FF. In other words, the value at $C*n*FF is the adddress (offset from $C*n*00) of the beginning of the driver routine for slot *n*..

The special locations in ROM are:

$C*n*FC-C*n*FD The total number of blocks on the device. This information is used for writing the disk's bit map and directory header after formatting. If the value at this location is $0000, the number of blocks must be obtained by making a `STATUS` call to the device.

$C*n*FE The device's status byte. Bits 0 and 1 must be set or ProDOS will not install the driver vector in the global page.

Bit no.	**Significance if set (=1)**
7	the medium is removable
6	the device is interruptable
5-4	the number of volumes on the device (0 to 3)
3	the device supports formatting
2	the device can be written to
1	the device can be read from
0	the device's status can be read

$CnFF The low bye of the driver vector (the entry point to the driver routine). ProDOS places $C*n*00 plus the value of this byte in the global page.

Calls to the Driver

The only calls ProDOS makes to the disk driver are `STATUS`, `READ`, `WRITE`, and `FORMAT`. On receiving the `STATUS` call the driver should perform a check to verify that the device is ready for a `READ` or `WRITE`:

- If the device is not ready, the driver should set the carry bit and return the appropriate error code in the accumulator.
- If the device is ready, the driver should clear the carry, place a zero in the accumulator, and return the number of blocks on the device in the X and Y registers (low byte in X, high byte in Y).

If you wish to install more than two drives or volumes per slot, you may do so by installing a Smartport card in slot 5 (Apple II Plus, IIe, IIc) or by chaining them to port 5 (the Smartport port) on a Cortland. The Smartport card or port will accept up to 4 units, mapped as follows:

Unit number	**ProDOS slot and drive**
Smartport 1	slot 5, drive 1
Smartport 2	slot 5, drive 2
Smartport 3	slot 2, drive 1
Smartport 4	slot 2, drive 2

Call parameters

Call parameters are passed by ProDOS to the device driver at the following zero-page locations:

Address	Parameter	Explanation
$42	command number:	0 = `STATUS` request 1 = `READ` request 2 = `WRITE` request 3 = `FORMAT` request
$43	unit number:	The unit number is a byte value with the following format:

Bit:	7	6	5	4	3	2	1	0
Value:	Dr	Slot			(not used)			

where
Dr = drive number (0 or 1)
Slot = slot number (1–7)

Address	Parameter	Explanation
$44-$45	buffer pointer:	The address of the start of a 512-byte buffer in memory, to transfer the data to or from.
$46-$47	block number:	The block on disk to transfer the data to or from.

Format Note: the `FORMAT` code in the driver need only lay down whatever address marks may be required. It is the responsibility of the calling routine to write the virgin (empty) directory and bit map for the appropriate operating system.

Error codes

The device driver should report errors by setting the carry flag and loading the error code into the accumulator. Include at least these error codes:

Error Number	Message
$27	I/O error
$28	No device connected
$2B	Write-protected

Part II

ProDOS 16 System Call Reference

This part of the manual describes the ProDOS 16 system calls in detail. The calls are grouped into five categories:

• File housekeeping calls	(Chapter 9)
• File access calls	(Chapter 10)
• Device calls	(Chapter 11)
• Environment calls	(Chapter 12
• Interrupt control calls	(Chapter 13)

Chapter 8 shows how to make the calls, and explains the format for the call descriptions in Chapters 9 through 13. Chapter 14 is a complete list of ProDOS 16 error codes.

Chapter 8

Making ProDOS 16 Calls

Any independent program in the Cortland that makes system calls is known as a ProDOS 16 **calling program** or **caller**. The current application, a desk accessory, and an interrupt handler are examples of potential callers. A ProDOS 16 caller makes a system call by executing a **system call block**. The call block contains a pointer to a **parameter block**. The parameter block is used for passing information between the caller and the called function; additional information about the call is reflected in the state of certain hardware registers. This chapter discusses these aspects of system calls and compares them with the calling method used in ProDOS 8.

> **Note:** The phrase *system call* as used here is synonymous with *operating system call* or *ProDOS 16 call*, and is equivalent to "MLI call" for ProDOS 8. It includes all calls to the operating system for accessing system information and manipulating open or closed files. It is not restricted to what are called "system calls" in the *ProDOS Technical Reference Manual.*

The call block

A system call block consists of a `JSL` (jump subroutine long) to the ProDOS 16 entry point, followed by a 2-byte system call number and a 4-byte parameter block pointer. ProDOS 16 performs the requested function, if possible, and returns execution to the instruction immediately following the call block.

Any new applications written for Cortland under ProDOS 16 must use the system call block format. When making the call, the caller may have the processor in emulation mode or full native mode or any state in between (see *Technical Introduction to the Cortland*). The call block looks like this:

```
PRODOS      GEQU $E100A8        ; fixed entry vector
            .
            .
            .
            JSL  PRODOS 8       ; 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) block of memory.

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 function it calls, as well as information returned by the function to the caller.

Every ProDOS 16 call requires a valid parameter block (`PARMBLOCK` in the example just given), referenced by a 4-byte pointer in the call block. The caller is responsible for constructing the parameter block for each call it makes; the list may be anywhere in memory. Formats for individual parameter blocks accompany the detailed system call descriptions in Chapters 9 through 13.

Types of parameters

Each field in a parameter block contains a single parameter. There are three types of parameters: values, results, and pointers. Each is either an *input* to ProDOS 16 from the caller, or an *output* from ProDOS 16 to the caller.

- A **value** is a numerical quantity, 1 or more words long, that the caller passes to ProDOS 16 through the parameter block. It is an *input* parameter.
- A **result** is a numerical quantity, 1 or more words long, that ProDOS 16 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 ProDOS 16 can receive or place data. The pointer itself is an input for all ProDOS 16 calls; 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.

> **Note:** A **handle** is a special type of pointer; it is a pointer to a pointer. It is the 4-byte address of a location that *itself* contains the address of a location containing data, code, or buffer space. ProDOS 16 uses a handle parameter only in the `OPEN` call (Chapter 10); in that call the handle is an *output* (result).

Parameter block format

All parameter fields that contain block numbers, block counts, file offsets, byte counts, and other file or volume dimensions are 4 bytes long. Requiring 4-byte fields ensures that ProDOS 16 will accommodate future large devices using **guest file systems.**

All parameter fields contain an even number of bytes, for ease of manipulation by the16-bit 65816 processor. Thus pointers, for example, are 4 bytes long even though 3 bytes are sufficient to address any memory location. Wherever such extra bytes occur they must be set to zero by the caller; if they are not, compatibility with future versions of ProDOS 16 will be jeopardized.

Pointers in the parameter block must be written with the low byte of the low word at the lowest address.

Comparison of ProDOS 16 parameter blocks with their ProDOS 8 counterparts reveals that in some cases the order of parameters is slightly different. These alterations have been made to facilitate sharing a single parameter block among a number of calls. For example, most file access calls can be made with a single parameter block for each open file; under ProDOS 8 this sharing of parameter blocks is not possible.

> **Important:** A parameter's field width in a ProDOS 16 parameter block is often very different from the range of permissible values for that parameter. The fact that all fields are an even number of bytes is one reason. Another reason is that certain fields are larger than presently needed in anticipation of the requirements of future guest file systems. For example, the ProDOS 16 `CREATE` call's parameter block includes a 4-byte `aux_type` field, even though, *on disk*, the `aux_type` field is only 2 bytes wide (see "Format of Directory Files" in Appendix A). The two high-order bytes in the field must therefore *always* be zero.
>
> Ranges of permissible values for all parameters are given as part of the system call descriptions in the following chapters. When coding a parameter block, note carefully the range of permissible values for each parameter, and make sure that the value you assign is within that range.

Setting up a parameter block in memory

Each ProDOS 16 call uses a 4-byte pointer to point to its 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 block holding such a parameter block will be.

There are two ways to set up a ProDOS 16 parameter block in memory:

1. Code the block directly into the program, referencing it with a label. This is the simplest and most typical way to do it. The parameter block will always be correctly referenced, no matter where in memory the program code is loaded.
2. Use Memory Manager and System Loader calls to place the block in memory:
   a. Request a memory block of the proper size from the Memory Manager. Use the procedures described in *Cortland Toolbox Reference.* The block should be either *nonmovable* or *locked.*
   b. Obtain a pointer to the block, using the memory handle returned by the Memory Manager. **Dereference** the block's memory handle (that is, read its contents and use that value as a pointer to the block).
   c. Set up your parameter block, starting at the address pointed to by the pointer obtained in step (b).

Register values

There are no register requirements on entry to a ProDOS 16 call. ProDOS 16 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 ProDOS 16 initially saves, and then restores when finished, the value the register had just before the `JSL PRODOS 8` instruction.

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	unchanged

Note: ProDOS 16 treats several flags differently than ProDOS 8. The n and z flags are undefined here; under ProDOS 8, they are set according to the value in the accumulator. Here the caller may check the c flag to see if an error has occurred; under ProDOS 8, both the c and z flags determine error status.

Comparison with the ProDOS 8 call method

With the exceptions noted in Chapter 1, ProDOS 16 provides an identical call for each ProDOS 8 system call. The ProDOS 16 call performs exactly the same function as its ProDOS 8 equivalent, but it is in a format that fits the Cortland environment:

- As in ProDOS 8, the system call is issued through a subroutine jump to a fixed system entry point. However, the jump instruction is a `JSL` (*long* jump to subroutine) rather than a `JSR`, and it is to a.location in bank $E1, rather than bank $00.
- The parameter block pointer in the system call is 4 bytes long rather than 2, so the parameter block may be anywhere in memory.
- All memory pointer fields within the parameter block are also 4 bytes long, so they may reference data anywhere in memory.
- All 1-byte parameters are extended to 1 word in length, for efficient manipulation in 16-bit processor mode.
- All file-position (such as EOF) and block-specification (such as block number or block count) fields in the parameter block are 4 bytes long, in anticipation of future "guest file systems" that may support files larger than 16 Mb or volumes larger than 32 Mb.

Note: Although only 3 bytes are needed for memory pointers and block numbers in the Cortland, 4-byte pointers are used for ease of programming. The high byte in each case is reserved and must be zero.

The ProDOS 16 Exerciser

To help you learn to make ProDOS 16 calls, there is a small program called the ProDOS 16 *Exerciser* included with this manual. It allows you to execute system calls from a menu, and examine the results of your calls. It has a hexadecimal memory editor for reviewing and altering the contents of memory buffers, and it includes a catalog command.

When you use the *Exerciser* to make a ProDOS 16 call, you first request the call by its call number and then specify its parameter list, just as if you were coding the call in a program. The call is executed when you press Return. You may then use the memory editor or catalog command to examine the results of your call.

Complete instructions for using the ProDOS 16 *Exerciser* program are in Appendix C.

Format for system call descriptions

The following five chapters list and describe all ProDOS 16 operating system functions that may be called by an application. They are divided into five categories:

- File housekeeping calls
- File access calls
- Device calls
- Environment calls
- Interrupt control calls

Each description includes these elements:

- the function's name and call number
- a short explanation of its use
- a diagram of its required parameter block
- a detailed description of all parameters in the parameter block
- a list of all possible operating system error messages.

The parameter block diagram accompanying each call's description is a simplified representation of the parameter block in memory. The width of the diagram represents one byte; the numbers down the left side represent byte offsets from the base address of the parameter block. Each parameter field is further identified as containing a value, result, or pointer.

The detailed parameter description that follows the diagram has the following headings:

- **Offset:** The position of the parameter (relative to the block's base address)
- **Label:** The suggested assembly-language label for the parameter
- **Description:** Detailed information on the parameter, including:

parameter name:	The full name of the parameter.
size and type:	The size of the parameter (word or long word), and its classification (value, result, or pointer). A word is 2 bytes; a long word is 4 bytes.
range of values:	The permissible range of values of the parameter. A parameter may have a range much smaller than its size in bytes.
	Any additional explanatory information on the parameter follows.

Chapter 9

File Housekeeping Calls

These calls might also be called "closed-file" calls; they are made to get and set information about files that need not be open when the calls are made. They do not alter the *contents* of the files they access.

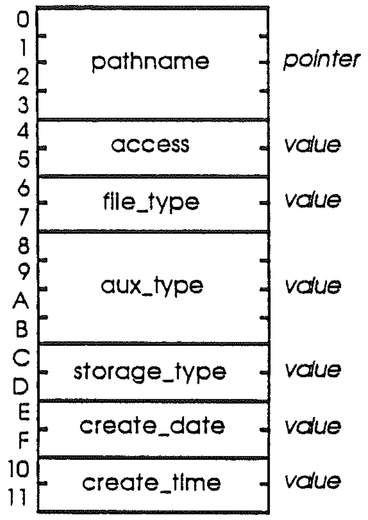
The ProDOS 16 file housekeeping calls are described in this order:

Number	**Function**	**Purpose**
$01	`CREATE`	creates a new file
$02	`DESTROY`	deletes a file
$04	`CHANGE_PATH`	changes a file's pathname
$05	`SET_FILE_INFO`	assigns attributes to a file
$06	`GET_FILE_INFO`	returns a file's attributes
$08	`VOLUME`	returns the volume on a device
$09	`SET_PREFIX`	assigns a pathname prefix
$0A	`GET_PREFIX`	returns a pathname prefix
$0B	`CLEAR_BACKUP_BIT`	zeroes a file's backup attribute

CREATE ($01)

Every disk file except the volume directory file (and any UCSD Pascal region on a partitioned disk) must be created with this call. It establishes a new directory entry for an empty file.

Parameter Block:



Offset	Label	Description

$00–$03 `pathname`

parameter name: pathname
size and type: long word pointer (high-order byte zero)
range of values: $0000 0000–$00FF FFFF

The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the pathname of the file to create.

$04–$05 `access`

parameter name: access
size and type: word value (high-order byte zero)
range of values: $0000–$00E3 with exceptions

A word whose low-order byte determines how the file may be accessed. The access byte's format is

Bit:	7	6	5	4	3	2	1	0
Value:	D	RN	B	reserved			W	R

where D = destroy-enable bit
RN = rename-enable bit
B = backup-needed bit
W = write-enable bit
R = read-enable bit

and for each bit, 1 = enabled, 0 = disabled. Bits 2 through 4 are reserved and must always be set to zero (disabled). The most typical setting for the access byte is $C3 (11000011).

$06–$07 `file_type`

parameter name: file type
size and type: word value (high-order byte zero)
range of values: $0000–$00FF

A number that categorizes the file by its contents (such as text file, binary file, ProDOS 16 system file). Currently defined file types are listed in Appendix A.

$08–$0B `aux_type`

parameter name: auxiliary type
size and type: long word value (high-order word zero)
range of values: $0000 0000–$0000 FFFF

A number that indicates additional attributes for certain file types. Definitions of all currently recognized auxiliary types and a list of the file types that use the auxiliary type field are given in Appendix A.

$0C–$0D `storage_type`

parameter name: storage type
size and type: word value/result (high-order byte zero)
range of values: $0000–$000D with exceptions

A number that describes the logical organization of the file (see Appendix A):

$00 = inactive entry
$01 = seedling file
$02 = sapling file
$03 = tree file
$04 = UCSD Pascal region on a partitioned disk
$0D = directory file

$01 and $0D are the most typical input values for this field in the CREATE call; any value in the range $00 through $03 is automatically converted to an input (and output) of $01.

Note: $0E and $0F are not valid storage types; they are subdirectory and volume key block identifiers.

$0E–$0F `create_date`

parameter name: Creation date
size and type: word value
range of values: limited range

The date on which the file was created. Its format is

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	Year							Month				Day				

If the value in this field is zero, ProDOS 16 supplies the date obtained from the system clock.

$10–$11 `create_time`	**parameter name:**	creation time
	size and type:	word value
	range of values:	limited range

The time at which the file was created. Its format is

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	0	0	0	Hour					0	0	Minute					

If the value in this field is zero, ProDOS 16 supplies the time obtained from the system clock.

Possible ProDOS 16 Errors

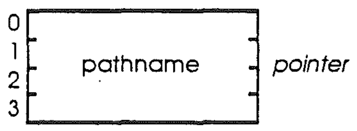
$07	ProDOS is busy
$10	Device not found
$27	I/O error
$2B	Disk write-protected
$40	Invalid pathname syntax
$44	Path not found
$45	Volume not found
$46	File not found
$47	Duplicate pathname
$48	Volume full
$49	Volume directory full
$4B	Unsupported storage type
$52	Unsupported volume type
$53	Invalid parameter
$58	Not a block device
$5A	Block number out of range

DESTROY ($02)

This function deletes the file specified by pathname. It removes the file's entry from the directory that owns it and returns the file's blocks to the volume bit map.

Volume directory files, files with unrecognized storage types, and open files cannot be destroyed. Subdirectory files must be empty before they can be destroyed.

Parameter Block:



Offset	Label	Description
$00–$03	`pathname`	**parameter name:** pathname **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the pathname of the file to delete.

Possible ProDOS 16 Errors

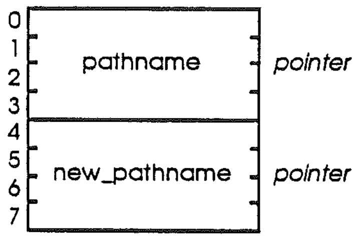
$07	ProDOS is busy
$10	Device not found
$27	I/O error
$2B	Disk write-protected
$40	Invalid pathname syntax
$44	Path not found
$45	Volume not found
$46	File not found
$4A	Version error
$4B	Unsupported storage type
$4E	Access: file not destroy-enabled
$50	File is open
$52	Unsupported volume type
$58	Not a block device
$5A	Block number out of range

CHANGE_PATH ($04)

This function performs an intravolume file move. It moves a file's directory entry from one subdirectory to another within the same volume—the file itself is never moved. The specified pathname and new pathname may be either full or partial pathnames in the same volume. See Chapter 5 for an explanation of partial pathnames.

If the two pathnames are identical except for the rightmost file name (that is, if both the old and new names are in the same subdirectory), this call produces the same result as the `RENAME` call in ProDOS 8.

Parameter Block:



Offset	Label	Description
$00–$03	`pathname`	**parameter name:** pathname **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the file's present pathname.
$04–$07	`new_pathname`	**parameter name:** new pathname **size and type:** long word pointer (high-order byte zero) **range of values:** 0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the file's new pathname.

Possible ProDOS 16 Errors

$07	ProDOS is busy
$27	I/O error
$2B	Disk write-protected
$40	Invalid pathname syntax
$44	Path not found
$45	Volume not found
$46	File not found
$47	Duplicate pathname
$4A	Version error

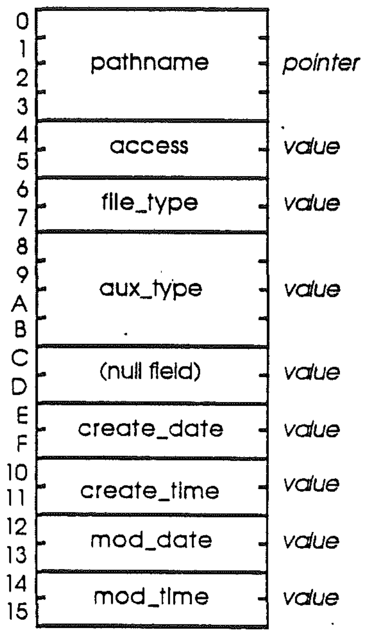
$4B	Unsupported storage type
$4E	Access: file not rename-enabled
$50	File is open
$52	Unsupported volume type
$57	Duplicate volume
$58	Not a block device

SET_FILE_INFO ($05)

This function modifies the information in the specified file's directory entry. The call can be made whether the file is open or closed; however, any changed access attributes are not recognized by an open file until the next time the file is opened. In other words, this call does not modify the accessibility of memory-resident information.

Note: ProDOS 16 ignores input values in the `create_date` and `create_time` fields of this function.

Parameter Block:



Offset	Label	Description
$00–$03	`pathname`	**parameter name:** pathname **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the file's pathname.
$04–$05	`access`	**parameter name:** access **size and type:** word value (high-order byte zero) **range of values:** $0000–$00E3 with exceptions A word whose low-order byte determines how the file may be accessed. The access byte's format is

Bit:	7	6	5	4	3	2	1	0
Value:	D	RN	B	reserved			W	R

where D = destroy-enable bit
RN = rename-enable bit
B = backup-needed bit
W = write-enable bit
R = read-enable bit

and for each bit, 1 = enabled, 0 = disabled. Bits 2 through 4 are reserved and must always be set to zero (disabled). The most typical setting for the access byte is $C3 (11000011).

$06–$07 `file_type`

parameter name: file type
size and type: word value (high-order byte zero)
range of values: $0000–$00FF

A number that categorizes the file by its contents (such as text file, binary file, ProDOS 16 system file). Currently defined file types are listed in Appendix A.

$08–$0B `aux_type`

parameter name: auxiliary type
size and type: long word value (high-order word zero)
range of values: $0000 0000–$0000 FFFF

A number that indicates additional attributes for certain file types. Definitions of all currently recognized auxiliary types and a list of the file types that use the auxiliary type field are given in Appendix A.

$0C–$0D `(null field)`

parameter name: (null field)
size and type: word value
range of values: (undefined)

Values in this field are ignored.

$0E–$0F `create_date`

parameter name: creation date
size and type: word value
range of values: limited range

The date on which the file was created. Its format is

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	Year							Month				Day				

(Values in this field are ignored.)

$10–$11 `create_time`

parameter name: creation time
size and type: word value
range of values: limited range

The time at which the file was created. Its format is

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	0	0	0	Hour					0	0	Minute					

(Values in this field are ignored.)

$12–$13 `mod_date`

parameter name: modification date
size and type: word value
range of values: limited range

The date on which the file was last modified. Its format is identical to the `create_date` format:

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	Year							Month				Day				

If the value in this field is zero, ProDOS 16 supplies the date obtained from the system clock.

$14–$15 `mod_time`

parameter name: modification time
size and type: word value
range of values: limited range

The time at which the file was last modified. Its format is identical to the `create_time` format:

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	0	0	0	Hour					0	0	Minute					

If the value in this field is zero, ProDOS 16 supplies the time obtained from the system clock.

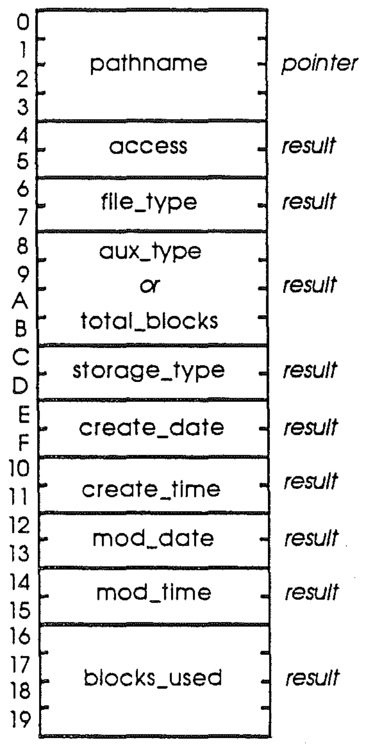
Possible ProDOS 16 Errors

$07	ProDOS is busy
$27	I/O error
$2B	Disk write-protected
$40	Invalid pathname syntax
$44	Path not found
$45	Volume not found
$46	File not found
$4A	Version error
$4B	Unsupported storage type
$4E	Access: file not write-enabled
$52	Unsupported volume type
$53	Invalid parameter
$58	Not a block device

GET_FILE_INFO ($06)

This function returns the information that is stored in the specified file's directory entry. The call can be made whether the file is open or closed. However, if you make the `SET_FILE_INFO` call to change the access byte of an open file, the access information returned by `GET_FILE_INFO` may not be accurate until the file is closed.

Parameter Block:



Offset	Label	Description
\$00–\$03	`pathname`	**parameter name:** pathname **size and type:** long word pointer (high-order byte zero) **range of values:** \$0000 0000–\$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the pathname..
\$04–\$05	`access`	**parameter name:** access **size and type:** word result (high-order byte zero) **range of values:** \$0000–\$00E3 with exceptions A word whose low-order byte determines how the file may be accessed. The access byte's format is

Bit:	7	6	5	4	3	2	1	0
Value:	D	RN	B	reserved			W	R

where D = destroy-enable bit
RN = rename-enable bit
B = backup-needed bit
W = write-enable bit
R = read-enable bit

and for each bit, 1 = enabled, 0 = disabled. Bits 2 through 4 are reserved and must always be set to zero (disabled). The most typical setting for the access byte is $C3 (11000011).

$06–$07 `file_type`

parameter name: file type
size and type: word result (high-order byte zero)
range of values: $0000–$00FF

A number that categorizes the file by its contents (such as text file, binary file, ProDOS 16 system file). Currently defined file types are listed in Appendix A.

$08–$0B `aux_type`

parameter name: auxiliary type
size and type: long word result (high-order word zero)
range of values: $0000 0000–$0000 FFFF

A number that indicates additional attributes for certain file types. Definitions of all currently recognized auxiliary types and a list of the file types that use the auxiliary type field are given in Appendix A.

or

`total_blocks`

parameter name: total blocks
size and type: long word result (high-order byte zero)
range of values: $0000 0000–$00FF FFFF

If the call is for a volume directory file, the total number of blocks on the volume is returned in this field.

$0C–$0D `storage_type`

parameter name: storage type
size and type: word result (high-order byte zero)
range of values: $0000–$000D with exceptions

A number that describes the logical organization of the file (see Appendix A):

$00 = inactive entry
$01 = seedling file
$02 = sapling file
$03 = tree file
$04 = UCSD Pascal region on a partitioned disk
$0D = directory file

Note: $0E and $0F are not valid storage types; they are subdirectory and volume key block identifiers.

$0E–$0F `create_date` **parameter name:** creation date
size and type: word result
range of values: limited range

The date on which the file was created. Its format is

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	Year							Month				Day				

$10–$11 `create_time` **parameter name:** creation time
size and type: word result
range of values: limited range

The time at which the file was created. Its format is

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	0	0	0	Hour					0	0	Minute					

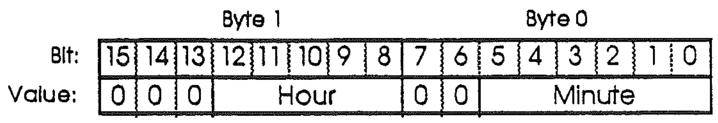
$12–$13 `mod_date` **parameter name:** modification date
size and type: word result
range of values: limited range

The date on which the file was last modified. Its format is identical to the `create_date` format:

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	Year							Month				Day				

$14–$15 `mod_time` **parameter name:** modification time
size and type: word result
range of values: limited range

The time at which the file was last modified. Its format is identical to the `create_time` format:



$16–$19 `blocks_used`	**parameter name:**	blocks used
	size and type:	long word result
	range of values:	$0000 0000–$FFFF FFFF

The total number of blocks used by the file. It equals the value of the `blocks_used` parameter in the file's directory entry.

or

The total number of blocks used by all files on the volume (if the call is for a volume directory).

Possible ProDOS 16 Errors

$07	ProDOS is busy
$27	I/O error
$40	Invalid pathname syntax
$44	Path not found
$45	Volume not found
$46	File not found
$4A	Version error
$4B	Unsupported storage type
$52	Unsupported volume type
$53	Invalid parameter
$58	Not a block device

VOLUME ($08)

When given the name of a device, this function returns:

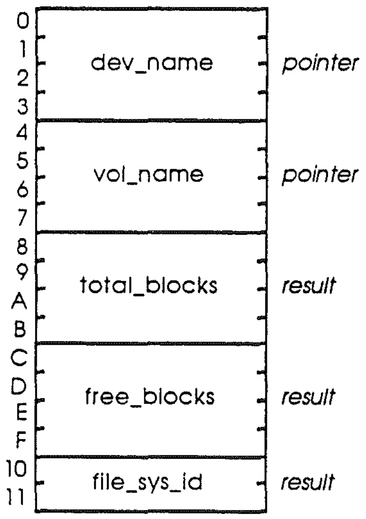
- the name of the volume that occupies that device
- the total number of blocks on the volume
- the current number of free (unallocated) blocks on the volume
- the file system identification number of the volume

The volume name is returned with a leading slash (`/`).

To generate a list of all mounted volumes (equivalent to calling `ON_LINE` in ProDOS 8 with a unit number of zero), call `VOLUME` repeatedly with successive device names (`.D1`, `.D2`, and so on). When there are no more online volumes to name, ProDOS 16 returns error $11 (Invalid device name).

> **Note:** Because ProDOS 16 cannot detect the difference between an empty device and a nonexistent device, in certain cases it assigns a device name where there is no device connected, just to make sure it hasn't skipped over an empty device. Therefore, in making `VOLUME` calls, you may occasionally find that there are more "valid" device names than there are devices on line.

Parameter Block:



Offset	Label		Description
$00–$03	`dev_name`	**parameter name:**	device name
		size and type:	long word pointer (high-order byte zero)
		range of values:	$0000 0000–$00FF FFFF

The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the device name.

$04–$07	`vol_name`	**parameter name:**	volume name
		size and type:	long word pointer (high-order byte zero)
		range of values:	$0000 0000–$00FF FFFF

The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the volume name (including a leading slash).

$08–$0B	`total_blocks`	**parameter name:**	total blocks
		size and type:	long word result (high-order byte zero)
		range of values:	$0000 0000–$00FF FFFF

The total number of blocks the volume contains.

$0C–$0F	`free_blocks`	**parameter name:**	free blocks
		size and type:	long word result (high-order byte zero)
		range of values:	$0000 0000–$00FF FFFF

The number of free (unallocated) blocks in the volume.

$10–$11	`file_sys_id`	**parameter name:**	file system ID
		size and type:	word result (high-order byte zero)
		range of values:	$0000–$00FF

A word whose low-order byte identifies the file system to which the specified file or volume belongs. The currently defined file system identification numbers are listed in Appendix A. They include

0 = {reserved}
1 = ProDOS/SOS
2 = DOS 3.3
3 = DOS 3.2, 3.1
4 = Apple II Pascal
5 = Macintosh
6 = Macintosh (HFS)
7 = LISA
8 = Apple CP/M
9-255 = {reserved}

Possible ProDOS 16 Errors

$07	ProDOS is busy
$10	Device not found
$27	I/O error
$2E	Disk switched: files open
$45	Volume not found
$4A	Version error
$52	Unsupported volume type
$55	Volume control block full
$57	Duplicate volume
$58	Not a block device

SET_PREFIX ($09)

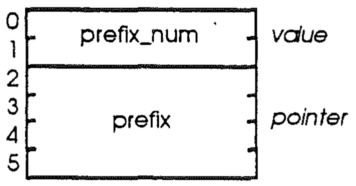
This function assigns the indicated directory to any of 8 prefixes. The prefixes are designated by a number followed by a slash: `0/`, `1/`, `2/`,..., `7/`. Three prefix designators have default values:

- `0/` is initially the ProDOS 16 system prefix—the name of the volume from which ProDOS 16 was booted.
- `1/` is initially the application subdirectory—the pathname of the subdirectory containing the currently running application.
- `2/` is initially the system library subdirectory—the subdirectory that contains the library modules used by the currently running application.

At any time, an application may change the pathnames assigned to any of the prefixes (including `0/`, `1/`, or `2/`). The pathname of the indicated directory specified in the `SET_PREFIX` call may be either a full or partial pathname—it itself may begin with a prefix designator. Specifying a pathname with a length of zero sets the designated prefix to null.

> **Note:** ProDOS 16 does *not* check to make sure that the designated volume is on line when you specify a prefix; it only checks the prefix string for correct syntax.

Parameter Block:



Offset	Label	Description
$00–$01	`prefix_num`	**parameter name:** prefix number **size and type:** word value **range of values:** $0000–$0007 One of the 8 prefix numbers, in binary (without a terminating slash).
$02–$05	`prefix`	**parameter name:** prefix **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing a directory pathname.

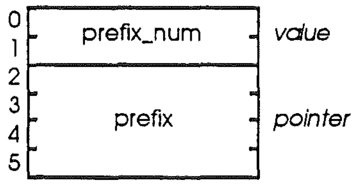
Possible ProDOS 16 Errors

$07	ProDOS is busy
$40	Invalid pathname syntax

GET_PREFIX ($0A)

This function returns any of the current prefixes (specified by number), placing it in the buffer pointed to by `prefix`. The returned prefix is bracketed by slashes (such as `/APPLE/` or `/APPLE/BYTES/`). If the requested prefix has been set to null (see `SET_PREFIX`), a count of zero is returned as the length byte in the prefix buffer.

Parameter Block:



Offset	Label	Description
$00–$01	`prefix_num`	**parameter name:** prefix number **size and type:** word value **range of values:** $0000–$0007 One of the 8 prefix numbers, in binary (without a terminating slash).
$02–$05	`prefix`	**parameter name:** prefix **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer, in which ProDOS 16 places a length byte followed by an ASCII string representing a directory pathname.

Possible ProDOS 16 Errors

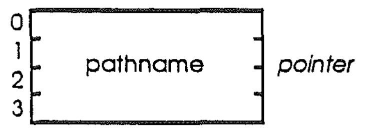
$07	ProDOS is busy
$53	Parameter out of range

CLEAR_BACKUP_BIT ($0B)

This is the only call that will clear the backup bit in a file's access byte. Once cleared, the bit indicates that the file has not been altered since the last backup. ProDOS 16 automatically resets the backup bit every time a file is altered.

Important: Only disk backup programs should use this function!

Parameter Block:



Offset	Label	Description
$00–$03	`pathname`	**parameter name:** pathname **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the file's pathname.

Possible ProDOS 16 Errors

$07	ProDOS is busy
$40	Invalid pathname syntax
$44	Path not found
$45	Volume not found
$46	File not found
$4A	Version error
$52	Unsupported volume type
$58	Not a block device

Chapter 10

File Access Calls

These might be called "open-file" calls. They are made to access and change the information *within* files, and therefore in most cases the files must be open before the calls can be made.

The ProDOS 16 file access calls are described in the following order:

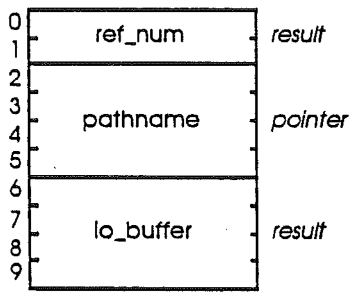
Number	**Function**	**Purpose**
$10	`OPEN`	prepares file for access
$11	`NEWLINE`	enables newline read mode
$12	`READ`	transfers data from file
$13	`WRITE`	transfers data to file
$14	`CLOSE`	ends access to file
$15	`FLUSH`	empties I/O buffer to file
$16	`SET_MARK`	sets current position in file
$17	`GET_MARK`	returns current position in file
$18	`SET_EOF`	sets size of file
$19	`GET-EOF`	returns size of file
$1A	`SET_LEVEL`	sets system file level
$1B	`GET_LEVEL`	returns system file level

OPEN ($10)

This function prepares a file to be read from or written to. It creates a file control block (FCB) that keeps track of the current characteristics of the file specified by pathname. It sets the current position in the file (MARK) to zero, and returns a reference number (`ref_num`) for the file; subsequent file access calls must refer to the file by its reference number. It also returns the address of a 1024-byte I/O buffer used by ProDOS 16 for reading from and writing to the file.

Up to 8 files may be open simultaneously.

Parameter Block:



Offset	Label	Description
$00–$01	`ref_num`	**parameter name:** reference number **size and type:** word result (high-order byte zero) **range of values:** $0000–$00FF An identifying number assigned to the file by ProDOS 16. It is used in place of the pathname in all subsequent file access calls.
$02–$05	`pathname`	**parameter name:** pathname **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the pathname of the file to open.
$06–$09	`io_buffer`	**parameter name:** I/O buffer **size and type:** long word result (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF A memory handle. It points to a location where the address of the I/O buffer allocated by ProDOS 16 is stored.

Possible ProDOS 16 Errors

$07	ProDOS is busy
$27	I/O error
$40	Invalid pathname syntax
$42	File control block table full
$44	Path not found
$45	Volume not found
$46	File not found
$4A	Version error
$4B	Unsupported storage type
$4E	Access: file not read-enabled
$50	File is open
$52	Unsupported volume type
$54	Out of memory
$57	Duplicate volume

NEWLINE ($11)

This function enables or disables the **newline** read mode for an open file. When newline is disabled, a `READ` call (described next) terminates only when the requested number of characters has been read (unless the end of the file is encountered first). When newline is enabled, the `READ` will also terminate when a newline character (as defined in the parameter block) is read.

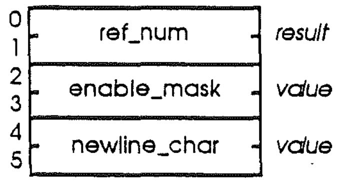
When a `READ` call is made and newline mode is enabled,

1. Each character read in is first transferred to the user's data buffer.
2. The character is ANDed with the low-order byte of the newline enable mask (specified in the `NEWLINE` call's parameter block).
3. The result is compared with the low-order byte of the newline character.
4. If there is a match, the read is terminated.

The enable mask is typically used to mask off unwanted bits in the character that is read in. For example, if the mask value is $7F (binary 0111 1111), a newline character will be correctly matched whether or not its high bit is set. If the mask value is $FF (1111 1111), the character will pass through the AND operation unchanged.

Newline read mode is disabled by setting the enable mask to $00.

Parameter Block:



Offset	Label		Description
$00–$01	`ref_num`	**parameter name:**	reference number
		size and type:	word result (high-order byte zero)
		range of values:	$0000–$00FF
			The identifying number assigned to the file by the `OPEN` function.
$02–$03	`enable_mask`	**parameter name:**	enable mask
		size and type:	word value (high-order byte zero
		range of values:	$0000–$00FF
			The current character is ANDed with the low order byte of this word.

$04–$05	`newline_char`	**parameter name:**	newline character
		size and type:	word value (high-order byte zero)
		range of values:	$0000–$00FF

Whatever character occupies the low-order byte of this field is defined as the newline character.

Possible ProDOS 16 Errors

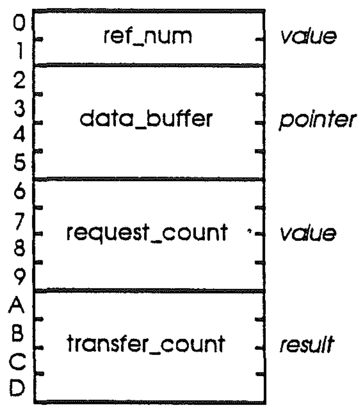
$07	ProDOS is busy
$43	Invalid reference number

READ ($12)

When called, this function attempts to transfer the requested number of bytes (starting at the current position of the file specified by *ref_num*) into the buffer pointed to by *data_buffer*. When finished, the function returns the number of bytes actually transferred.

No more than 16,777,215 ($FF FF FF) bytes may be read in a single call.

Parameter Block:



Offset	Label	Description	
$00–$01	`ref_num`	**parameter name:**	reference number
		size and type:	word value (high-order byte zero)
		range of values:	$0000–$00FF
		The identifying number assigned to the file by the `OPEN` function.	
$02–$05	`data_buffer`	**parameter name:**	data buffer
		size and type:	long word pointer (high-order byte zero)
		range of values:	$0000 0000–$00FF FFFF
		The long word address of a buffer. The buffer should be large enough to hold the requested data.	
$06–$09	`request_count`	**parameter name:**	request count
		size and type:	long word value (high-order byte zero)
		range of values:	$0000 0000–$00FF FFFF
		The number of bytes to be transferred.	
$0A–$0D	`transfer_count`	**parameter name:**	transfer count
		size and type:	long word result (high-order byte zero)
		range of values:	$0000 0000–$00FF FFFF

The actual number of bytes transferred.

Possible ProDOS 16 Errors

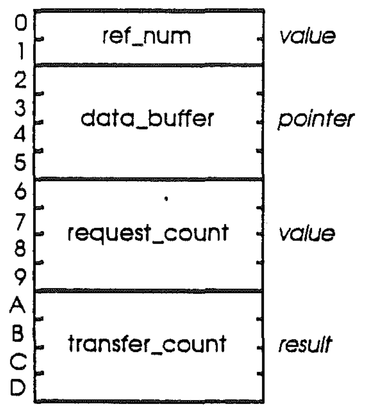
$07	ProDOS is busy
$27	I/O error
$43	Invalid reference number
$4C	EOF encountered (Out of data)
$4E	Access: file not read-enabled

WRITE ($13)

When called, this function attempts to transfer the specified number of bytes from the buffer pointed to by *data_buffer* to the file specified by *ref_num* (starting at the current position in the file). When finished, the function returns the number of bytes actually transferred.

No more than 16,777,216 ($FF FF FF) bytes may be written in a single call.

Parameter Block:



Offset	Label	Description	
$00–$01	`ref_num`	**parameter name:**	reference number
		size and type:	word value (high-order byte zero)
		range of values:	$0000–$00FF
		The identifying number assigned to the file by the `OPEN` function.	
$02–$05	`data_buffer`	**parameter name:**	data buffer
		size and type:	long word pointer (high-order byte zero)
		range of values:	$0000 0000–$00FF FFFF
		The long word address of a buffer. The buffer should be large enough to hold the requested data.	
$06–$09	`request_count`	**parameter name:**	request count
		size and type:	long word value (high-order byte zero)
		range of values:	$0000 0000–$00FF FFFF
		The number of bytes to be transferred.	
$0A–$0D	`transfer_count`	**parameter name:**	transfer count
		size and type:	long word result (high-order byte zero)
		range of values:	$0000 0000–$00FF FFFF

The actual number of bytes transferred.

Possible ProDOS 16 Errors

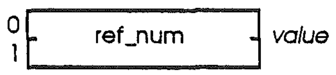
$07	ProDOS is busy
$27	I/O error
$2B	Disk write-protected
$43	Invalid reference number
$48	Volume full
$4E	Access: file not write-enabled
$5A	Block number out of range

CLOSE ($14)

This function is called to release all resources used by an open file. The file control block (FCB) is released; if necessary, the file's I/O buffer is emptied (written to disk) and the directory entry for the file is updated. Once a file is closed, any subsequent calls using its `ref_num` will fail (until that number is assigned to another open file).

If the specified `ref_num` is zero, all open files at or above the current file level (see `SET_LEVEL` and `GET_LEVEL` calls) are closed. For example, if files are open at levels 0, 1, and 2 and you have set the current level to 1, a `CLOSE` call with `ref_num` set to 0 will close all files at levels 1 and 2, but leave files at level 0 open.

Parameter Block:



Offset	Label		Description
$00–$01	`ref_num`	**parameter name:**	reference number
		size and type:	word value (high-order byte zero)
		range of values:	$0000–$00FF
			The identifying number assigned to the file by the `OPEN` function.

Possible ProDOS 16 Errors

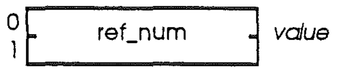
$07	ProDOS is busy
$27	I/O error
$2B	Disk write-protected
$43	Invalid reference number
$5A	Block number out of range

FLUSH ($15)

This function is called to empty an open file's buffer and update its directory. If *ref_num* is zero, all open files are flushed.

> **Note:** ProDOS 16 ignores *ref_num* in this call. The `FLUSH` call flushes all open files. The parameter is required for this call because future versions of ProDOS 16 will have the capability to flush individual files.

Parameter Block:



Offset	Label	Description
$00–$01	`ref_num`	**parameter name:** reference number **size and type:** word value (high-order byte zero) **range of values:** $0000–$00FF The identifying number assigned to the file by the `OPEN` function.

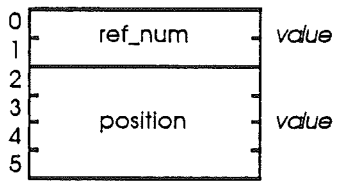
Possible ProDOS 16 Errors

$07	ProDOS is busy
$27	I/O error
$2B	Disk write-protected
$43	Invalid reference number
$48	Volume full
$5A	Block number out of range

SET_MARK ($16)

For the specified open file, this function sets the current position (MARK, the position at which subsequent reading and writing will occur) to the point specified by the position parameter. The value of the current position may not exceed EOF (end-of-file; the size of the file in bytes).

Parameter Block:



Offset	Label	Description
$00–$01	`ref_num`	**parameter name:** reference number **size and type:** word value (high-order byte zero) **range of values:** $0000–$00FF The identifying number assigned to the file by the `OPEN` function.
$02–$05	`position`	**parameter name:** position **size and type:** long word value (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The value assigned to MARK. It is the position, in bytes relative to the beginning of the file, at which the next read or write will occur.

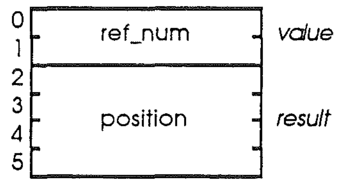
Possible ProDOS 16 Errors

$07	ProDOS is busy
$27	I/O error
$43	Invalid reference number
$4D	Position out of range
$5A	Block number out of range

GET_MARK ($17)

This function returns the current position (MARK, the position at which subsequent reading and writing will occur) for the specified open file.

Parameter Block:



Offset	**Label**		**Description**
$00–$01	`ref_num`	**parameter name:**	reference number
		size and type:	word value (high-order byte zero)
		range of values:	$0000–$00FF
			The identifying number assigned to the file by the `OPEN` function.
$02–$05	`position`	**parameter name:**	position
		size and type:	long word result (high-order byte zero)
		range of values:	$0000 0000–$00FF FFFF
			The current value of MARK. It is the position, in bytes relative to the beginning of the file, at which the next read or write will occur.

Possible ProDOS 16 Errors

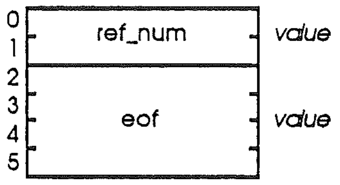
$07	ProDOS is busy
$43	Invalid reference number

SET_EOF ($18)

For the specified file, this function sets its logical size (in bytes) to the value spec[illegible]d by EOF (end-of-file). If the specified EOF is less than the current EOF, then disk b[illegible]ks past the new EOF are released to the system. However, if the specified EOF is equal [illegible]r greater than the current EOF, no new blocks are allocated until data are actually w[illegible]en to them.

The value of EOF cannot be changed unless the file is write-enabled.

Parameter Block:



Offset	Label		Description
$00–$01	`ref_num`	**parameter name:**	reference number
		size and type:	word value (high-order byte zer[illegible]
		range of values:	$0000–$00FF
		The identifying number assigned to the file by the [illegible] function.	
$04–$07	`eof`	**parameter name:**	end-of-file
		size and type:	long word value (high-order by[illegible]ero)
		range of values:	$0000 0000–$00FF FFFF
		The specified logical size of the file. It represents the [illegible] number of bytes that may be read from the file.	

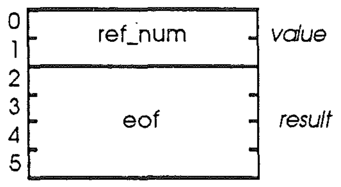
Possible ProDOS 16 Errors

$07	ProDOS is busy
$27	I/O error
$43	Invalid reference number
$4D	Position out of range
$4E	Access: file not write-enabled
$5A	Block number out of range

GET_EOF ($19)

For the specified open file, this function returns its logical size, or EOF (end-of-file; the number of bytes that can be read from it).

Parameter Block:



Offset	Label	Description
$00–$01	`ref_num`	**parameter name:** reference number **size and type:** word value (high-order byte zero) **range of values:** $0000–$00FF The identifying number assigned to the file by the `OPEN` function.
$04–$07	`eof`	**parameter name:** end-of-file **size and type:** long word result (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The current logical size of the file. It represents the total number of bytes that may be read from the file.

Possible ProDOS 16 Errors

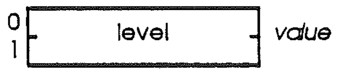
$07	ProDOS is busy
$43	Invalid reference number

SET_LEVEL ($1A)

This function sets the current value of the system file level (see Chapter 2). All subsequent `OPEN` calls will assign this level to the files opened. All subsequent `CLOSE` calls for *multiple* files (that is, those calls using a specified `ref_num` of zero) will be effective only on those files that were opened when the system level was greater than or equal to the new level.

The range of legal system level values is $0000–$00FF. The file level initially defaults to zero.

Parameter Block:



Offset	Label		Description
$00–$01	`level`	**parameter name:**	system file level
		size and type:	word value (high-order byte zero)
		range of values:	$0000–$00FF
		The specified value of the system file level.	

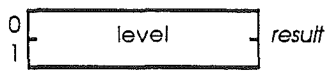
Possible ProDOS 16 Errors

$07	ProDOS is busy
$59	Invalid file level

GET_LEVEL ($1B)

This function returns the current value of the system file level (see Chapter 2). All subsequent `OPEN` calls will assign this level to the files opened All subsequent `CLOSE` calls for *multiple* files (that is, those calls using a specified `ref_num` of zero) will be effective only on those files that were opened when the system level was greater than or equal to its current level.

Parameter Block:



Offset	Label		Description
$00–$01	`level`	**parameter name:**	system file level
		size and type:	word result (high-order byte zero)
		range of values:	$0000–$00FF
		The current value of the system file level.	

Possible ProDOS 16 Errors

$07	ProDOS is busy

Chapter 11

Device Calls

Device calls access storage devices directly, rather than through the volumes or files on them.

The ProDOS 16 device calls are described in the following order:

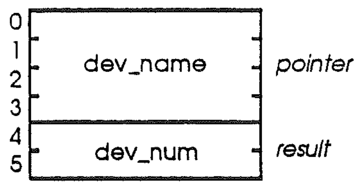
Number	**Function**	**Purpose**
$20	`GET_DEV_NUM`	returns a device's number
$22	`READ_BLOCK`	transfers 512 bytes from a device
$23	`WRITE_BLOCK`	transfers 512 bytes to a device

GET_DEV_NUM ($20)

For the device specified by name, this function returns its device number. All other device calls must refer to the device by its number.

Device numbers are assigned by ProDOS 16 at startup (boot) time. They are consecutive integers, assigned in the order in which ProDOS 16 polls external devices (see Chapter 4).

Parameter Block:



Offset	Label	Description
$00–$03	`dev_name`	**parameter name:** device name **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the device name.
$04–$05	`dev_num`	**parameter name:** device reference number **size and type:** word result (high-order byte zero) **range of values:** $0000–$00FF The device's reference number, to be used in other device calls.

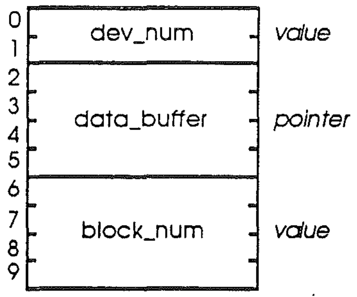
Possible ProDOS 16 Errors

$07	ProDOS is busy
$10	Device not found
$11	Invalid device number
$40	Invalid pathname (device name) syntax

READ_BLOCK ($22)

This function reads one block of information from a disk device (specified by *dev_num*) into memory starting at the address pointed to by *data_buffer*. The buffer must be at least 512 bytes in length, because existing devices define a block as 512 bytes.

Parameter Block:



Offset	**Label**		**Description**
$00–$01	`dev_num`	**parameter name:**	dreference number
		size and type:	word value (high-order byte zero)
		range of values:	$0000–$00FF
			The device's reference number, as returned by `GET_DEV_NUM`.
$02–$05	`data_buffer`	**parameter name:**	data buffer
		size and type:	long word pointer (high-order byte zero)
		range of values:	$0000 0000–$00FF FFFF
			The long word address of a buffer that will hold the data to be read in.
$06–$09	`block_num`	**parameter name:**	block number
		size and type:	long word value (high word zero)
		range of values:	$0000 0000–$0000 FFFF
			The number of the block to be read in.

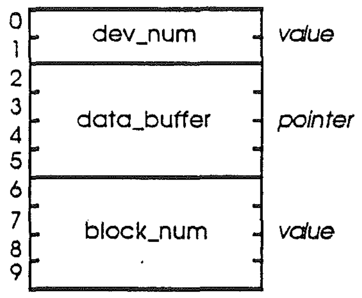
Possible ProDOS 16 Errors

$07	ProDOS is busy
$11	Invalid device number
$27	I/O error
$28	No device connected

WRITE_BLOCK ($23)

This function transfers one block of data from the memory buffer pointed to by *data_buffer* to the disk device specified by *dev_name*. The block is placed in the specified logical block of the volume occupying that device. For currently defined devices, the data buffer must be at least 512 bytes long.

Parameter Block:



Offset	Label	Description
$00–$01	`dev_num`	**parameter name:** reference number **size and type:** word value (high-order byte zero) **range of values:** $0000–$00FF The device's reference number, as returned by `GET_DEV_NUM`.
$02–$05	`data_buffer`	**parameter name:** data buffer **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer that holds the data to be written.
$06–$09	`block_num`	**parameter name:** block number **size and type:** long word value (high word zero) **range of values:** $0000 0000–$0000 FFFF The number of the block to be written to.

Possible ProDOS 16 Errors

$07	ProDOS is busy
$11	Invalid device number
$27	I/O error
$28	No device connected
$2B	Disk write-protected

Chapter 12

Environment Calls

These calls deal with the Cortland operating environment, the software and hardware configuration within which applications run. They include calls to start and end ProDOS 16 applications, and to determine pathnames and versions of system software.

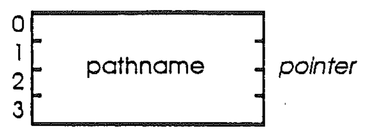
The ProDOS 16 environment calls are described in the following order:

Number	**Function**	**Purpose**
$27	`GET_PATHNAME`	returns application pathname
$28	`GET_BOOT_VOL`	returns ProDOS 16 volume name
$29	`QUIT`	terminates present application
$2A	`GET_VERSION`	returns ProDOS 16 version

GET_PATHNAME ($27)

This function returns the complete pathname of the currently running application.

Parameter Block:



Offset	**Label**	**Description**
$00–$03	`pathname`	**parameter name:** pathname **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the application's pathname.

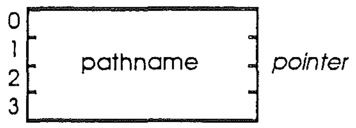
Possible ProDOS 16 Errors

$07	ProDOS is busy
$27	I/O error
$40	Invalid pathname syntax
$44	Path not found
$45	Volume not found
$46	File not found
$4A	Version error
$4B	Unsupported storage type

GET_BOOT_VOL ($28)

This function returns the name of the volume from which ProDOS 16 was last loaded.

Parameter Block:



Offset	Label	Description
$00–$03	`pathname`	**parameter name:** pathname **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the boot volume's name.

Possible ProDOS 16 Errors

$07	ProDOS is busy

QUIT ($29)

Calling this function terminates the present application. It also closes all files, sets the current system file level to zero, and deallocates any installed interrupt handlers. ProDOS 16 can then either 1.) launch a file specified by the quitting program or by the user, or 2.) automatically launch a program specified in the quit return stack.

The **quit return stack** is a table maintained in memory by ProDOS 16. It provides a convenient means for a shell program to pass execution to subsidiary programs (even other shells), while ensuring that control eventually returns to the shell.

For example, a program selector may push its UserID onto the quit return stack whenever it launches an application (by making a `QUIT` call). That program may or may not specify yet another program when it quits, and it may or may not push its own UserID onto the quit return stack. Eventually, however, when no more programs have been specified and no others are waiting for control to return to them, the program selector's UserID will be pulled from the stack and it will be executed once again.

Two `QUIT` call parameters control these options, as follows:

1. Pathname pointer:
    a. If the pathname pointer in the parameter block indicates a nonzero pathname, the indicated program is loaded and executed.
    b. If `pathname` is null (the address it points to contains a zero length byte), ProDOS 16 pulls a UserID from the quit return stack and executes the program with that ID.
    c. If `pathname` is null *and* the quit return stack is empty, ProDOS 16 executes a built-in interactive dispatcher that prompts the user for the name of the next system file to launch.
2. Flag word:

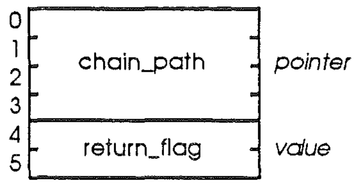
    The flag word contains two boolean values: a return flag and a restart-from-memory flag.

    a. If the return flag value is TRUE (bit 15=1), the UserID of the program making the `QUIT` call is pushed onto the quit return stack. If the return flag is FALSE, no ID is pushed onto the stack.
    b. If the value of the restart-from-memory flag is TRUE (bit 14=1), the program is capable of being restarted from a dormant state in the computer's memory. If the restart-from-memory flag is FALSE, the program must always be reloaded from disk when it is run. Every time a program's UserID is pushed onto the quit return stack, the information from this flag is saved along with it.

Note: Programs that may be restarted from memory are called **reentrant**: they initialize all their variables each time they execute, and make no assumptions about the state of the machine when they gain control.

Caution: The ProDOS 16 `QUIT` call (unlike its ProDOS 8 equivalent) may return an error.

Parameter Block:



Offset	Label	Description	
$00–$03	`pathname`	**parameter name:**	pathname
		size and type:	long word pointer (high-order byte zero)
		range of values:	$0000 0000–$00FF FFFF

The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the pathname of the next file to execute.

Offset	Label	Description	
$04–$05	`flags`	**parameter name:**	flag word
		size and type:	word value
		range of values:	$0000–$C000

Two boolean flags in a 16-bit field. The bits are defined as follows:

bit	significance
15	if = 1, place calling program's UserID on return stack
14	if = 1, calling program may be restarted from memory
13–0	(reserved)

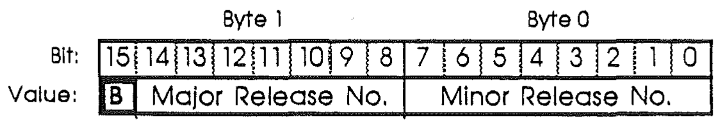
Possible ProDOS 16 Errors

$27	I/O error
$40	Invalid pathname syntax
$44	Path not found
$45	Volume not found
$46	File not found
$4A	Incompatible file format
$4B	Unsupported stroage type
$53	Block number out of range
$54	Out of memory
$5A	Parameter out of range
$5C	Not an executable system file
$5D	Operating system not available

GET_VERSION ($2A)

This function returns the version number of the currently running ProDOS 16 operating system.

The returned version number is placed in the version parameter field. Both byte and bit values are significant. Its format is



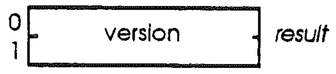
where

- Byte 0 is the minor release number (= 0 for ProDOS 16 version 1.0)
- Byte 1 is the major release number (= 1 for ProDOS 16 version 1.0)
- B = The MSB (most significant bit) of byte 1

and

B = 0 for final releases
B = 1 for all prototype releases

Parameter Block:



Offset	Label	Description
$00–$01	`version`	**parameter name:** version **size and type:** word result (high-order byte zero) **range of values:** $0000–$FFFF The version number of ProDOS 16.

Possible ProDOS 16 Errors

$07	ProDOS is busy

Chapter 13

Interrupt Control Calls

These calls allocate and deallocate interrupt handling routines.

The ProDOS 16 interrupt control calls are described in the following order:

Number	Function	Purpose
$31	`ALLOC_INTERRUPT`	installs an interrupt handler
$32	`DEALLOC_INTERRUPT`	removes an interrupt handler

ALLOC_INTERRUPT ($31)

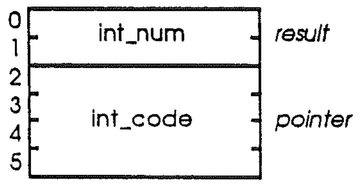
This function places the address of an interrupt handling routine into the interrupt vector table. You should make this call before enabling the hardware that can cause the interrupt. It is your responsibility to make sure that the routine is installed at the proper location and that it follows interrupt conventions (see Chapter 7).

The returned *int_num* is a reference number for the handler. Its only use is to identify the handler when deallocating it; you must refer to a routine by its interrupt handler number to remove it from the system (with `DEALLOC_INTERRUPT`).

When ProDOS 16 receives an interrupt, it polls the installed handlers in sequence, according to their order in the interrupt vector table. The first handler installed has the highest priority. Each new handler installed is added to the end of the table; each one deallocated is removed from the list and the table is compacted.

> **Note:** Under ProDOS 8, the interrupt handler number is equal to the handler's position in the polling sequence. By contrast, the value of `int_num` under ProDOS 16 is unrelated to the order in which handlers are polled.

Parameter Block:



Offset	Label	Description
$00–$01	`int_num`	**parameter name:** interrupt handler number **size and type:** word result (high-order byte zero) **range of values:** $0000–$00FF The identifying number assigned to the interrupt handler by ProDOS 16.
$02–$05	`int_code`	**parameter name:** interrupt code **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of the interrupt handler routine.

Possible ProDOS 16 Errors

$07	ProDOS is busy
$25	Interrupt vector table full
$53	Invalid parameter

DEALLOC_INTERRUPT ($32)

This function clears the entry (specified by *int_num*) for an interrupt handler from the interrupt vector table. You must disable the associated interrupt hardware before making this call; a fatal error will result if a hardware interrupt occurs after its entry has been cleared from the vector table.

`DEALLOC_INTERRUPT` has no effect on the order of the polling sequence for the remaining handlers. Any subsequently allocated handlers will be added to the end of the polling sequence.

Parameter Block:

0
1
int_num
value

Offset	**Label**		**Description**
$00–$01	`int_num`	**parameter name:**	interrupt handler number
		size and type:	word value (high-order byte zero)
		range of values:	$0000–$00FF
			The identifying number assigned to the interrupt handler by ProDOS 16.

Possible ProDOS 16 Errors

$07	ProDOS is busy
$53	Invalid parameter

Chapter 14

ProDOS 16 Error Codes

The following is a complete list of operating system errors returned by ProDOS 16. When it returns an error, ProDOS 16 places the error number in the accumulator (A-register) and sets the Status register carry bit.

Each error code is followed by the error name and a brief description of its significance.

Nonfatal errors

A nonfatal error signifies that a requested call could not be completed properly, but program execution may continue.

Number	Message and Description
	General Errors:
$00	(no error)
$01	**Invalid call number:** a nonexistent command has been issued.
$07	**ProDOS is busy:** the call cannot be made because ProDOS 16 is busy with another call.
	Device call errors:
$10	**Device not found:** there is no device on line with the given name (`GET_DEV_NUM` call)
$11	**Invalid device name or number:** the given device name or reference number is not in ProDOS 16's list of active devices (`VOLUME`, `READ_BLOCK` and `WRITE_BLOCK` calls)
$25	**Interrupt vector table full:** the maximum number of user-defined interrupt handlers (16) has already been installed; there is no room for another (`ALLOC_INTERRUPT` call).

$27 **I/O error:** a hardware failure has prevented proper data transfer to or from a disk device. This is a general code covering many possible error conditions.

$28 **No device connected:** There is no device in the slot and drive specified by the given device number (`READ_BLOCK` and `WRITE_BLOCK` calls).

$2B **Write-protected:** The specified volume is write-protected (the "write-protect" tab or notch on the disk jacket has been enabled). No operation that requires writing to the disk can be performed.

$2E **Disk switched:** The requested operation cannot be performed because a disk containing an open file has been removed from its drive.

Warning: Apple II drives have no hardware method for detecting disk switches. This error is therefore returned only when ProDOS 16 checks a volume name during the normal course of a call. Since most disk access calls do not involve a check of the volume name, a disk-switched error can easily go undetected.

$2F **Device not online:** A device specified in a call is not connected tothe system. This error may be returned by device drivers that can sense whether or not a specific device is on line.

$30 - $3F **Device-specific errors:** (error codes in this range are to be defined and used by individual device drivers.)

File call errors:

$40 **Invalid pathname syntax:** The specified pathname or device name contains illegal characters

$42 **FCB table full:** The table of file control blocks is full; the maximum permitted number of open files (8) has already been reached. You may not open another file (`OPEN` call).

$43 **Invalid file reference number:** the specified file reference number does not match that of any currently open file.

$44 **Path not found:** A subdirectory name in the specified pathname does not exist (the pathname's syntax is otherwise valid).

$45 **Volume not found:** The volume name in the specified pathname does not exist (the pathname's syntax is otherwise valid).

$46 **File not found:** The last file name in the specified pathname does not exist (the pathname's syntax is otherwise valid).

$47 **Duplicate pathname:** An attempt has been made to create or rename a file, using an already existing pathname (`CREATE`, `CHANGE_PATH` calls).

$48	**Volume full:** an attempt to allocate blocks on a disk device has failed, due to lack of space on the volume in the device (`CREATE`, `WRITE` calls). If this error occurs during a write, ProDOS 16 writes data is until the disk is full, and still permits you to close the file.
$49	**Volume directory full:** No more space for entries is left on the volume directory (`CREATE` call). In ProDOS 16, a volume directory can hold no more than 51 entries. No more files can be added to this directory until others are destroyed (deleted).
$4A	**Version error (incompatible file format):** The version number in the specified file's directory entry does not match the present ProDOS 8-ProDOS 16 file format version number. This error can only occur in future versions of ProDOS 16, since for all present versions of ProDOS 8 and ProDOS 16 the file format version number is zero.

Note: The version number referred to by this error code concerns the *file format only*, not the version number of the operating system as a whole. In particular, it is unrelated to the ProDOS 16 version number returned by the `GET_VERSION` call.

$4B	**Unsupported (or incorrect) storage type:** The organization of the specified file is unknown to ProDOS 16. See Appendix A for a list of valid storage types. This error may also be returned if a directory has been tampered with, or if a prefix has been set to a nondirectory file.
$4C	**End-of-file encountered (out of data):** A read has been attempted, but the current file position (MARK) is equal to end-of-file (EOF), and no further data can be read.
$4D	**Position out of range:** The specified file position parameter (MARK) is greater than the size of the file (EOF).
$4E	**Access not allowed:** One of the attributes in the specified file's access byte forbids the attempted operation (renaming, destroying, reading, or writing).
$50	**File is open:** An attempt has been made to perform a disallowed operation on an open file (`OPEN`, `CHANGE_PATH`, `DESTROY` calls).
$51	**Directory structure damaged:** The number of entries indicated in the directory header does not match the number of entries the directory actually contains.
$52	**Unsupported volume type:** The specified volume is not a ProDOS 16, ProDOS 8, or SOS disk. Its directory format is incompatible with ProDOS 16.
$53	**Parameter out of range:** The value of one or more parameters in the parameter block is out of its range of permissible values.

$54	**Out of Memory:** A ProDOS 8 program specified by the QUIT call is too large to fit into the memory space available for ProDOS 8 applications.
$55	**VCB table full:** The table of volume control blocks is full; the maximum permitted number of online volumes/devices (8) has already been reached. You may not add another device to the system. The error occurs when 8 devices are on line and a `VOLUME` call is made for another device that has no open files.
$57	**Duplicate volume:** Two or more online volumes have identical volume directory names. This message is a warning; it does not prevent access to either volume. However, ProDOS 16 has no way of knowing which volume is intended if the volume name is specified in a call.
$58	**Not a block device:** An attempt has been made to access a device that is not a block device. ProDOS 16 (v. 1.0) supports access to block devices only.
$59	**Invalid level:** The value specified for the system file level is out of range (`SET_LEVEL` call).
$5A	**Block number out of range:** The volume bit map indicates that the volume contains blocks beyond the block count for the volume. This error can occur only if the disk directory is damaged.
$5B	**Illegal pathname change:** the pathnames on a `CHANGE_PATH` call specify two different volumes. `CHANGE_PATH` can move files among directories only on the *same* volume.
$5C	**Not an executable system file:** The file specified in a `QUIT` call is not type $B3 or $FF. All applications launched by the `QUIT` call must be type $B3 (Cortland application) or $FF (ProDOS 8 system file).
$5D	**Operating system not available:** A ProDOS 8 application has been specified by the QUIT call, but the ProDOS 8 operating system is not on the system disk.

Fatal errors

A fatal error signifies the occurrence of a malfunction so serious that processing must halt. To resume execution following a fatal error, you must reboot the system.

Number	Message and Description
$01	**Unclaimed interrupt:** An interrupt signal has occurred and none of the installed handlers claims responsibility for it. This error may occur if interrupt-producing hardware is installed before its associated interrupt handler is allocated.

$0A	**VCB unusable:** The volume control block table has been damaged. The values of certain check bytes are not what they should be, so ProDOS 16 cannot use the VCB table.
$0B	**FCB unusable:** The file control block table has been damaged. The values of certain check bytes are not what they should be, so ProDOS 16 cannot use the FCB table.
$0C	**Block zero allocated illegally:** Write-access to block zero on a disk volume has been attempted. Block zero on all volumes is reserved for boot code.
$0D	**Interrupt occurred while I/O shadowing off:** The Cortland has soft switches that control **shadowing** from banks $E0 and $E1 to banks $00 and $01. If an interrupt occurrs while those switches are off, the firmware interrupt-handling code will not be enabled. See *Cortland Firmware Reference.*

Bootstrap errors

Bootstrap errors can occur when the Cortland attempts to start up a ProDOS 16 system disk. Errors can occur at several points in this process:

1. If there is no disk in any drive, a "sliding apple" symbol appears on the screen along with the message:

   ```
   check startup device
   ```

   Place a system disk in a drive and press Control--Reset to restart the boot procedure.

2. If there is a disk in a drive, but it is not a ProDOS 8 or ProDOS 16 system disk (that is, there is no type $FF file named `PRODOS` on it), the following message appears:

   ```
   cannot find ProDOS
   ```

   Remove the disk and replace it with another containing the proper files, then press Control--Reset to restart the boot procedure.

3. If the file named `PRODOS` is found, but another essential file is missing, a message such as

   ```
   No SYSTEM/P16 file found
   ```

   or

   ```
   No x.SYSTEM or x.SYS16 file found
   ```

   may appear. Remove the disk and replace it with another containing the proper files, then press Control--Reset to restart the boot procedure.

Another type of ProDOS 16 bootstrap error occurs on other Apple II computers. If you try to boot a ProDOS 16 system disk on any Apple II computer that is *not* a Cortland, the following error message is displayed:

```
PRODOS/16 REQUIRES APPLE //-16 HARDWARE
```

When this occurs the disk will not boot. You can only boot a Cortland System Disk on a Cortland computer.

Part III

The System Loader

The System Loader is a Cortland Tool Set that works closely with ProDOS 16. It is responsible for loading all code and data into the Cortland's memory. It is capable of static and dynamic loading and relocating of code or data segments, subroutines and libraries.

Chapter 15 explains in general terms how the System Loader works. Chapter 16 details some of its functions and data structures. Chapter 17 gives programming suggestions for using the System Loader. Chapter 18 shows how to make loader calls and describes each call in detail. Chapter 19 is a complete list of System Loader error codes.

Chapter 15

Introduction to the System Loader

This chapter gives a basic picture of the System Loader, defines some of the important terms needed to describe what the loader does, describes its interactions with the Memory Manager, and presents an outline of the procedures it follows when loading a program into memory. Additional related terms are defined in the Glossary.

What is the System Loader?

The System Loader is a set of software routines that manages the loading of program segments into the Cortland. It is a Cortland Tool Set; as such, it is independent of ProDOS 16. However, it works very closely with ProDOS 16 and with the Memory Manager, another Tool Set. The System Loader has several improvements over the loading method under ProDOS 8 on other Apple II computers:

- It makes loading easier and more convenient. Under ProDOS 8, the only automatic loading is performed by the boot code, which searches the boot disk for the first `.SYSTEM` file (type $FF) and loads it into location $2000. If a system program needs to call another application it must do all the work itself, either by making ProDOS 8 calls or by providing its own loader. On the Cortland, calls to the System Loader perform the task more simply.
- It is a **relocating loader:** it loads relocatable programs at any available location in memory. Under ProDOS 8, a program must be loaded at a fixed memory address, or at an address specified by the system program that does the loading. The relocating loader relieves the programmer of the burden (and restriction) of deciding where to load programs.
- It is a **segment loader:** it can load different segments of a program independently, to use memory efficiently.
- It is a **dynamic loader:** it can load certain program segments as they are needed during execution, rather than at boot time only.

The System Loader handles files generated by the **CPW Linker**; the linker handles files produced by a Cortland **assembler** or **compiler.** The linker, assembler, and compilers are part of the Cortland Programmer's Workshop (CPW), a powerful and flexible set of development programs designed to help programmers produce Cortland applications efficiently and conveniently. See Chapter 6 of this manual for more information and references on Cortland Programmer's Workshop.

Loader terminology

The System Loader is a program that processes **load files.** Load files are ProDOS 16 applications (file-type $B3), run-time library files (file-type $B4), or other types of system files (file-types $B6-$BF; see Appendix D). They must follow Object Module Format (OMF) specifications, as defined in the *Cortland Programmer's Workshop Reference.* Each load file consists of **load segments** that can be loaded into memory independently; load segments in a given load file are numbered sequentially, starting with one.

Load files can contain segments of various kinds. Some segments consist of **program code** or **data**; others provide location information to the loader. The **Jump Table segment**, when loaded into memory, provides a mechanism by which segments in memory can trigger the loading of other needed segments. Each load file can have only one Jump Table segment. A load file may also have one segment called the **Pathname segment**, which provides a cross-reference between file numbers (in the Jump Table segment) and pathnames (on disk) of the segments to be loaded. A third special type of segment is the **initialization segment.** It contains any code that has to be executed first, before the rest of the segments are loaded. All three types of segments are discussed more fully in the next chapter.

Code and data segments can be either **static** or **dynamic.** A program's static load segments are loaded into memory at initial load time (when the program is first started up); they must stay in memory until the program is complete. Dynamic load segments, on the other hand, are not placed in memory at initial load time; they are loaded as needed during program execution. Dynamic loading can be automatic (through the Jump Table) or manual, at the specific request of the application (through System Loader function calls). When a dynamic segment is no longer needed by the program that called it, it can be **purged**, or deleted, by the Memory Manager.

Segments may be either **absolute** or **relocatable.** An absolute segment must be loaded into a specific location in memory, or it will not function properly. A relocatable segment can execute correctly wherever the System Loader places it.

A **controlling program** is a program that requests the System Loader to perform an **initial load** on another major program (usually an **application**). The **UserID Manager** (a Cortland Toolset) assigns a unique ID number (UserID) to that application, so the loader may quickly locate all of the application's segments if necessary. A switcher is an example of a controlling program; a word processor is an example of an application.

When a program (load file) is initially loaded, only the static load segments are placed in memory; at that point the System Loader has all the information it needs to resolve all symbolic references among them. Until a dynamic segment is loaded, however, the loader cannot resolve references to it because it does not know where in memory it will be. Thus static segments may be directly referenced (by each other and by dynamic segments), but dynamic segments can be referenced only through `JSL` (long subroutine jump) calls to the Jump Table.

When the System Loader is called to load a program, it loads all static load segments including the Jump Table segment (if any), and the Pathname segment (if any). The **Jump Table** and the **Pathname Table** are constructed from these two segments, respectively. During this process, a **Memory Segment Table** is also constructed in memory. These three tables are discussed in more detail in the next chapter.

Interface with the Memory Manager

The System Loader and the Memory Manager work closely together. The Memory Manager is a Cortland Toolset (firmware program) that is responsible for allocating memory in the Cortland; it provides space for load segments, tells the System Loader where to place them, and moves segments around within memory when additional space is needed.

When the System Loader loads a program segment, it calls the Memory Manager to allocate a corresponding **memory block**. If the program segment is static, and therefore must not be unloaded or moved, its memory block is marked as **unpurgeable** and **unmovable**. That means that the Memory Manager cannot change that segment's position or contents as long as the program is running. If the program segment is dynamic, its memory block is initially marked as **purgeable** but **locked** (temporarily unpurgeable and unmovable; subject to change during execution of the program). If the dynamic segment is **position-independent**, its memory block is marked as movable; otherwise, it is unmovable.

To *unload* a segment, the System Loader calls the Memory Manager to make the coresponding memory block purgeable. If the controlling program wishes to unload *all* segments associated with a particular application (for example, at shutdown), it calls the System Loader's Application Shutdown function, which in turn calls the Memory Manager to (1) purge the memory blocks of all dynamic segments for the application, and then (2) make the memory blocks of all static segments purgeable. This process frees space but keeps the application's static segments in memory, thus speeding up execution of a finder or switcher that may shortly need to reload that application. Of course, if memory runs out and the Memory Manager *is* forced to purge one of those static segments, the application can no longer be used. The next time it is needed, the application must be loaded from scratch.

Load segments have attributes that are closely related to their corrresponding memory blocks. Load segments may be: dynamic or static, position-independent (or not), and absolute or relocatable. Memory blocks may be: purgeable or unpurgeable, fixed (= unmovable) or not fixed (= movable), and locked or unlocked. A typical load segment will be placed in a memory block that is

Locked
Fixed
Purge Level = 0 (if the segment is static)
Purge Level = 1 (if the segment is dynamic)

Depending on other requirements the segment may have, such as alignment in memory, the load segment-memory block relationship may be more complex. Table 15-1 shows all possible relationships between the two that may hold at load time. The direct-page/stack segment has special characteristics described in Chapter 6.

Table 15-1. Load segment-memory block relationships (at load time)

Load Segment Attribute	**Memory Block Attribute**
static	unpurgeable, fixed (unmovable)
dynamic	purgeable, locked
absolute (`ORG` > 0)	fixed (unmovable)
relocatable	(no specific relation)
position-independent	not fixed (movable)
not postion-independent	fixed (unmovable)
`KIND` = 11	fixed-bank
`BANKSIZE` = 0	may cross bank boundary
`BANKSIZE` = $10 000	may not cross bank boundary
`ALIGN` = 0	not bank- or page-aligned
`ALIGN` = $100	page-aligned
`ALIGN` = $10 000	bank-aligned
direct-page/stack (`KIND` = 92)	dynamic, fixed bank ($00), page-aligned

Note: `BANKSIZE` and `ALIGN` are segment header fields, described in Appendix D of this manual and under "Object Module Format" in *Cortland Programmer's Workshop Reference.*

A memory block can be locked or unlocked through a call to the System Loader, but other attributes can be changed only through Memory Manager calls. Memory block attributes useful to an application may include

• Start location	
• Size of block	
• UserID	(identifies the application the block is part of)
• Purge level	(0 to 3: 0 = unpurgeable, 3 = most purgeable)

These attributes are not in the Memory Segment Table itself but may be accessed through the memory handle, which is part of the Memory Segment Table. If the memory handle is NIL (0), the memory block has been purged. See Chapter 16 for further explanation.

Note: Strictly speaking, load segments are never *purged* or *locked*; those are actions taken on the memory blocks inhabited by the segments. For simplicity, however, this manual may in certain cases apply terms such as purged or locked to segments.

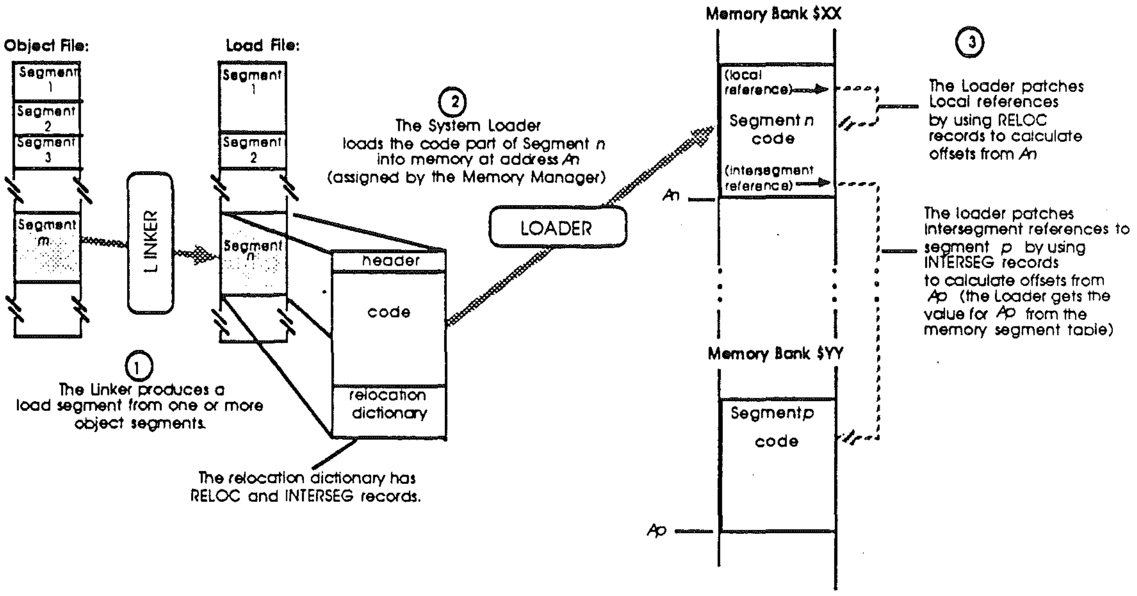
Loading a relocatable segment

This brief description of parts of the operation of the System Loader shows how the linker, loader, and Memory Manager work together to produce and load a relocatable program segment. Figure 15-1 shows the process in a simplified form.

Load-file structure

Load files conform to a subset of **object module format** (OMF). In OMF, each module (file) consists of one or more segments; each segment is further made up of one or more **records.** In a load file specifically, each segment (apart from specialized segments such as the load file tables described in Chapter 16) consists of program code or data, followed (if the segment is relocatable) by a **relocation dictionary.** The relocation dictionary is created by the linker as it converts an object segment into a load segment. The program code or data consists of a single record of a particular type, and the relocation dictionary consists of only two types of records: **RELOC** records, which give the loader the information it needs to resolve local (within-segment) references, and **INTERSEG** records, which give the loader the information it needs to resolve external (intersegment) references. The detailed formats of both types of records are presented in *Cortland Programmer's Workshop Reference.*

When a relocatable segment is loaded into memory, it is loaded at a location determined by the Memory Manager. Furthermore, only the first part of the segment (the program code itself) is loaded into the part of memory reserved by the Memory Manager; the relocation dictionary, if present, is loaded into a buffer or work area used by the loader. After loading the segment, the loader *relocates* it, using the information in the relocation dictionary.



Object File:
Segment 1
Segment 2
Segment 3
Segment m
LINKER
Load File:
Segment 1
Segment 2
Segment n
header
code
relocation dictionary
1
The Linker produces a load segment from one or more object segments.
The relocation dictionary has RELOC and INTERSEG records.
2
The System Loader loads the code part of Segment n into memory at address An (assigned by the Memory Manager)
LOADER
Memory Bank $XX
(local reference)
Segment n code
(intersegment reference)
An
Memory Bank $YY
Segment p code
Ap
3
The Loader patches Local references by using RELOC records to calculate offsets from An
The loader patches Intersegment references to segment p by using INTERSEG records to calculate offsets from Ap (the Loader gets the value for Ap from the memory segment table)

Figure 15-1. Loading a relocatable segment

Relocation

After the System Loader has placed a load segment in memory, it must (unless the segment consists of absolute code) relocate its address references. **Relocation** describes the processing of a load segment so that it will execute properly at the memory location at which it has been loaded. It consists of **patching** (substituting the proper values for) address operands that refer to locations both within and external to the segment. The relocation dictionary part of the segment contains all the information needed by the loader to do this patching. Relocation is performed as follows:

1. Local references in the load segment (coded in the original object file as offsets from the beginning of the segment) are patched from `RELOC` records in the relocation dictionary. Using the starting address of the segment (available from the Memory Manager through the Memory Segment Table), the loader adds that address to each offset, so that the correct memory address is referenced.
2. External references (references to other segments) are coded in the original object module as global variables (subroutine names or entry points). The linker and loader handle them as follows:
    a. If the reference is to a static segment, the *linker* will have calculated the proper file number, segment number, and offset of the referenced (external) segment, and placed that information in an `INTERSEG` record in the relocation dictionary. When the load segment is loaded, the *loader* uses the `INTERSEG` record and the memory location of the external segment (available from the Memory Manager through the Memory Segment Table), and then patches the external reference with the proper memory address of the external segment.

    b. If the reference is to a dynamic segment, the *linker* will have created a slightly different `INTERSEG` record: instead of referencing the file number, segment, and offset of the referenced external segment itself, the `INTERSEG` record references the file number, segment number, and offset of an entry in the *Jump Table*. Therefore, when the load segment is loaded, the *loader* patches the reference to point to the Jump Table entry. That entry, in turn, is what transfers control to the external segment at its proper memory address (if and when the referenced segment is loaded).

    The Jump Table and the reasons for this indirect referencing are described further in Chapter 16. The main point of interest here is that, when it performs relocation, the loader doesn't care whether an intersegment reference is to a static or to a dynamic segment—it treats both in exactly the same way.

The System Loader performs several other functions when it loads dynamic segments, including searching for the name of the segment in the Pathname Table before loading, and patching the appropriate Jump Table entry afterward. These and other functions are described in more detail in the next two chapters.

Chapter 16

System Loader Data Tables

This chapter describes the data tables set up in memory during a load, to provide cross-reference information to the loader. The **Memory Segment Table** allows the loader to keep track of which segments have been loaded, the addresses at which they have been loaded, and whether or not a segment is presently being called from other segments. The **Jump Table** allows programs to reference routines in dynamic segments that may not currently be in memory. The **Pathname Table** provides a cross-reference between file numbers and file pathnames of dynamic segments.

Memory Segment Table

The Memory Segment Table is a linked list, each entry of which describes a memory block known to the System Loader. Memory blocks are allocated by the Memory Manager during loading of segments from a load file, and each block corresponds to a single load segment. Figure 16-1 shows the format of each entry in the Memory Segment Table.

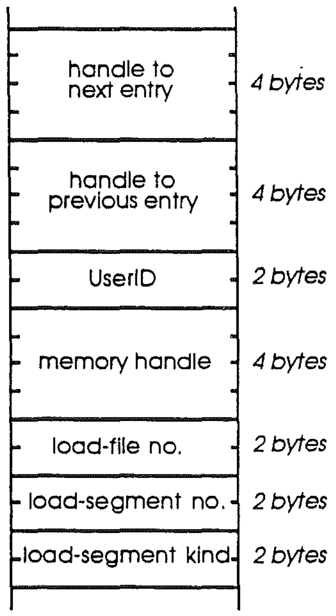


Figure 16-1. Memory Segment Table entry

The fields have the following meanings:

Handle to next entry: The memory handle of the next entry in the Memory Segment Table. This number is 0 for the last entry.

Handle to previous entry: The memory handle of the previous entry in the Memory Segment Table. This number is 0 for the first entry.

UserID: The identification number assigned to the memory block this segment inhabits. Normally, the UserID is available directly from the Memory Manager through the memory handle. However, if the block has been purged its handle is NIL and the UserID must be read from this field.

Memory handle: The identifying number of the memory block, obtained from the Memory Manager. Additional memory block information is available through this handle. This handle is NIL if the block has been purged.

Load-file number: The number of the load file from which the segment was obtained. If the segment is in the initial load file, the number is 1.

Load-segment number: The segment number of the segment in the load file.

Load-segment kind: The value of the `KIND` field in the load segment's header. Segment kinds are described in Appendix D.

Jump Table

The Jump Table allows a program to reference dynamic segments. It consists of the Jump Table Directory and one or more Jump Table segments.

On disk, Jump Table segments are load segments (of kind $02), created by the linker to resolve references to dynamic segments. Any load file or run-time library file may contain a Jump Table segment.

In memory, the Jump Table Directory is created by the loader as it loads Jump Table segments. The Jump Table Directory is a linked list, each entry of which points to a single Jump Table segment encountered by the loader. Figure 16-2 shows the format of an entry in the Jump Table Directory.

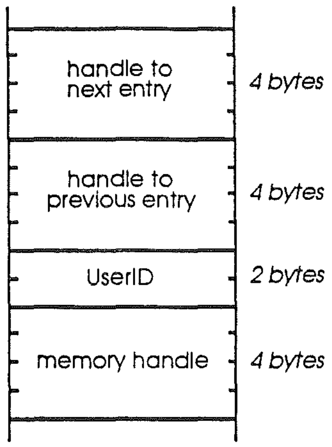


Figure 16-2. Jump Table Directory entry

The fields have the following meanings:

Handle to next entry: The memory handle of the next entry in the Jump Table Directory. This number is 0 for the last entry.

Handle to previous entry: The memory handle of the previous entry in the Jump Table Directory. This number is 0 for the first entry.

UserID: The identification number assigned to the Jump Table segment that this Directory entry refers to.

Memory handle: The handle of the memory block containing the Jump Table segment that this Directory entry refers to.

Like the Directory, the individual Jump Table segments consist of a series of entries. The next three subsections describe the creation, loading, and use of a single Jump Table *segment* entry. The entry is used to resolve a single JSL instruction in a program segment.

Note: Throughout this manual, the term *Jump Table entry* refers to a Jump Table *segment* entry, not a Jump Table *directory* entry.

Creation of a Jump Table entry

The Jump Table load segment is created by the linker, as it processes an object file. Each time the linker encounters a `JSL` to a routine in an external dynamic segment, it creates an `INTERSEG` record in the relocation dictionary of the load segment, and (if it has not done so already) an entry for that routine in the Jump Table segment. The `INTERSEG` record links the `JSL` to the Jump Table entry that was just created. Figure 16-3 shows the format of the Jump Table entry that the linker creates. See also Figure 16-5 (a).

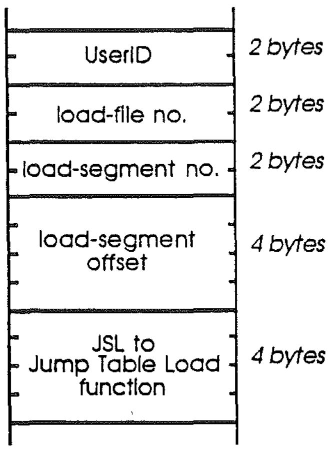


Figure 16-3. Jump Table entry (unloaded state)

The fields have the following meanings:

UserID: The USerID of the referenced dynamic segment.

Load-file number: The load-file number of the referenced dynamic segment.

Load-segment number: The load-segment number of the referenced dynamic segment.

Load-segment offset: The location of the referenced address within the referenced dynamic segment.

`JSL` to Jump Table Load function: A long subroutine jump to the Jump Table Load function. The Jump Table Load function is described in Chapter 18.

The final entry in a Jump Table segment has a load-file number of zero, to indicate that there are no more entries in the segment.

Modification at load time

At load time, the loader places the program segment and the Jump Table segment into memory (it does *not* yet load the referenced dynamic segment). To link the Jump Table segment with any other Jump Table segments it may have loaded, it creates the Jump Table Directory. The Jump Table is now complete.

Using the information in the `INTERSEG` record, the loader patches the `JSL` instruction in the program segment so that it references the proper part of the Jump Table in memory. It also patches the actual address of the Jump Table Load function into the Jump Table entry. The Jump Table segment is now in its **unloaded state.** See Figure 16-5 (b).

Use during execution

During program execution, when the `JSL` instruction in the original load segment is encountered, the following sequence of events takes place:

1. Control transfers to the proper Jump Table entry.
2. The `JSL` in the entry transfers control to the System Loader's Jump Table Load function.
3. The Jump Table Load function gets (a) the load-file number, load-segment number, and load-segment offset of the dynamic segment from the Jump Table entry; and (b) the file pathname of the dynamic segment from the Pathname Table.
4. The System Loader loads the dynamic segment into memory.
5. The loader changes the dynamic segment's entry in the Jump Table to its **loaded state.** The loaded state is identical to the unloaded state, except that the `JSL` to the Jump Table Load function is replaced by a `JMP` to the external reference itself. Figure 16-4 shows the format for the loaded state.

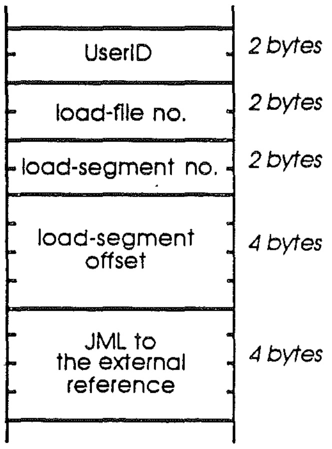


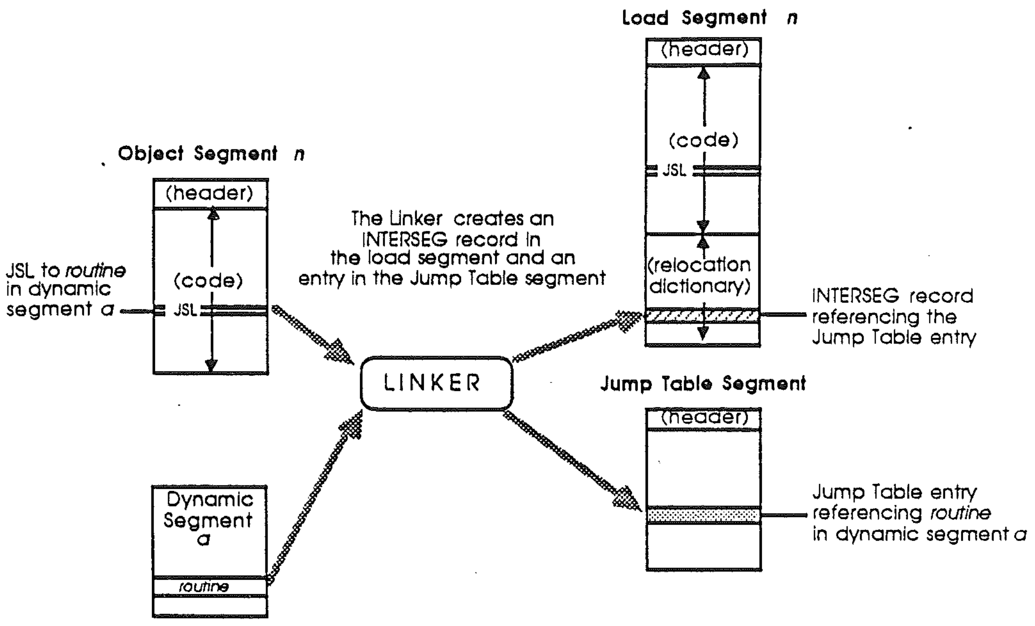
Figure 16-4. Jump Table entry (loaded state)

6. The loader transfers control to the dynamic segment. When the new segment has finished its task (typically it is a subroutine and exits with an `RTL`), control returns to the statement following the original `JSL` instruction. See Figure 16-5 (c).

Jump Table diagram

Figure 16-5 is a simplified diagram of how the Jump Table works. It follows the creation, loading, and use of a single Jump Table entry, needed to resolve a single instruction in load segment *n*. The instruction is a `JSL` to a subroutine named *routine* in dynamic segment *a*.

a. Creation by the linker:


Load Segment *n*
(header)
(code)
JSL
(relocation dictionary)
INTERSEG record referencing the Jump Table entry
Object Segment *n*
(header)
(code)
JSL
JSL to *routine* in dynamic segment *a*
The Linker creates an INTERSEG record in the load segment and an entry in the Jump Table segment
LINKER
Jump Table Segment
(header)
Jump Table entry referencing *routine* in dynamic segment *a*
Dynamic Segment *a*
routine

b. Modification at Load Time:

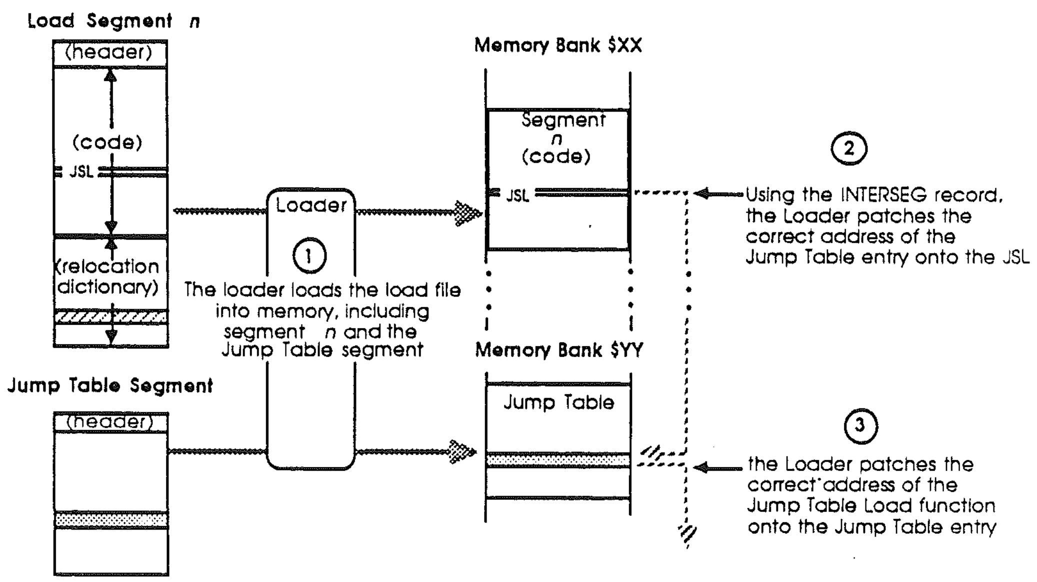

Load Segment *n*
(header)
(code)
JSL
(relocation dictionary)
Jump Table Segment
(header)
Loader
1
The loader loads the load file into memory, including segment *n* and the Jump Table segment
Memory Bank $XX
Segment *n* (code)
JSL
2
Using the INTERSEG record, the Loader patches the correct address of the Jump Table entry onto the JSL
Memory Bank $YY
Jump Table
3
the Loader patches the correct address of the Jump Table Load function onto the Jump Table entry

Figure 16-5. How the Jump Table works

c. Use During Execution:

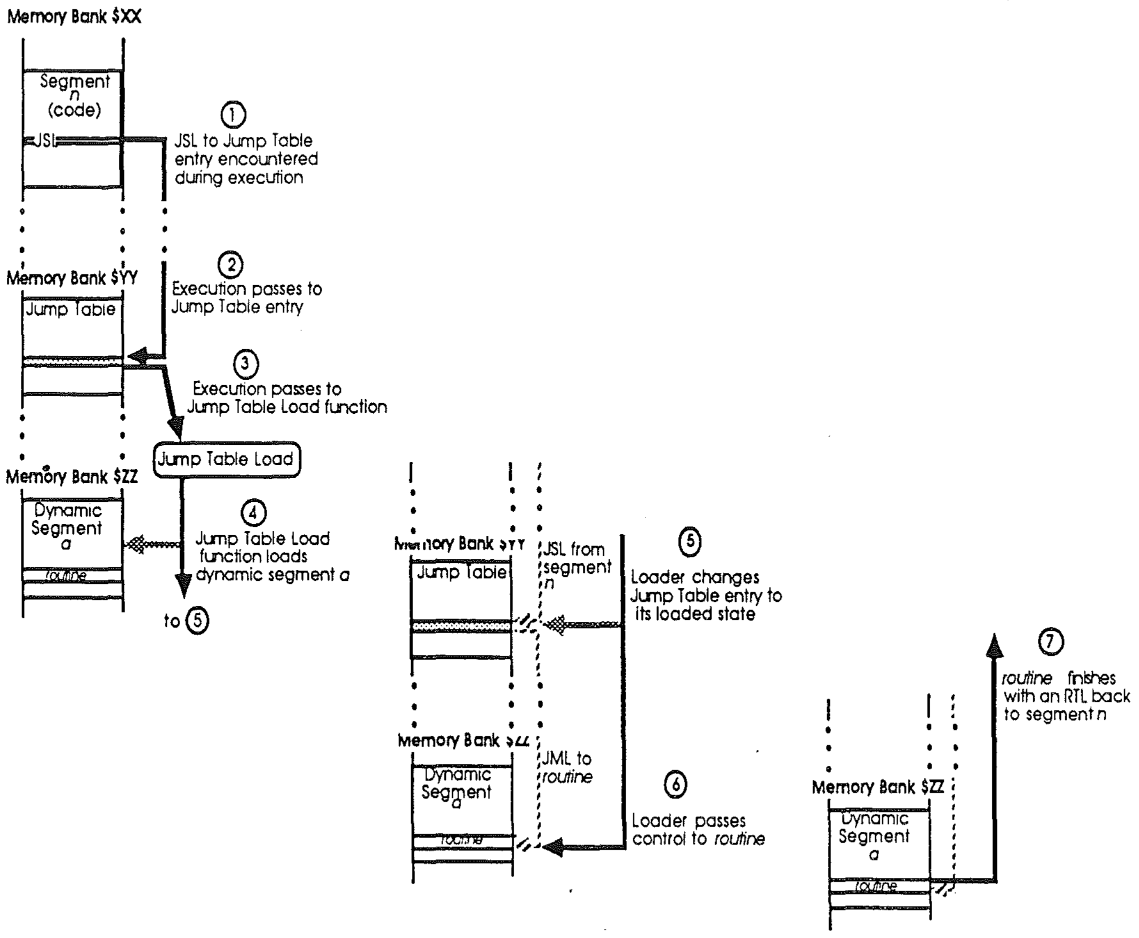

Memory Bank $XX
Segment n (code)
JSL
1
JSL to Jump Table entry encountered during execution
Memory Bank $YY
Jump Table
2
Execution passes to Jump Table entry
3
Execution passes to Jump Table Load function
Jump Table Load
Memory Bank $ZZ
Dynamic Segment a
routine
4
Jump Table Load function loads dynamic segment a
to 5
Memory Bank $YY
Jump Table
JSL from segment n
5
Loader changes Jump Table entry to its loaded state
Memory Bank $ZZ
Dynamic Segment a
routine
JML to routine
6
Loader passes control to routine
7
routine finishes with an RTL back to segment n
Memory Bank $ZZ
Dynamic Segment a
routine

Figure 16-5. How the Jump Table works (continued)

Pathname Table

The Pathname Table provides a cross-reference between file numbers and file pathnames, to help the System Loader find the load segments that must be loaded dynamically. The Pathname Table is a linked list of individual pathname entries; it starts with an entry for the pathname of the initial load file, and includes any entries from segments of kind $04 (Pathname segments) that the loader encounters during the load. Also, if run-time library files are referenced during program execution, their own pathname segments are linked to the original one.

A load file's Pathname *segment* (KIND = $04) contains one entry for each run-time library file referenced by the file. Each entry consists of a load-file number, file date and time, and a pathname. A load file number of 0 indicates that there are no more entries in the segment.

The Pathname *Table* is constructed in memory; its entries are identical to Pathname segment entries, except that each also contains two link handles and a UserID field. Figure 16-6 shows the format of a Pathname Table entry.

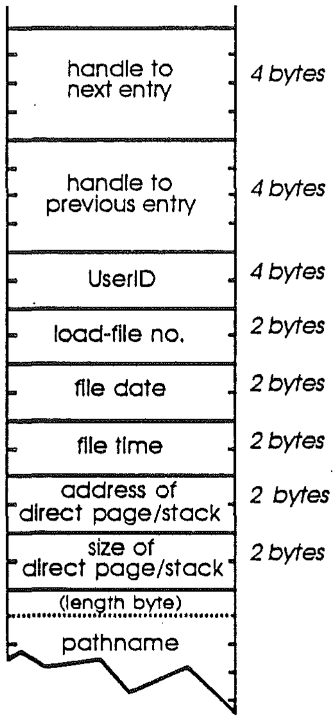


Figure 16-6. Pathname Table entry

The fields have the following meanings:

Handle to next entry: the memory handle of the next entry in the Pathname Table. For the last entry, the value of the handle is 0.

Handle to previous entry: the memory handle of the previous entry in the Pathname Table. For the first entry, the value of the handle is 0.

UserID: the ID associated with this entry. Generally, each load file has a unique UserID, and a single entry in the Pathname Table. Each new run-time library encountered during execution is assigned the application's UserID.

File number: the number assigned to a specific load file by the linker. File number 1 is reserved for the initial load file.

File date: the date on which the file was last modified.

File time: the time at which the file was last modified.

The *file·date* and *file time* are ProDOS 16 directory items retrieved by the linker during linking. They are included in the Pathname Table as an identity check on run-time library files (they are ignored for other file types). To ensure that the run-time library file used at program execution is the same one originally linked by the linker, the System Loader compares these values to the directory entries of the run-time library file to be loaded. If they do not match, the System Loader will not load the file.

Direct-page/stack address: the starting address of the buffer allocated (at initial load) for the file's direct page (zero page) and stack.

Direct-page/stack size: the size (in bytes) of the buffer allocated for the file's direct page and stack.

The direct-page/stack address and size fields are in the Pathname Table to allow the Restart function to more quickly resurrect a dormant application (see "Restart" and "User Shutdown" Chapter 18). These two fields are ignored for run-time library files.

File pathname: the full or partial pathname of this entry. Partial pathnames are prepended with one of 8 prefix designators (see Chapter 5), three of which are reserved as follows:

`0/` = ProDOS 16 system prefix (initially the boot volume name)

`1/` = the current application's subdirectory

`2/` = system library subdirectory (initially `/V/SYSTEM/LIBS`, where `/V/` is the boot volume name)

The pathname is a *Pascal string*, meaning that it consists of a length byte (of value *n*) followed by an ASCII string (*n* bytes long).

Chapter 17

Programming With the System Loader

This chapter discusses how you can use the capabilities of the System Loader at several different levels, depending on the complexity of the programs you wish to write. It also gives reqirements for designing controlling programs (shells)—programs that control the loading and execution of other programs.

Programming suggestions for ProDOS 16 are in Chapter 6 of this manual. More general information on how to program for the Cortland is available in *Programmer's Introduction to the Cortland.* For language-specific programming instructions, consult the appropriate language manual in the Cortland Programmer's Workshop (see "Cortland Programmer's Wokshop" in Chapter 6).

Static programs

The functioning of the System Loader is completely transparent to simple applications. Any program that is loaded into memory in its entirety at the beginning of execution, and which does not call any other programs or routines that must be loaded during run time, need not know anything about the System Loader. If such a static program is in proper object module format, it will be automatically loaded, relocated, and executed whenever it is called.

Programming With dynamic segments

You may write Cortland programs that use memory more efficiently than the simple application described above. If your program is divided into static and dynamic segments, only the static segments are loaded when the program is started up. Dynamic segments are loaded only as needed during execution, and the memory they occupy is available again when they are no longer needed.

Dynamic loading also is transparent to the typical application; no System Loader commands are necessary to invoke it. If you segment your program as you write the source code, and if you define the proper segments as dynamic and static when the object code is linked, the loading and execution of dynamic segments will be completely automatic.

Because segments are specified as static or dynamic at link time, you may experiment with several configurations of a single program after it has been assembled. For example, you might first run the program as a single static segment, then run several different static-dynamic combinations to see which gives the best performance for the amount of memory required. In this way the same program could be tailored to different machines with different memory configurations.

In general, the least-used parts of a program are the best candidates for dynamic segments, since loading and executing a dynamic segment takes longer than executing a static segment. Furthermore, making a large, seldom-used segment dynamic might make the *initial* load of a program faster, since the static part of the load file will be smaller.

Dynamic segments can be used as **overlays** (segments with the same fixed starting address that successively occcupy the same memory area), but this structure is not recommended for the Cortland. If all segments are instead relocatable, the Memory Manager has more flexibility in finding the best place for each allocated segment, whether or not it happens to be a space formerly occupied by another segment of the same program.

Programming With run-time libraries

> **Note:** Although the System Loader supports run-time libraries, initial releases of other Cortland system software may not. This section discusses how to program for run-time libraries when full support for them becomes available.

A run-time library is a load file. Like other libraries or subroutine files, it contains general routines that may be referenced by a program. As with other libraries, references to it are resolved by the linker.

Unlike other libraries, however, its segments are not physically *appended* to the program that references it; instead, the linker creates a reference to it in the program's load file. The run-time library remains on disk (or in memory) as an independent load file; when one of its segments is referenced during program execution, the segment is then loaded and executed dynamically.

As with dynamic segments, loading of run-time library segments is transparent to the typical application. No System Loader commands are necessary to invoke it; as far as the loader is concerned, the run-time library is just another load file with dynamic segments.

The most useful difference between run-time library segments and other dynamic segments is that they may be *shared* among programs. Routines for drawing or calculating, dialog boxes or graphic images, or any other segments that might be of use to more than one program can be put into run-time libraries. And, being dynamic, they help keep the initial load file small.

> **Important:** In using both run-time libraries and other dynamic segments, make sure that the volumes containing all needed segments and libraries are *on line* at run time. A fatal error occurs if the System Loader cannot find a dynamic segment it needs to load.

User control of segment loading

To make the greatest use of the System Loader, programs may make loader calls directly. For most applications this is not necessary, but for programs with specialized needs the System Loader offers this capability.

Your application can manually load other segments using the Load Segment By Number and Load Segment By Name calls. Load Segment By Number requires the application to know the load file number and segment number of the segment to load; Load Segment By Name uses the load file pathname and segment name of the desired segment. Both require UserID as an input; the UserID for each segment and each pathname are available from the Memory Segment Table and Pathname Table, respectively. Other segment information available through the Get Load Segment Info call.

One advantage of manually loading a dynamic segment is that it can be referenced in a more direct manner. Automatically-loaded dynamic segments can be referenced only through a `JSL` to the Jump Table; however, if the segment is data such as a table of values, you may wish to simply access those values rather than pass execution to the segment. By manually loading the segment, locking it, and dereferencing its memory handle (obtaining a pointer to the start of the segment), you may then directly reference any location in the table. Of course, since the loader does not resolve any symbolic references in the manually loaded segment, the application must know its exact structure.

Note: Manually-loaded dynamic segments on the Cortland can be used for the same purposes as *resource files* on the Macintosh.

A program is responsible for managing the segments it loads. That is, it must unload them (using Unload Segment By Number) or make them purgeable and unlocked (through Memory Manager calls) when they are no longer needed.

Designing a controlling program

A program may cause the loading of another program in one of two ways:

- The program can make a ProDOS 16 `QUIT` call. ProDOS 16 and the System Loader remove the quitting program from memory, then load and execute the specified new program.
- The program can call the System Loader directly. The loader loads the specified new program without unloading the original program, then hands control back to the original program.

A **controlling program** is an application that loads and executes other programs using method 2. It uses powerful System Loader calls that are normally reserved for use by ProDOS 16. Certain types of finders, switchers and shells may be controlling programs; if you are writing such a program you should follow the conventions given here.

An application needs to be a controlling program only if it must *remain* in memory after it calls another program. If it is necessary only that control *return* to the original program after the called program quits, the ProDOS 16 `QUIT` call is sufficient for that. For example, a finder, which always returns after an application it calls quits, does not have to be a controlling program; it is not in memory while the application is running. On the other hand, the Cortland Programmer's Workshop Shell, which has functions needed by the subprograms that it calls, *is* a controlling program; it remains active in memory while its subprograms execute.

If a controlling program loads another program, it is totally responsible for the subprogram's disposition. It should pass execution to the subprogram with a JSL. Likewise, the subprogram must not end with a `QUIT`; it should return to the controlling program with an `RTL`.***shell load file requirements are not yet finalized***

The subprogram is first loaded using the System Loader's Initial Load function; the function returns the subprogram's starting address and UserID to the controlling program. The controlling program can then decide when and where to pass control to the subprogram.

When the subprogram is finished, the controlling program is responsible for removing it from memory. The best way to do this is to call the User Shutdown function.

Shutting down and restarting applications

Through alternate use of the User Shutdown and Restart functions, a controlling program can rapidly switch execution among several applications. If none of an application's static segments have been removed from memory since shutdown, Restart rapidly brings the application back because disk access is not required.

However, only software that is *reentrant* can be restarted in this way. Reentrant software reinitializes its variables every time it gains control; it makes no assumptions about the state of the machine it will find when it starts up. The controlling program is responsible for deciding whether a program can be restarted.

Summary: loader calls categorized

The following table categorizes System Loader calls by the types of programs that make them. Most applications, whether their segments are static or dynamic, and whether or not they use run-time libraries, need make none of these calls. Applications that load dynamic segments manually may call any of the *user-callable* functions. Controlling programs and ProDOS 16 call the *system-wide* functions. Only the System Loader itself may call the *internal* functions. Functions not listed in Table 17-1 either do nothing or are executed only at system startup.

Table 17-1. System Loader functions categorized by caller

User-Callable	**System-Wide**	**Internal**
Loader Version	Initial Load	Jump Table Load
Loader Status	Restart	Cleanup
Load Segment By Number	User Shutdown	
Unload Segment By Number		
Load Segment By Name		
Unload Segment		
Get Load Segment Info		

Chapter 18

System Loader Calls

Introduction

This chapter explains how System Loader functions are called, and describes the following calls:

Number	Function	Purpose
$01	Loader Initialization	(executed at system startup)
$02	Loader Startup	(no function)
$03	Loader Shutdown	(no function)
$04	Loader Version	returns System Loader version
$05	Loader Reset	(no function)
$06	Loader Status	returns initialization status
$09	Initial Load	loads an application
$0A	Restart	restarts a dormant application
$0B	Load Segment By Number	loads a single segment
$0C	Unload Segment By Number	unloads a single segment
$0D	Load Segment By Name	loads a single segment
$0E	Unload Segment	unloads a single segment
$0F	Get Load Segment Info	returns a segment's handle
$12	User Shutdown	makes an application dormant
--	Jump Table Load	loads a dynamic segment
--	Cleanup	frees memory space

How calls are made

The System Loader is a Cortland Tool Set (tool number 17, or hexadecimal $11). You call its functions using the standard (non-macro) Cortland tool calling sequence:

1. Push any required space for returned results onto the stack.
2. Push each input value onto the stack, in the proper order.
3. Execute the following call block:

```
LDX    #$11+FuncNum|8
JSL    Dispatcher
```

where

> **#$11** is the System Loader Tool number
> **FuncNum** is the number of the function being called
> (|8 means "shift left by 8 bits".)
> **Dispatcher** is the address of the Tool Dispatcher ($E1 00 00).

It is the responsibility of the controlling program (caller) to prepare the stack for each function it calls, and to pull any results off the stack. Error status is returned in the accumulator (A register); furthermore, the carry bit is set (1) if the call is unsuccessful, and cleared (0) if the call is sucessful.

The Jump Table Load function does not use the above calling sequence, and cannot be called directly by an application. It is called indirectly, through a call to a Jump Table entry. The absolute address of the function is patched into the Jump Table by the System Loader at load time.

Parameter types

There are four types of parameters passed in the stack: values, results, pointers, and handles. Each is either an *input to* or an *output from* the loader function being called.

- A **value** is a numerical quantity, either 2 bytes (**word**) or 4 bytes (**long word**) in length, that the caller passes to the System Loader. It is an input parameter.
- A **result** is a numerical quantity, either 2 bytes (**word**) or 4 bytes (**long word**) in length, that the System Loader passes back to the caller. It is an output parameter.
- A **pointer** is the address of a location containing data, code, or buffer space in which the System Loader can receive or place data. A pointer may be 2 bytes (**word**) or 4 bytes (**long word**) in length. The pointer itself, and the data it points to, may be either input or output.
- A **handle** is a special type of pointer: it is a pointer to a pointer. It is the 4-byte address of a location that *itself* contains the address of a location containing data, code, or buffer space. In System Loader calls, a handle is always an output.

Format for System Loader call descriptions

The following sections describe the System Loader calls in detail. Each description contains these elements:

- the full name of the call
- a brief description of what function it performs
- the call's function number
- the call's assembly-language macro name
- the call's parameter list (input and output)

- the stack configuration both before and after making the call
- a list of possible error codes
- the sequence of events the call invokes (if the brief description is not complete enough).
- code examples,where appropriate

Parameter list note: In the parameter lists, *input* parameters are listed in the order in which they are pushed *onto* the stack; *output* parameters are listed in the order in which they are pulled *from* the stack. Check the stack diagrams if you are uncertain of the proper order in which to push any of the parameters.

Stack diagram note: Unlike most other memory tables in this manual, the stack diagrams are organized in units of *words*—that is, each tick mark represents *two bytes* of stack space.

Loader Initialization ($01)

This routine initializes the System Loader; it is called by the system software at startup (boot) time. It clears all loader tables and sets the initial state of the system, making no assumptions about the current or previous state of the machine. The System Loader's global variables (see Appendix D) are defined at this time.

The Initialization routine is required for all Cortland Tools.

Function Number: $01

Macro Name: none

Parameters:

(none)

Possible Errors:

(none)

Loader Startup ($02)

The Startup routine is required for all Cortland Tools. For the System Loader, this function does nothing and need never be called.

Function Number: $02

Macro Name: `LoaderStartup`

Parameters:

(none)

Possible Errors:

(none)

Loader Shutdown ($03)

The Shutdown routine is required for all Cortland Tools. For the System Loader, this function does nothing and need never be called.

Function Number: $03

Macro Name: `LoaderShutdown`

Parameters:

(none)

Possible Errors:

(none)

Loader Version ($04)

The Loader Version function returns the version number of the System Loader currently in use.

The Version routine is required for all Cortland Tools.

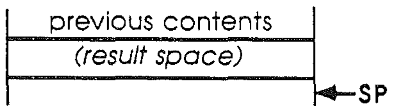
Function Number: $04

Macro Name: `LoaderVersion`

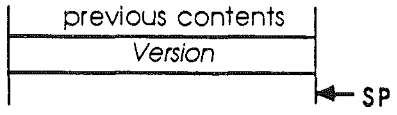
Parameters:

	Parameter Name	Size and Type
Input:	(none)	
Output:	Loader version	word result (2 bytes)

Stack Before Call:



Stack After Call:



Possible Errors:

(none)

Loader Reset ($05)

The Reset routine is required for all Cortland Tools. For the System Loader, this function does nothing and need never be called.

Function Number: $05

Macro Name: `LoaderReset`

Parameters:

(none)

Possible Errors:

(none)

Loader Status ($06)

This routine returns the current status (initialized or uninitialized) of the System Loader. A nonzero result means TRUE (initialized); a zero result means FALSE (uninitialized). A result of TRUE is always returned by this call because the System Loader is always in the initialized state.

The Status routine is required for all Cortland Tools.

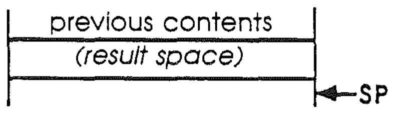
Function Number: $06

Macro Name: `LoaderStatus`

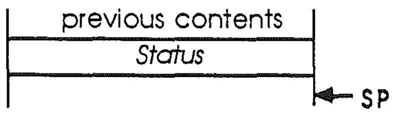
Parameters:

	Parameter Name	Size and Type
Input:	(none)	
Output:	status	word result (2 bytes)

Stack Before Call:



Stack After Call:



Possible Errors:

(none)

Initial Load ($09)

This function is called by a controlling program (such as a shell or a switcher) to ask the System Loader to perform an initial load of a program.

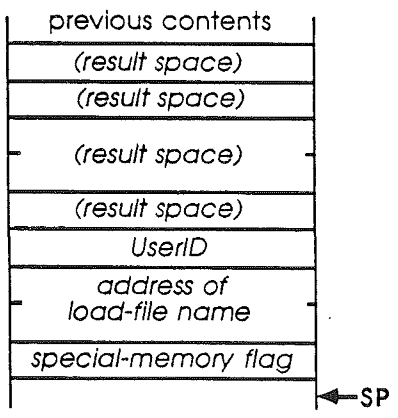
Function Number: $09

Macro Name: `InitialLoad`

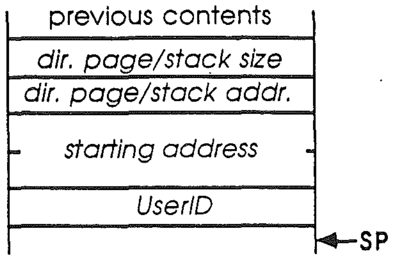
Parameters:

	Parameter Name	**Size and Type**
Input:	UserID	word value (2 bytes)
	address of load-file pathname	long word pointer (4 bytes)
	special-memory flag	word value (2 bytes)
Output:	UserID	word result (2 bytes)
	starting address	long word pointer (4 bytes)
	address of direct page/ stack buffer	word pointer (2 bytes)
	size of direct page/ stack buffer	word result (2 bytes)

Stack Before Call:



Stack After Call:



Possible Errors:

$1104	File is not a load file
$1105	System Loader is busy
$1109	SegNum out of sequence
$110A	Illegal load record found
$110B	Load segment is foreign
$00*xx*	ProDOS 16 error
$02*xx*	Memory Manager error

Sequence of Events:

When the Initial Load function is called, the following sequence of events occurs.

1. The function checks the *Type ID* and *Main ID* parts of the specified UserID.
   a. If both parts of the specified UserID are nonzero, the System Loader uses it to allocate space for the segments to be loaded.
   b. If the Type ID part of the specified UserID is zero,the System Loader obtains a new UserID from the UserID Manager, to assign to all segments of that file. The new Type ID is given the value 1, meaning that the new file is classified as an application.
   c. If only the Main ID part of the specified UserID is zero, the System Loader obtains a new UserID from the UserID Manager, using the supplied Type and Aux ID's.

   The UserID Manager (described in *Cortland Toolbox Reference*) guarantees that UserID's are unique to each application, tool, desk accessory, and so forth. See Appendix D of this manual for a brief description of the UserID format and Type ID's.
2. The function checks the value of the special-memory flag. If it is TRUE (nonzero), the System Loader will not load any *static* segments into special memory (banks $00 and $01—see Chapter 3). The special-memory flag does not affect the load addresses of dynamic segments.
3. The function calls ProDOS 16 to open the specified (by pathname) load file. If any ProDOS 16 error occurs, or if the file is not a load file (type $B3-$BE), the System Loader returns the appropriate error code.

Note: If the load file is a ProDOS 8 system file (type $FF) or a ProDOS 8 binary file (type $06), the loader will not load it.

4. Once the load file is opened, the System Loader adds the load-file information to the Pathname Table, and calls the Load Segment By Number function for each static segment in the load file.
   - If any static segment loaded is an Initialization Segment (segment kind=$10), the System Loader immediately transfers control to it. When the System Loader regains control, it loads the rest of the static segments without passing control to them.
   - If a direct-page/stack segment (`KIND`=$92) is loaded, the System Loader returns the segment's starting address and size.

Note: The System Loader treats a direct-page/stack segment as a locked, dynamic segment. The segment cannot be moved or purged as long is the application is active, but it *is* purged at shutdown.

- If any of the static segments cannot be loaded, the System Loader aborts the load and returns the error from the Load Segment By Number function.

4. Once it has loaded all the static segments, the System Loader returns the starting address of the first segment (other than an initialization segment) of load file 1 to the controlling program. It then transfers execution to the controlling program. The controlling program itself is responsible for setting the stack and direct registers and for transferring control to the just-loaded program.

Restart ($0A)

This function is called by a controlling program (such as a shell or a switcher) to ask the System Loader to resurrect an application that has been shut down (by the User Shutdown function), but is still in memory.

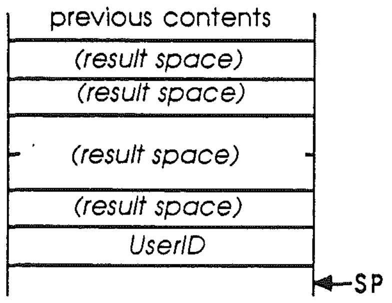
Function Number: $0A

Macro Name: `Restart`

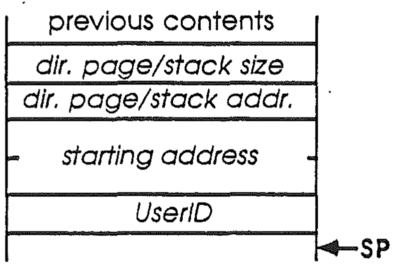
Parameters:

	Parameter Name	Size and Type
Input:	UserID	word value (2 bytes)
Output:	UserID	word result (2 bytes)
	starting address	long word pointer (4 bytes)
	address of direct page/ stack buffer	word pointer (2 bytes)
	size of direct page/ stack buffer	word result (2 bytes)

Stack Before Call:



Stack After Call:



Possible Errors:

$1101	Application not found
$1105	System Loader is busy
$1108	UserID error
$00*xx*	ProDOS 16 error
$02*xx*	Memory Manager error

Sequence of Events:

When the Restart function is called, the following sequence of events occurs.

1. An existing, nonzero UserID must be specified (the Aux ID part is ignored). If the UserID is zero, error $1108 is returned. If the UserID is unknown to the System Loader, error $1101 is returned.
2. The Restart function can work only if all of the specified program's static segments are still in memory. What that means is that no segments in the Memory Segment Table with the specified UserID can have been purged.
   a. The System Loader checks the memory handle of each Memory Segment Table entry with that UserID. If none are set to NIL the segments are all in memory.
   b. The System Loader then resurrects the application by calling the Memory Manager to make each of the application's segments unpurgeable and locked.
   c. The application's complete UserID, the first segment's starting address, and the direct page and stack information (from the Pathname Table) are returned to the caller.
3. If any of the application's static segments are no longer in memory, the function does the following:
   a. It calls the Cleanup routine to purge all references to that UserID (and any other unused UserID's) from the System Loader's tables.
   b. It calls the UserID Manager to delete that UserID.
   c. Finally, it calls the Initial Load function to load the application. The application receives a new UserID, which is returned to the user.

Load Segment By Number ($0B)

The Load Segment By Number routine is the workhorse function of the System Loader. Other System Loader functions that load segments do so by calling this function. It loads a specific load segment into memory; the segment is specified by its load-file and load-segment numbers, and UserID.

> **Note:** Applications use this function to manually load dynamic segments. An application may also use Load Segment By Number to manually load a *static* segment. However, in that case the System Loader does *not* patch the correct address of the newly loaded segment onto any existing references to it. Therefore the segment can be accessed only through its starting address.

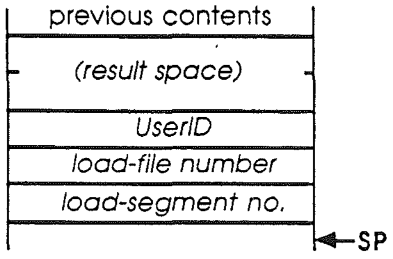
Function Number: $0B

Macro Name: `LoadSegNum`

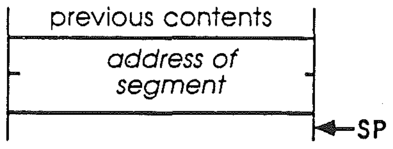
Parameters:

	Parameter Name	Size and Type
Input:	UserID	word value (2 bytes)
	load-file number	word value (2 bytes)
	load-segment number	word value (2 bytes)
Output:	address of segment	long word pointer (4 bytes)

Stack Before Call:



Stack After Call:



Possible Errors:

$1101	Segment not found
$1104	File is not a load file
$1105	System Loader is busy
$1107	File version error
$1109	SegNum out of sequence
$110A	Illegal load record found
$110B	Load segment is foreign
$00*xx*	ProDOS 16 error
$02*xx*	Memory Manager error

Sequence of Events:

When the Load Segment By Number function is called, the following sequence of events occurs.

1. First the loader checks to find out if the requested load segment is already in memory: it searches the Memory Segment Table to determine if there is an entry for the segment. If the entry exists, the loader checks the value of the memory handle to find out whether the corresponding memory block is still in memory. If so, the function terminates without returning an error. If an entry exists but the memory block has been purged, the entry is deleted.
2. If the segment is not already in memory, the System Loader looks in the Pathname Table to get the load-file pathname from the load-file number. If the Pathname Table has a partial pathname for this entry, the System Loader completes the pathname according to ProDOS 16 conventions (see Chapter 5).
3. The System Loader checks the file type of the referenced file. If it is not a load file (type $B3-$BE), then error $1104 is returned.
4. If the file is type $B4 (run-time library file), the System Loader compares the file's modification date and time values to the file date and file time in the Pathname Table. If they do not match, error $1107 is returned and the load is not performed.
5. ProDOS 16 is called to open the file. If ProDOS 16 cannot open the file, it returns an appropriate error code.
6. After ProDOS 16 successfully opens the load file, the System Loader searches the file for a load segment corresponding to the specified load-segment number. If none is found, error $1101 is returned.

   If the load segment is found, its header is checked (segment headers are described under "Object Module Format" in *Cortland Programmer's Workshop Reference*). If the segment is not dynamic, error $1102 is returned. If the value in the header's `SEGNUM` field does not match the specified load-segment number, error $1109 is returned. If the values in the `NUMSEX` and `NUMLEN` fields are not 0 and 4, respectively, error $110B is returned.
7. If the load segment is found and the header is correct, a memory block of the size specified in the `LENGTH` field of the segment header is requested from the Memory Manager. If the `ORG` field in the segment header is not zero, then a memory block

starting at the address specified by ORG is requested (ORG is normally zero for Cortland; that is, most segments are relocatable). Other segment attributes are set according to values in other segment header fields—see Chapter 15.

8. If a nonzero UserID is specified, the memory block is given that UserID. If the specified UserID is zero, the memory block is given the current UserID (value of USERID global variable).
9. If the requested memory is not available, the Memory Manager and System Loader use these techniques to free space:
    a. The Memory Manager unloads unneeded segments by purging their corresponding memory blocks. Blocks are purged according to their *purge levels*. For example, all level-3 blocks are purged before the first level-2 block is purged. Any dynamic segment whose memory block's purge level is zero cannot be unloaded.
    b. If all purgeable segments have been unloaded and the Memory Manager still cannot allocate enough memory, it moves any *movable* blocks to enlarge contiguous memory areas.
    c. If all eligible memory blocks have been purged or moved, and the Memory Manager still cannot allocate enough memory, the System Loader Cleanup routine is called to free any unused parts of the System Loader's memory. The Memory Manager then tries once more to allocate the requested memory.
    d. If the Memory Manager is still unsuccessful, the System Loader returns the last Memory Manager error that occurred.
10. Once the Memory Manager has allocated the requested memory, the System Loader puts the load segment into memory, and processes the relocation dictionary (if any). It patches location-dependent code according to the load address of the segment, and replaces external references to dynamic segments with references to the Jump Table.

Note: If any records within the segment are not of a proper type ($E2, $E3, $F1, $F2, or $00), error $110A is returned. See Appendix D for an explanation of record types.

11. An entry for the segment is added to the Memory Segment Table.
12. The System Loader returns the starting address of the segment to the controlling program.

Example of a Segment Search

Load segments in a load file are numbered sequentially starting at 1. To find a load segment of a particular number, the System Loader scans through the segments in order, counting until it reaches the one it is searching for. Although each segment has a segment number in its header, the loader uses that number only as a check, not as the means by which it identifies the number of the segment.

To find load segment number 5, for example, the System Loader must first scan through the first four load segments. Each load segment header must be processed, because load segments vary in length. The scanning process is facilitated by the fact that (a) load segments start on block boundaries, and (b) the number of blocks in each load segment is given in the first field of the segment header (the BLKCNT field).

The following code sample shows the steps involved in loading the first block of a specified load segment:

```
SegNum:=load-segment number;
FileID:=load file;
open(FileID);
Block:=0;
for i:=1 to SegNum do
   begin
      seek(FileID,Block);             {find block}
      get(FileID);                    {read block}
      Block:=Block+FileID^.BLKCNT     {add BLKCNT to block}

   end;
```

Unload Segment By Number ($0C)

This function unloads a specific load segment from memory; the segment is specified by its load-file and load-segment numbers,and UserID.

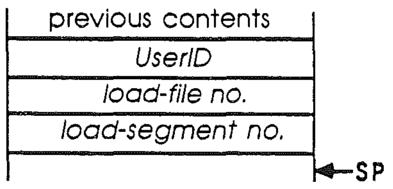
Function Number: $0C

Macro Name: `UnLoadSegNum`

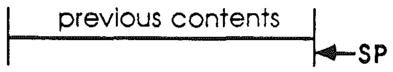
Parameters:

	Parameter Name	Size and Type
Input:	UserID	word value (2 bytes)
	load-file number	word value (2 bytes)
	load-segment number	word value (2 bytes)
Output:	(none)	

Stack Before Call:



Stack After Call:



Possible Errors:

$1101	Segment not found
$1105	System Loader is busy
$00*xx*	ProDOS 16 error
$02*xx*	Memory Manager error

Sequence of Events:

When the Unload Segment By Number function is called, the following sequence of events occurs.

1. The System Loader searches the Memory Segment Table for the specified load-file number and load-segment number. If there is no such entry, error $1101 is returned.
2. If the Memory Segment Table entry is found, the loader calls the Memory Manager to make *purgeable* (purge level = 3) the memory block in which the dynamic segment resides .
3. The loader changes all entries in the Jump Table that reference the unloaded segment to their unloaded states.

Special conditions:

- If the specified UserID is zero, the current UserID (value of `USERID`) is assumed.
- If both the load-file number and load-segment number are nonzero, the specified segment is unloaded regardless of whether it is static or dynamic. If either input is zero, only *dynamic* segments are unloaded, as noted next.
- If the specified load-file number is zero, all dynamic segments for that UserID are unloaded.
- If the specified load-segment number is zero, all dynamic segments for the specified load file are unloaded.

Note: If a *static* segment is unloaded, the application that it is part of cannot be restarted from a dormant state. See "Restart" and "User Shutdown," in this chapter.

Load Segment By Name ($0D)

This function loads a *named* segment into memory. The segment is named by its load file's pathname, and its segment name (from the `SEGNAME` field in the segment header). A nonzero UserID may be specified if the loaded segment is to have a UserID different from the current UserID.

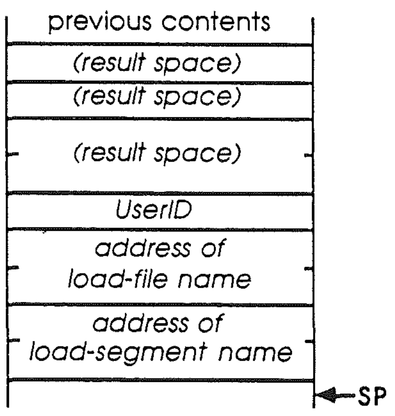
Function Number: $0D

Macro Name: `LoadSegName`

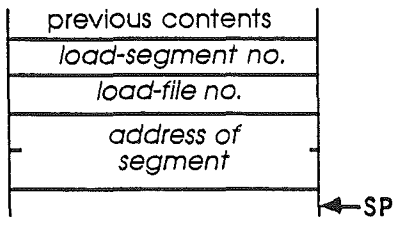
Parameters:

	Parameter Name	**Size and Type**
Input:	UserID	word value (2 bytes)
	address of load-file name	long word pointer (4 bytes)
	address of load-segment name	long word pointer (4 bytes)
Output:	address of segment	long word pointer (4 bytes)
	load-file number	word result (2 bytes)
	load-segment number	word result (2 bytes)

Stack Before Call:



Stack After Call:



Possible Errors:

$1101	Segment not found
$1104	File is not a load file
$1105	System Loader is busy
$1107	File version error
$1109	SegNum out of sequence
$110A	Illegal load record found
$110B	Load segment is foreign
$00*xx*	ProDOS 16 error
$02*xx*	Memory Manager error

Sequence of Events:

When the Load Segment By Name function is called, the following sequence of events occurs.

1. The System Loader gets the load-file pathname from the pointer given in the function call.
2. The System Loader checks the file type of the referenced file, from the file's disk directory entry. If it is not a load file (type $B3–$BE), error $1104 is returned.
3. If it is a load file, the loader calls ProDOS 16 to open the file. If ProDOS 16 cannot open the file, it returns the appropriate error code.
4. After the load file has been successfully opened by ProDOS 16, the System Loader searches the file for a segment with the specified name. If it finds none, error $1101 is returned. If the load segment is found but it is not a dynamic segment, error $1102 is returned.
5. If the load segment is found, the System Loader notes the segment number. It also checks the Pathname Table to see if the load file is listed. If it is, it gets the load file number from the table; if not, it adds a new entry to the Pathname Table, assigning an unused file number to the load file.
6. Now that it has both the load-file number and the segment number of the requested segment, the System Loader calls the Load Segment By Number function to load the segment. If the Load Segment By Number function returns an error, the Load Segment By Name function returns the same error. If the Load Segment By Number function is successful, the Load Segment By Name function returns the load file number, the load segment number, and the starting address of the memory block in which the load segment was placed.

Unload Segment ($0E)

This function unloads the load segment containing the specified address. By using Unload Segment, an application can unload a segment without having to know its load-segment number, load-file number, name or UserID.

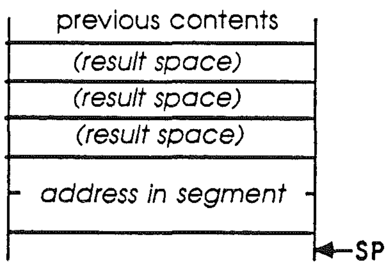
Function Number: $0E

Macro Name: `UnloadSeg`

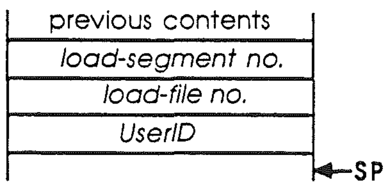
Parameters:

	Parameter Name	Size and Type
Input:	address in segment	long word pointer (4 bytes)
Output:	UserID	word result (2 bytes)
	load-file number	word result (2 bytes)
	load-segment number	word result (2 bytes)

Stack Before Call:



Stack After Call:



Possible Errors:

$1101	Segment not found
$1105	System Loader is busy
$00*xx*	ProDOS 16 error
$02*xx*	Memory Manager error

Sequence of Events:

When the Unload Segment function is called, the following sequence of events occurs.

1. The function calls the Memory Manager to identify the memory block containing the specified address. If the address is not within an allocated memory block, error $1101 is returned.
2. If the memory block is found, the function uses the memory handle returned by the Memory manager to find the block's UserID. It then scans the Memory Segment Table for an entry with that UserID and handle. If no such entry is found, error $1101 is returned.
3. If the Memory Segment Table entry is found, the function does one of two things:
    a. If the Memory Segment Table entry refers to any segment other than a Jump Table segment, the function extracts the load-file number and load-segment number from the entry.
    b. If the Memory Segment Table entry refers to a Jump Table segment, the function extracts the load-file number and load-segment number in the *Jump Table entry* at the address specified in the function call.
4. The function then calls the Unload Segment By Number function to unload the segment.

The outputs of this function (load-file number, load-segment number, and UserID) can be used as inputs to other System Loader functions such as Load Segment By Number.

Get Load Segment Info ($0F)

This function returns the Memory Segment Table entry corresponding to the specified (by number) load segment.

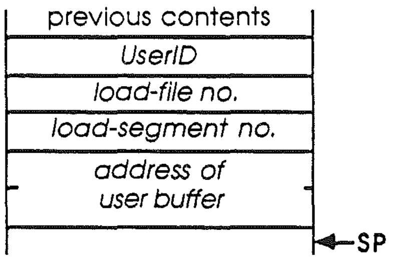
Function Number: $0F

Macro Name: `GetLoadSegInfo`

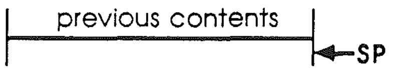
Parameters:

	Parameter Name	Size and Type
Input:	UserID	word value (2 bytes)
	load-file number	word value (2 bytes)
	load-segment number	word value (2 bytes)
	address of user buffer	long word pointer (4 bytes)
Output:	(filled user buffer)	

Stack Before Call:



Stack After Call:



Possible Errors:

$1101	Entry not found
$1105	System Loader is busy
$00*xx*	ProDOS 16 error
$02*xx*	Memory Manager error

Sequence of Events:

When the Get Load Segment Info function is called, the following sequence of events occurs.

1. The Memory Segment Table is searched for the specified entry. If the entry is not found, error $1101 is returned.
2. If the entry is found, the contents of the entry (except for the link pointers) are copied into the user buffer.

User Shutdown ($12)

This function is called by the controlling program to close down an application that has just terminated.

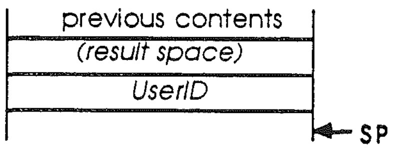
Function Number: $12

Macro Name: `UserShutdown`

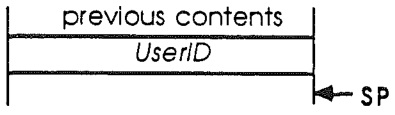
Parameters:

	Parameter Name	Size and Type
Input:	UserID	word value (2 bytes)
Output:	UserID	word result (2 bytes)

Stack Before Call:



Stack After Call:



Possible Errors:

$1105	System Loader is busy
$00*xx*	ProDOS 16 error
$02*xx*	Memory Manager error

Sequence of Events:

When the User Shutdown function is called, the following sequence of events occurs.

1. The System Loader checks the specified UserID. If it is zero, the System Loader assumes it is the current UserID (= value of `USERID` global variable).
2. Using the UserID (with Aux ID set to zero), the loader calls the Memory Manager to purge all *dynamic* segments' memory blocks with that UserID. The segments are purged regardless of their purge level and whether or not they are locked.
3. The loader again calls the Memory Manager, to make all *static* segments with the specified UserID purgeable.

The application is now in a *dormant* state—disconnected but not gone. It may be resurrected very quickly by the System Loader because all its static segments are still in memory. Once any one of its static segments is purged by the Memory Manager, however, it is truly lost and must be reloaded from its load file if it is ever needed again.

Jump Table Load

This function is called by an *unloaded* Jump Table entry in order to load a dynamic load segment. Besides the function call, the unloaded Jump Table entry includes the load-file number and load-segment number of the dynamic segment to be loaded. The Jump Table is described in Chapter 16.

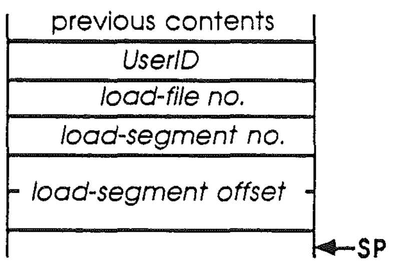
Function Number: none

Macro Name: none

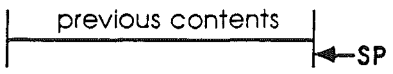
Parameters:

	Parameter Name	Size and Type
Input:	UserID	word value (2 bytes)
	load-file number	word value (2 bytes)
	load-segment number	word value (2 bytes)
	load-segment offset	long word value (4 bytes)
Output:	(none)	

Stack Before Call:



Stack After Call:



Possible Errors:

$1101	Segment not found
$1104	File is not a load file
$1105	System Loader is busy
$00*xx*	ProDOS 16 error
$02*xx*	Memory Manager error

Note: Because this function is never called directly by a controlling program, the program need not know what parameters it requires.

Sequence of Events:

When the Jump Table Load function is called, the following sequence of events occurs.

1. The function calls the Load Segment By Number function, using the load-file number and load-segment number in the Jump Table entry. If the Load Segment By Number function returns any error, the System Loader considers it a fatal error and calls the System Death Manager.
2. If the Load Segment By Number function successfully loads the segment, the Jump Table Load function changes the Jump Table entry to its *loaded* state: it replaces the JSL to the Jump Table Load function with a JML to the absolute address of the reference in the just-loaded segment.
3. The function transfers control to the address of the reference.

Cleanup

This routine is used to free additional memory when needed. It scans the Memory Segment Table and removes all entries that reference purged segments.

Note: Because this function is never called directly by a controlling program, the program need not know what parameters it requires.

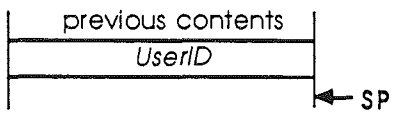
Function Number: none

Macro Name: none

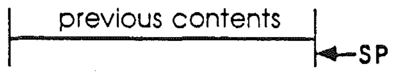
Parameters:

	Parameter Name	Size and Type
Input:	UserID	word value (2 bytes)
Output:	(none)	

Stack Before Call:



Stack After Call:



Possible Errors:

(none)

Sequence of Events:

When the Cleanup routine is called, the following sequence of events occurs.

1. If the specified UserID is 0:
   a. The System Loader scans all entries in the Memory Segment Table.

   b. All dynamic segments for all UserID's are purged.

2. If the specified UserID is nonzero:

   a. The System Loader scans all entries in the Memory Segment Table with that UserID.

   b. *All* load segments (both dynamic *and* static) for that UserID are purged.

   b. All entries in the Pathname Table and Memory Segment Table for that UserID are deleted.

   c. The UserID itself is then removed through a call to the UserID Manager.

Chapter 19

System Loader Error Codes

Nonfatal errors

The System Loader returns errors in the same way ProDOS 16 does. Upon return from a call, the carry bit reflects the error status (set to 1 if an error occurred, cleared to 0 if no error), and the accumulator contains the error code ($0000 if no error).

If a ProDOS 16 or Memory Manager error occurs during the execution of a System Loader call, the appropriate error code is placed in the accumulator. ProDOS 16 error codes are described in Chapter 14 of this manual; Memory Manager error codes are described in *Cortland Toolbox Reference.*

The following table lists error numbers and gives for each a suggested screen message and a brief description of its significance.

Number	Message and Description
$0000	(no error)
$1101	**Not found:** The specified segment (in the load file) or entry (in the Pathname Table or Memory Segment Table) does not exist. If the specified load file itself is not found, a ProDOS 16 error $46 (file not found) is returned.
$1104	**File is not a load file:** the specified load file is not type $B3-$BE. See Appendix A or D for descriptions of these file types.
$1105	**Loader is busy:** The call cannot be made because the System Loader is busy with another call.
$1107	**File version error:** The specified file cannot be loaded because its creation date and time do not match those on its entry in the Pathname Table.

Note: this error applies to run-time library files only.

Number	Message and Description
$1108	**UserID error:** The specified UserID either doesn't exist (Application Shutdown), or doesn't match the UserID of the specified segment (Unload Segment By Number).

$1109	**SegNum out of sequence:** the value of the `SEGNUM` field in the segment's header doesn't match the number by which the segment was specified (Load Segment By Number, Initial Load).
$110A	**Illegal load record found:** A record in the segment is of a type not accepted by the loader. Load files can consist only of record types $E2 (`RELOC`), $E3 (`INTERSEG`), $F1 (`DS`), $F2 (`LCONST`), and $00 (`END`).
$110B	**Load segment is foreign:** The values in the `NUMSEX` and `NUMLEN` fields in the specified segment's header are not 0 and 4, respectively (Load Segment By Number).
$001-$05F	(ProDOS 16 I/O errors)
$201-$20A	(Memory Manager errors)

Fatal errors

If a ProDOS 16 error or Memory Manager error occurs while the System Loader is making an internal call, it is a fatal error. The most common case is when a Jump Table Load is attempted for a dynamic load segment or run-time library segment whose volume is not on line. Control is transferred to the System Death Manager, and the following message appears on the screen:

```
Error loading Dynamic Segment-XXXX
```

where *XXXX* is the error code of the ProDOS 16 or Memory manager error that occurred.

Appendixes

Appendix A

ProDOS 16 File Organization

This appendix contains a detailed description of the way that ProDOS 16 stores files on disks. For most applications, the operating system insulates you from this level of detail. However, you must use this information if, for example, you want to

- List the files in a directory
- Copy a sparse file without increasing the file's size
- Compare two sparse files

This appendix first explains the organization of information on volumes. Next, it shows the format and organization of volume directories, subdirectories, and the various stages of standard files. Finally it presents a set of diagrams showing the formats of individual header and entry fields.

> **Note:** In this appendix, *format* refers to the arrangement of information (such as headers, pointers and data) within a file. *Organization* refers to the manner in which a single file is stored on disk, in terms of individual 512-byte blocks.

Organization of information on a volume

When a volume is formatted for use with ProDOS 16, its surface is partitioned into an array of tracks and sectors. In accessing a volume, ProDOS 16 requests not a track and sector, but a logical block from the device corresponding to that volume. That device's driver translates the requested block number into the proper track and sector number; the physical location of information on a volume is unimportant to ProDOS 16 and to an application that uses ProDOS 16. This appendix discusses the organization of information on a volume in terms of logical blocks, not tracks and sectors.

When the volume is formatted, information needed by ProDOS 16 is placed in specific logical blocks, starting with the first block (block 0). A **loader program** is placed in blocks 0 and 1 of the volume. This program enables ProDOS 16 (or ProDOS 8) to be booted from the volume. Block 2 of the volume is the **key block** (the first block) of the **volume directory file**; it contains descriptions of (and pointers to) all the files in the volume directory. The volume directory occupies a number of consecutive blocks, typically four, and is immediately followed by the **volume bit map**, which records whether each block on the volume is used or unused. The volume bit map occupies consecutive blocks, one for every 4,096 blocks, or fraction thereof, on the volume. The rest of the blocks on the disk contain subdirectory file information, standard file information, or are empty. The first blocks of a volume look something like Figure A-1.

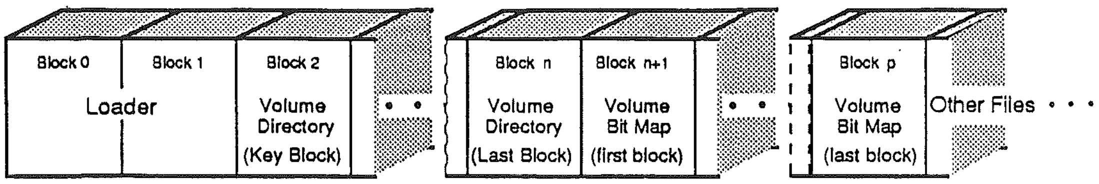

Block 0
Block 1
Block 2
Loader
Volume Directory (Key Block)
Block n
Block n+1
Volume Directory (Last Block)
Volume Bit Map (first block)
Block p
Volume Bit Map (last block)
Other Files

Figure A-1. Block organization of a volume

The precise format of the volume directory, volume bit map, subdirectory files and standard files are explained in the following sections.

Format and organization of directory files

The format and organization of the information contained in volume directory and subdirectory files is quite similar. Each consists of a **key block** followed by zero or more blocks of additional directory information. The fields in a directory's key block are:

- a pointer to the next block in the directory
- a header that describes the directory
- a number of file entries describing, and pointing to, the files in that directory
- zero or more unused bytes.

The fields in subsequent (nonkey) blocks in a directory are:

- pointers to the preceding and succeeding blocks in the directory
- a number of entries describing, and pointing to, the files in that directory
- zero or more unused bytes.

The format of a directory file is represented in Figure A-2.

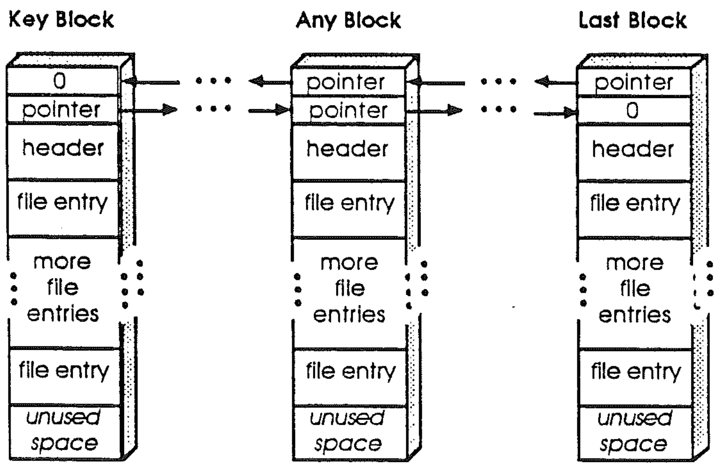


Figure A-2. Directory file format and organization

The header is the same length as all other entries in a directory file. The only difference between a volume directory file and a subdirectory file is in the header format.

Pointer fields

The first four bytes of each block used by a directory file contain pointers to the preceding and succeeding blocks in the directory file, respectively. Each pointer is a two-byte logical block number—low-order byte first, high-order byte second. The key block of a directory file has no preceding block; its first pointer is zero. Likewise, the last block in a directory file has no successor; its second pointer is zero.

> **Note:** The block pointers described in this appendix, which hold *disk* addresses, are two bytes long. All other ProDOS 16 pointers, which hold *memory* addresses, are four bytes long. In either case, ProDOS 16 pointers are always stored with the low-order byte first and the high-order byte last. See Chapter 2, "ProDOS 16 and Cortland Memory."

Volume directory headers

Block 2 of a volume is the key block of that volume's directory file. The volume directory header is at byte position $0004 of the key block, immediately following the block's two pointers. Thirteen fields are currently defined to be in a volume directory header: they contain all the vital information about that volume. Figure A-3 illustrates the structure of a volume directory header. Following Figure A-3 is a description of each of its fields.

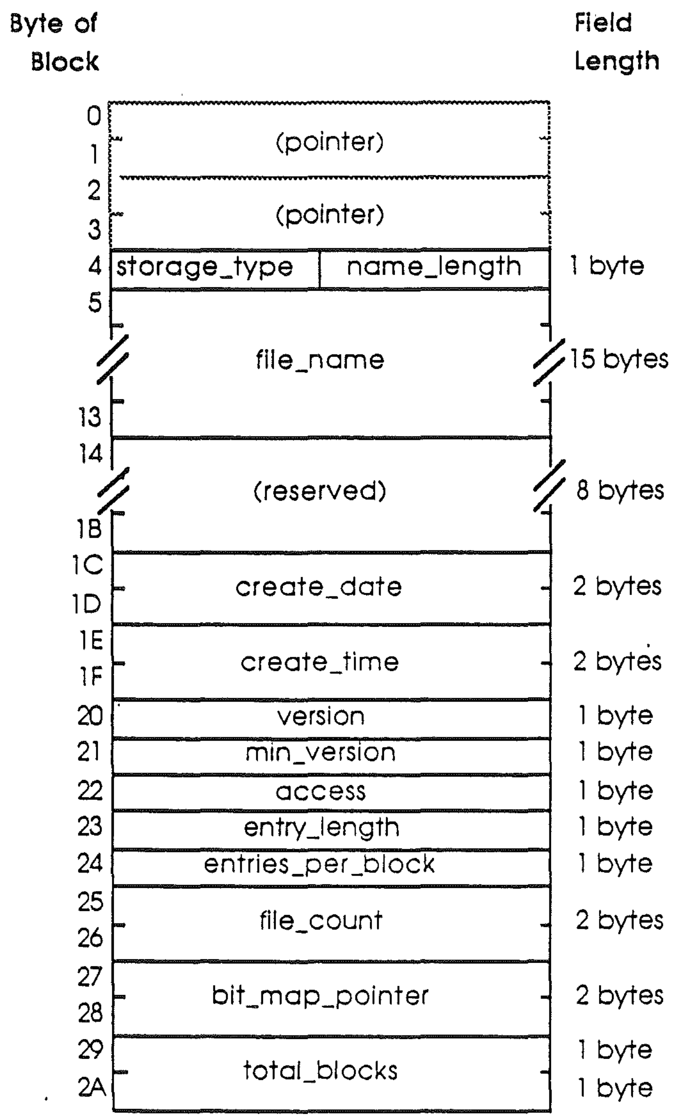


Figure A-3. The volume directory header

storage_type and **name_length** (1 byte): Two four-bit (nibble) fields are packed into this byte. A value of $F in the high-order nibble (`storage_type`) identifies the current block as the key block of a volume directory file. The low-order nibble contains the length of the volume's name (see the `file_name` field, below). The value of `name_length` can be changed by a `CHANGE_PATH` call.

file_name (15 bytes): The first *n* bytes of this field, where *n* is the value of `name_length`, contain the volume's name. This name must conform to the file name (volume name) syntax explained in Chapter 2. The name does not begin with the slash that usually precedes volume names. This field can be changed by the `CHANGE_PATH` call.

reserved (8 bytes): Reserved for future expansion of the file system.

create_date (2 bytes): The date on which this volume was initialized. The format of these bytes is described under "Header and Entry Fields," later in this appendix.

create_time (2 bytes): The time at which this volume was initialized. The format of these bytes is described under "Header and Entry Fields," later in this appendix.

version (1 byte): The file system version number of ProDOS 8 or ProDOS 16 under which the file pointed to by this entry was created. This byte allows newer versions of ProDOS 16 to determine the format of the file, and adjust their interpretation processes accordingly. For ProDOS 16, `version` = 0.

> **Note:** Version in this sense refers to the *file system* version only. At present, all ProDOS operating systems use the same file system and therefore have the same file system version number. The file system version number is unrelated to the *program* version number returned by the `GET_VERSION` call.

min_version: Reserved for future use. For ProDOS 16, it is 0.

access (1 byte): Determines whether this volume directory can be read, written, destroyed, or renamed. The format of this field is described under "Header and Entry Fields," in this Appendix.

entry_length (1 byte): The length in bytes of each entry in this directory. The volume directory header itself is of this length. For ProDOS 16, `entry_length` = $27.

entries_per_block (1 byte): The number of entries that are stored in each block of the directory file. For ProDOS 16, `entries_per_block` = $0D.

file_count (2 bytes): The number of active file entries in this directory file. An active file is one whose `storage_type` is not 0. Figure A-5 shows the format of file entries.

bit_map_pointer (2 bytes): The block address of the first block of the volume's bit map. The bit map occupies consecutive blocks, one for every 4,096 blocks (or fraction thereof) on the volume. You can calculate the number of blocks in the bit map using the `total_blocks` field, described below.

The bit map has one bit for each block on the volume: a value of 1 means the block is free; 0 means it is in use. If the number of blocks used by all files on the volume is not the same as the number recorded in the bit map, the directory structure of the volume has been damaged.

total_blocks (2 bytes): The total number of blocks on the volume.

Subdirectory headers

The key block of every subdirectory file is pointed to by an entry in a parent directory; for example, by an entry in a volume directory (Figure A-2). A subdirectory's header begins at byte position $0004 of the key block of that subdirectory file, immediately following the two pointers.

Its internal structure is quite similar to that of a volume directory header (only its last three fields are different). There are fourteen fields in a subdirectory header: they contain all the vital information about that subdirectory. Figure A-4 illustrates the structure of a subdirectory header. A description of all the fields in the header follows the figure.

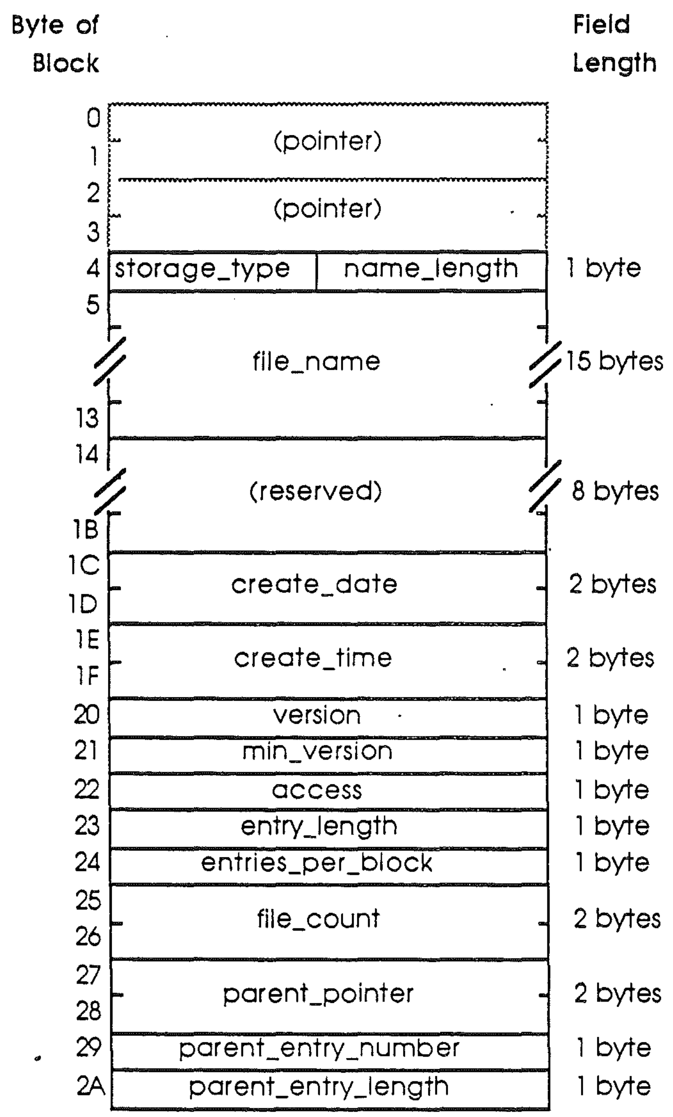


Figure A-4. The subdirectory header

storage_type and **name_length** (1 byte): Two four-bit (nibble) fields are packed into this byte. A value of $E in the high-order nibble (`storage_type`) identifies the current block as the key block of a subdirectory file. The low-order nibble contains the length of the subdirectory's name (see the `file_name` field, below). The value of `name_length` can be changed by a `CHANGE_PATH` call.

file_name (15 bytes): The first *name_length* bytes of this field contain the subdirectory's name. This name must conform to the file name syntax explained in Chapter 2. This field can be changed by the `CHANGE_PATH` call.

reserved (8 bytes): Reserved for future expansion of the file system.

create_date (2 bytes): The date on which this subdirectory was created. The format of these bytes is described under "Header and Entry Fields," later in this appendix.

create_time (2 bytes): The time at which this subdirectory was created. The format of these bytes is described under "Header and Entry Fields," later in this appendix.

version (1 byte): The file system version number of ProDOS 8 or ProDOS 16 under which the file pointed to by this entry was created. This byte allows newer versions of ProDOS 16 to determine the format of the file, and adjust their interpretation processes accordingly. For ProDOS 16, `version` = 0.

> **Note:** Version in this sense refers to the *file system* version only. At present, all ProDOS operating systems use the same file system and therefore have the same file system version number. The file system version number is unrelated to the *program* version number returned by the `GET_VERSION` call.

min_version (1 byte): The minimum version number of ProDOS 8 or ProDOS 16 that can access the information in this file. This byte allows older versions of ProDOS 8 and ProDOS 16 to determine whether they can access newer files. For ProDOS 16, `min_version` = 0.

access (1 byte): Determines whether this subdirectory can be read, written, destroyed, or renamed, and whether the file needs to be backed up. The format of this field is described under "Header and Entry Fields," in this Appendix. A subdirectory's access byte can be changed by the `SET_FILE_INFO` and `CLEAR_BACKUP_BIT` calls.

entry_length (1 byte): The length in bytes of each entry in this subdirectory. The subdirectory header itself is of this length. For ProDOS 16, `entry_length` = $27.

entries_per_block (1 byte): The number of entries that are stored in each block of the directory file. For ProDOS 16, `entries_per_block` = $0D.

file_count (2 bytes): The number of active file entries in this subdirectory file. An active file is one whose `storage_type` is not 0. See "File Entries" for more information about file entries.

parent_pointer (2 bytes): The block address of the directory file block that contains the entry for this subdirectory. This and all other two-byte pointers are stored low-order byte first, high-order byte second.

parent_entry_number (1 byte): The entry number for this subdirectory within the block indicated by `parent_pointer`.

parent_entry_length (1 byte): The `entry_length` for the directory that owns this subdirectory file. Note that with these last three fields you can calculate the precise position on a volume of this subdirectory's file entry. For ProDOS 16, `parent_entry_length` = $27.

File Entries

Immediately following the pointers in any block of a directory file are a number of entries. The first entry in the key block of a directory file is a header, all other entries are file entries. Each entry has the length specified by that directory's `entry_length` field, and each file entry contains information that describes, and points to, a single subdirectory file or standard file.

An entry in a directory file may be active or inactive, that is, it may or may not describe a file currently in the directory. If it is inactive, the first byte of the entry (`storage_type` and `name_length`) has the value zero.

The maximum number of entries, including the header, in a block of a directory is recorded in the `entries_per_block` field of that directory's header. The total number of active file entries, not including the header, is recorded in the `file_count` field of that directory's header.

Figure A-5 describes the format of a file entry.

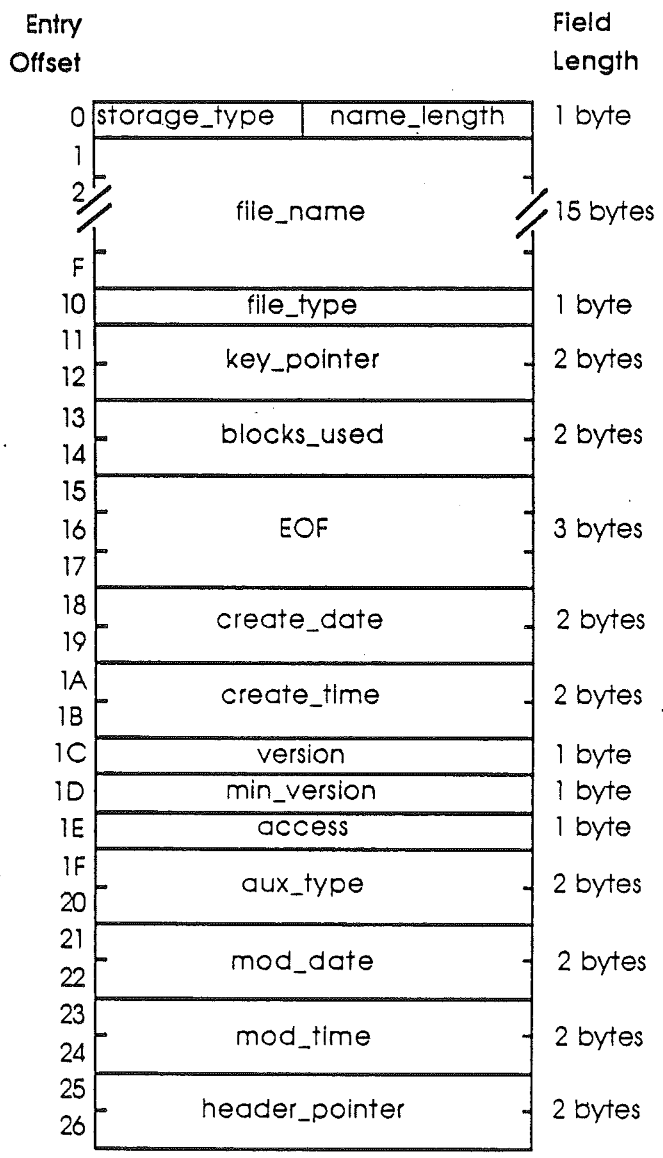


Figure A-5. The file entry

storage_type and **name_length** (1 byte): Two four-bit (nibble) fields are packed into this byte. The value in the high-order nibble (`storage_type`) specifies the type of file pointed to by this file entry:

$1	= Seeding file
$2	= Sapling file
$3	= Tree file
$4	= Pascal area
$D	= Subdirectory

Seedling, sapling, and tree files are described under "Format and Organization of Standard Files," in this Appendix. The low-order nibble contains the length of the file's name (see the `file_name` field, below). The value of `name_length` can be changed by a `CHANGE_PATH` call.

file_name (15 bytes): The first *name_length* bytes of this field contain the file's name. This name must conform to the file name syntax explained in Chapter 2. This field can be changed by the `CHANGE_PATH` call.

file_type (1 byte): A descriptor of the internal structure of the file. Table A-1 (at the end of this appendix) is a list of the currently defined values of this byte.

key_pointer (2 bytes): The block address of:

• the master index block	(if the file is a tree file)
• the index block	(if the file is a sapling file)
• the data block	(if the file is a seedling file)

blocks_used (2 bytes): The total number of blocks actually used by the file. For a subdirectory file, this includes the blocks containing subdirectory information, but not the blocks in the files pointed to. For a standard file, this includes both informational blocks (index blocks) and data blocks. See "Format and Format and Organization of Standard Files" in this Appendix.

EOF (3 bytes): A three-byte integer, lowest byte first, that represents the total number of bytes readable from the file. Note that in the case of sparse files, EOF may be greater than the number of bytes actually allocated on the disk.

create_date (2 bytes): The date on which the file pointed to by this entry was created. The format of these bytes is described under "Header and Entry Fields," later in this appendix.

create_time (2 bytes): The time at which the file pointed to by this entry was created. The format of these bytes is described under "Header and Entry Fields," later in this appendix.

version (1 byte): The file system version number of ProDOS 8 or ProDOS 16 under which the file pointed to by this entry was created. This byte allows newer versions of ProDOS 16 to determine the format of the file, and adjust their interpretation processes accordingly. For ProDOS 16, `version` = 0.

> **Note:** Version in this sense refers to the *file system* version only. At present, all ProDOS operating systems use the same file system and therefore have the same

file system version number. The file system version number is unrelated to the *program* version number returned by the GET_VERSION call.

min_version (1 byte): The minimum version number of ProDOS 8 or ProDOS 16 that can access the information in this file. This byte allows older versions of ProDOS 8 and ProDOS 16 to determine whether they can access newer files. For ProDOS 16, min_version = 0.

access (1 byte): Determines whether this file can be read, written, destroyed, or renamed, and whether the file needs to be backed up. The format of this field is described under "Header and Entry Fields," later in this appendix. The value of this field can be changed by the SET_FILE_INFO and CLEAR_BACKUP_BIT calls. You cannot delete (destroy) a subdirectory that contains any files.

aux_type (2 bytes): A general-purpose field in which an application can store additional information about the internal format of a file. For example, the ProDOS 8 BASIC system program uses this field to record the load address of a BASIC program or binary file, or the record length of a text file.

mod_date (2 bytes): The date on which the last CLOSE operation after a WRITE was performed on this file. The format of these bytes is described under "Header and Entry Fields," later in this appendix. This field can be changed by the SET_FILE_INFO call.

mod_time (2 bytes): The time at which the last CLOSE operation after a WRITE was performed on this file. The format of these bytes is described under "Header and Entry Fields," later in this appendix. This field can be changed by the SET_FILE_INFO call.

header_pointer (2 bytes): This field is the block address of the key block of the directory that owns this file entry. This and all two-byte pointers are stored low-order byte first, high-order byte second.

Reading a directory file

This section deals with the general techniques of reading from directory files, not with the specifics. The ProDOS 16 calls with which these techniques can be implemented are explained in Chapters 9 and 10.

Before you can read from a directory, you must know the directory's pathname. With the directory's pathname, you can open the directory file, and obtain a reference number (*ref_num*) for that open file. Before you can process the entries in the directory, you must read three values from the directory header:

- Length of each entry in the directory (*entry_length*)
- Number of entries in each block of the directory (*entries_per_block*)
- Total number of files in the directory (*file_count*).

Using the reference number to identify the file, read the first 512 bytes from the file, and into a buffer (ThisBlock in the example below). The buffer contains two two-byte pointers, followed by the entries; the first entry is the directory header. The three values are at positions $1F through $22 in the header (positions $23 through $26 in the buffer).

In the example below, these values are assigned to the variables `EntryLength`, `EntriesPerBlock`, and `FileCount`.

```
Open(DirPathname, RefNum);                {Get reference number     }
ThisBlock       := Read512Bytes(RefNum); {Read a block into buffer}
EntryLength     := ThisBlock[$23];       {Get directory info      }
EntriesPerBlock := ThisBlock[$24];
FileCount       := ThisBlock[$25] + (256 * ThisBlock[$26]);
```

Once these values are known, an application can scan through the entries in the buffer, using a pointer (`EntryPointer`) to the beginning of the current entry , a counter (`BlockEntries`) that indicates the number of entries that have been examined in the current block, and a second counter (`ActiveEntries`) that indicates the number of active entries that have been processed.

An entry is active and is processed only if its first byte, the `storage_type` and `name_length`, is nonzero. All entries have been processed when *ActiveEntries* is equal to *FileCount*. If all the entries in the buffer have been processed, and *ActiveEntries* doesn't equal *FileCount*, then the next block of the directory is read into the buffer.

```
EntryPointer    := EntryLength + $04;         {Skip header entry}
BlockEntries    := $02;              {Prepare to process entry two}
ActiveEntries   := $00;              {No active entries found yet }

while ActiveEntries < FileCount do begin
      if ThisBlock[EntryPointer] <> $00 then begin  {Active entry}
           ProcessEntry(ThisBlock[EntryPointer]);
           ActiveEntries := ActiveEntries + $01
      end;
      if ActiveEntries < FileCount then  {More entries to process}
           if BlockEntries = EntriesPerBlock
                then begin            {ThisBlock done. Do next one}
                     ThisBlock    := Read512Bytes(RefNum);
                     BlockEntries := $01;
                     EntryPointer := $04
                end
                else begin            {Do next entry in ThisBlock }
                     EntryPointer := EntryPointer + EntryLength;
                     BlockEntries := BlockEntries + $01
                end
end;
Close(RefNum);
```

This algorithm processes entries until all expected active entries have been found. If the directory structure is damaged, and the end of the directory file is reached before the proper number of active entries has been found, the algorithm fails.

Format and organization of standard files

Each active entry in a directory file points to the key block (the first block) of a file. As shown below, the key block of a standard file may have several types of information in it. The `storage_type` field in that file's entry must be used to determine the contents of the key block. This section explains the structure of the three stages of standard file: seedling, sapling, and tree. These are the files in which all programs and data are stored.

Growing a tree file

The following scenario demonstrates the *growth* of a tree file on a volume. This scenario is based on the block allocation scheme used by ProDOS 16 on a 280-block flexible disk that contains four blocks of volume directory, and one block of volume bit map. Larger capacity volumes might have more blocks in the volume bit map, but the process would be identical.

A formatted, but otherwise empty, ProDOS 16 volume is used like this:

Blocks 0-1	Loader
Blocks 2-5	Volume directory
Block 6	Volume bit map
Blocks 7-279	Unused

If you open a new file of a nondirectory type, one data block is immediately allocated to that file. An entry is placed in the volume directory, and it points to block 7, the new data block, as the key block for the file. The key block is indicated below by an arrow.

The volume now looks like this:

	Blocks 0-1	Loader
	Blocks 2-5	Volume directory
	Block 6	Volume bit map
—>	**Block 7**	**Data block 0**
	Blocks 8-279	Unused

This is a **seedling file**: its key block contains up to 512 bytes of data. If you write more than 512 bytes of data to the file, the file *grows* into a **sapling file**. As soon as a second block of data becomes necessary, an **index block** is allocated, and it becomes the file's key block: this index block can point to up to 256 data blocks (it uses two-byte pointers). A second data block (for the data that won't fit in the first data block) is also allocated.

The volume now looks like this:

	Blocks 0-1	Loader
	Blocks 2-5	Volume directory
	Block 6	Volume bit map
	Block 7	**Data block 0**
—>	**Block 8**	**Index block 0**
	Block 9	**Data block 1**
	Blocks 10-279	Unused

This sapling file can hold up to 256 data blocks: 128K of data. If the file becomes any bigger than this, the file *grows* again, this time into a **tree file**. A **master index block** is allocated, and it becomes the file's key block: the master index block can point to up to 128 index blocks, and each of these can point to up to 256 data blocks. Index block 0 becomes the first index block pointed to by the master index block. In addition, a new index block is allocated, and a new data block to which it points.

Here's a new picture of the volume:

	Blocks 0-1	Loader
	Blocks 2-5	Volume directory
	Block 6	Volume bit map
	Block 7	**Data block 0**
	Block 8	**Index block 0**
	Blocks 9-263	**Data blocks 1-255**
—>	**Block 264**	**Master index block**
	Block 265	**Index block 1**
	Block 266	**Data block 256**
	Blocks 267-279	Unused

As data is written to this file, additional data blocks and index blocks are allocated as needed, up to a maximum of 129 index blocks (one a master index block), and 32,768 data blocks, for a maximum capacity of 16,777,215 bytes of data in a file. If you did the multiplication, you probably noticed that a byte was lost somewhere. The last byte of the last block of the largest possible file cannot be used becauseEOF cannot exceed 16,777,216. If you are wondering how such a large file might fit on a small volume such as a flexible disk, refer to the description of sparse files in this appendix.

This scenario shows the growth of a single file on an otherwise empty volume. The process is a bit more confusing when several files are growing—or being deleted—simultaneously. However, the block allocation scheme is always the same: when a new block is needed, ProDOS 16 always allocates the first unused block in the volume bit map.

Seedling files

A **seedling file** is a standard file that contains no more than 512 data bytes ($0 <= EOF <= $200). This file is stored as one block on the volume, and this data block is the file's key block.

The organization of such a seedling file appears in Figure A-6.

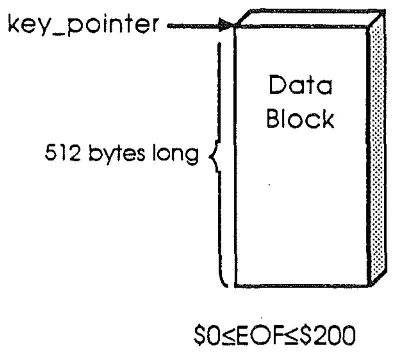


Figure A-6. Format and organization of a seedling file

The file is called a seedling file because it is the smallest possible ProDOS 16 standard file; if more than 512 data bytes are written to it, it grows into a sapling file, and thence into a tree file.

The `storage_type` field of a directory entry that points to a seedling file has the value $1.

Sapling files

A **sapling file** is a standard file that contains more than 512 and no more than 128K bytes ($200 < **EOF** <= $20000). A sapling file comprises an index block and 1 to 256 data blocks. The index block contains the block addresses of the data blocks. See Figure A-7.

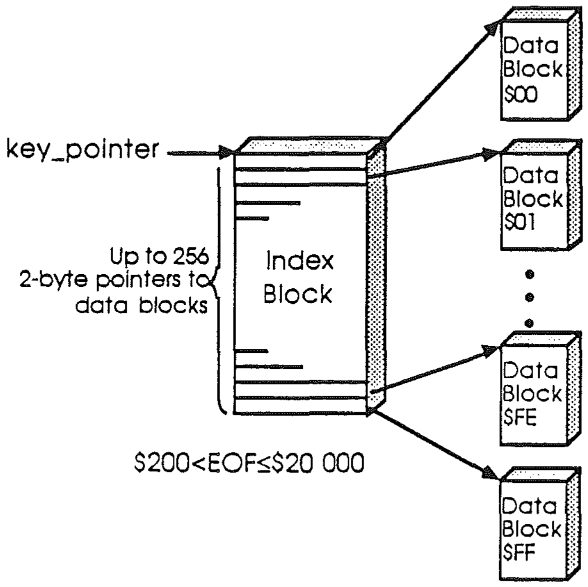


Figure A-7. Format and organization of a sapling file

The key block of a sapling file is its index block. ProDOS 16 retrieves data blocks in the file by first retrieving their addresses in the index block.

The `storage_type` field of a directory entry that points to a sapling file has the value $2.

Tree files

A **tree file** contains more than 128K bytes, and less than 16Mb ($20000 < **EOF** < $1000000). A tree file consists of a master index block, 1 to 128 index blocks, and 1 to 32,768 data blocks. The master index block contains the addresses of the index blocks, and each index block contains the addresses of up to 256 data blocks. The structure of a tree file is shown in Figure A-8.

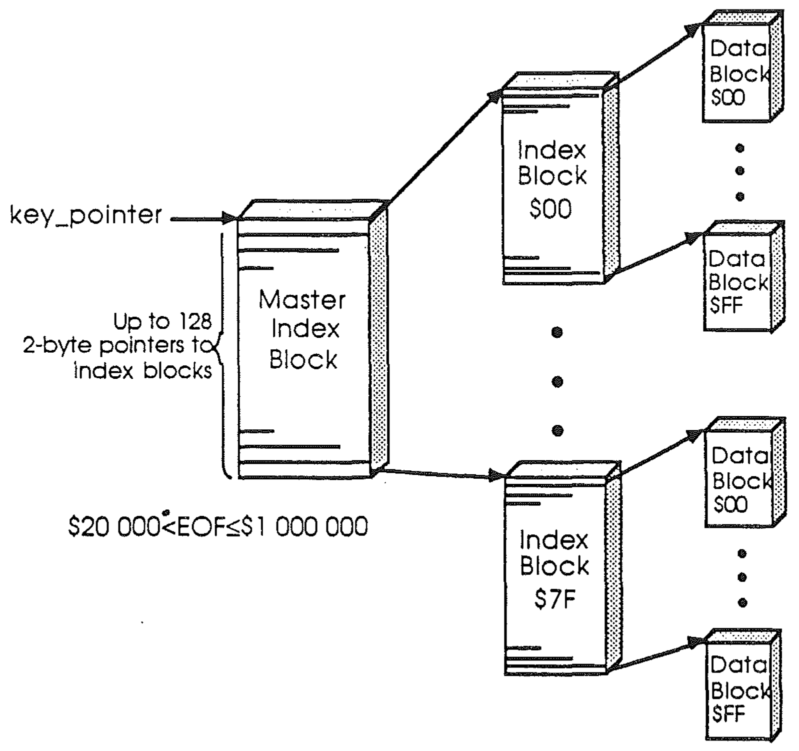


Figure A-8. Format and organization of a tree file

The key block of a tree file is the master index block. By looking at the master index block, ProDOS 16 can find the addresses of all the index blocks; by looking at those blocks, it can find the addresses of all the data blocks.

The `storage_type` field of a directory entry that points to a tree file has the value $3.

Using standard files

An application program operates the same on all three types of standard files, although the `storage_type` in the file's entry can be used to distinguish between the three. A program rarely reads index blocks or allocates blocks on a volume: ProDOS 16 does that. The program need only be concerned with the data stored in the file, not with how they are stored.

All types of standard files are read as a sequence of bytes, numbered from 0 to (EOF–1), as explained in Chapter 2.

Sparse files

A **sparse file** is a sapling or tree file in which the number of data bytes that can be read from the file exceeds the number of bytes physically stored in the data blocks allocated to the file. ProDOS 16 implements sparse files by allocating only those data blocks that have had data written to them, as well as the index blocks needed to point to them.

For example, you can define a file whose **EOF** is 16K, that uses only three blocks on the volume, and that has only four bytes of data written to it. Refer to figure A-9 during the following explanation.

1. If you create a file with an **EOF** of $0, ProDOS 16 allocates only the key block (a data block) for a seedling file, and fills it with null characters (ASCII $00).
2. If you then set the **EOF** and **MARK** to position $0565, and write four bytes, ProDOS 16 calculates that position $0565 is byte $0165 ($0564–($0200 * 2)) of the third block (block $2) of the file. It then allocates an index block, stores the address of the current data block in position 0 of the index block, allocates another data block, stores the address of that data block in position 2 of the index block, and stores the data in bytes $0165 through $0168 of that data block. The **EOF** is now $0569.
3. If you now set the **EOF** to $4000 and close the file, you have a 16K sapling file that takes up three blocks of space on the volume: two data blocks and an index block (shaded in figure A-9). You can read 16384 bytes of data from the file, but all the bytes before $0565 and after $0568 are nulls.

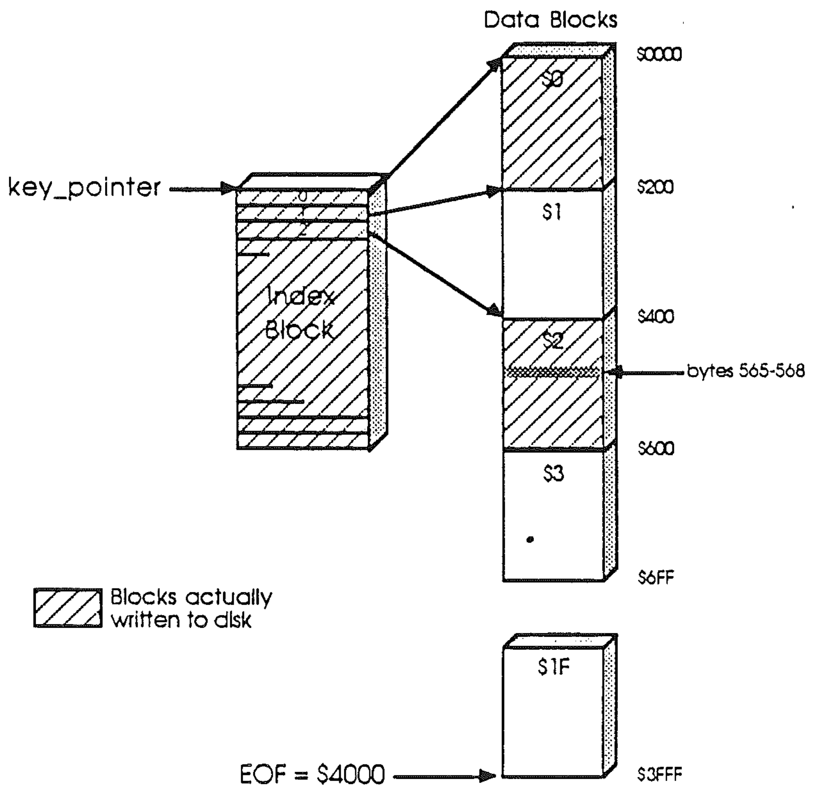


Figure A-9. An example of sparse file organization

Thus ProDOS 16 allocates volume space only for those blocks in a file that actually contain data. For tree files, the situation is similar: if none of the 256 data blocks assigned to an index block in a tree file have been allocated, the index block itself is not allocated.

Note: The first data block of a standard file, be it a seedling, sapling, or tree file, is always allocated. Thus there is always a data block to be read in when the file is opened.

Locating a byte in a file

This is how to find a specific byte within a standard file:

The MARK is a three-byte value that indicates an absolute byte position within a file. If the file is a tree file, then the high-order seven bits of the MARK determine the number (0 to 127) of the index block that points to the byte. The value of the seven bits indicates the location of the low byte of the index block address within the master index block. The location of the high byte of the index block address is indicated by the value of these seven bits plus 256.

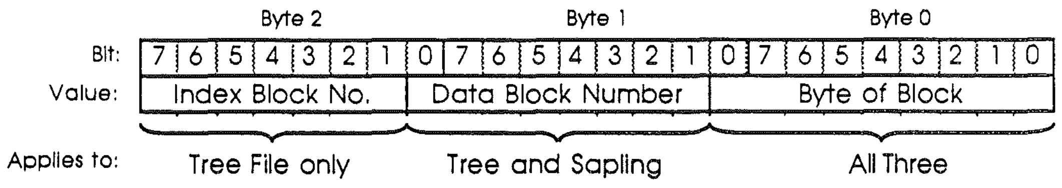


Figure A-10. MARK format

If the file is a tree file or a sapling file, then the next eight bits of the MARK determine the number (0-255) of the data block pointed to by the indicated index block. This 8-bit value indicates the location of the low byte of the data block address within the index block. The high byte of the index block address is found at this offset plus 256.

For tree, sapling, and seedling files, the low nine bits of the MARK are the absolute position of the byte within the selected data block.

Header and entry fields

The storage type attribute

The value in the `storage_type` field, the high-order four bits of the first byte of an entry, defines the type of header (if the entry is a header) or the type of file described by the entry. Table A-1 lists the currently defined storage type values.

Table A-1. Storage type values

$0	indicates an inactive file entry
$1	indicates a seedling file entry (EOF <= 256 bytes)
$2	indicates a sapling file entry (256 < EOF <= 128K bytes)
$3	indicates a tree file entry (128K < EOF < 16M bytes)

$4	indicates a Pascal operating system area on a partitioned disk
$D	indicates a subdirectory file entry
$E	indicates a subdirectory header
$F	indicates a volume directory header

ProDOS 16 automatically changes a seedling file to a sapling file and a sapling file to a tree file when the file's EOF grows into the range for a larger type. If a file's EOF shrinks into the range for a smaller type, ProDOS 16 changes a tree file to a sapling file and a sapling file to a seedling file.

The creation and last-modification fields

The date and time of the creation and last modification of each file and directory is stored as two four-byte values, as shown in Figure A-11.

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	Year							Month				Day				

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	0	0	0	Hour					0	0	Minute					

Figure A-11. Date and time format

The values for the year, month, day, hour, and minute are stored as binary integers, and may be unpacked for conversion to normal integer values.

The access attribute

The access attribute field, or access byte (Figure A-12), determines whether the file can be read from, written to, deleted, or renamed. It also contains a bit that can be used to indicate whether a backup copy of the file has been made since the file's last modification.

Bit:	7	6	5	4	3	2	1	0
Value:	D	RN	B	reserved			W	R

where

D = destroy-enable bit
RN = rename-enable bit
B = backup-needed bit
W = write-enable bit
R = read-enable bit

Figure A-12. Access byte format

A bit set to 1 indicates that the operation is enabled; a bit cleared to 0 indicates that the operation is disabled. The reserved bits are always 0. The most typical setting for the access byte is $C3 (11000011).

ProDOS 16 sets bit 5, the **backup bit,** to 1 whenever the file is changed (that is, after a `CREATE`, `RENAME`, `CLOSE` after `WRITE`, or `SET_FILE_INFO` operation). This bit should be reset to 0 whenever the file is duplicated by a backup program.

> **Note:** Only ProDOS 16 may change bits 2-4; only backup programs should clear bit 5 (using `CLEAR_BACKUP_BIT`).

The file type attribute

The `file_type` field in a directory entry identifies the type of file described by that entry. This field should be used by applications to guarantee file compatibility from one application to the next. The currently recognized values of this byte are listed in Table A-2.

Table A-2. Apple file types

File Type	Preferred Use
$00	Uncategorized file (SOS and ProDOS 8)
$01	Bad block file
$02 †	Pascal code file
$03 †	Pascal text file
$04	ASCII text file (SOS and ProDOS 8)
$05 †	Pascal data file
$06	General binary file (SOS and ProDOS 8)
$07 †	Font file
$08	Graphics screen file
$09 †	Business BASIC program file
$0A †	Business BASIC data file
$0B †	Word Processor file
$0C †	SOS system file
$0D,$0E †	SOS reserved
$0F	Directory file (SOS and ProDOS 8)
$10 †	RPS data file
$11 †	RPS index file
$12 †	AppleFile discard file
$13 †	AppleFile model file
$14 †	AppleFile report format file
$15 †	Screen Library file
$16–$18 †	SOS reserved
$19	AppleWorks Data Base file
$1A	AppleWorks Word Proc. file
$1B	AppleWorks Spreadsheet file
$1C–$EE	Reserved
$B3	ProDOS 16 application
$B4	ProDOS 16 run-time library file
$B5	ProDOS 16 shell load file
$B6	ProDOS 16 startup load file
$B7–$BE	Reserved for ProDOS 16 load files
$BF	ProDOS 16 document file

$EF	Pascal area on a partitioned disk
$F0	ProDOS 8 CI added command file
$F1–$F8	ProDOS 8 user defined files 1-8
$F9	ProDOS 8 reserved
$FA	Integer BASIC program file
$FB	Integer BASIC variable file
$FC	Applesoft program file
$FD	Applesoft variables file
$FE	Relocatable code file (EDASM)
$FF	ProDOS 8 system file

†apply to Apple III only

Appendix B

Apple II Operating Systems

This appendix explains the relationships between ProDOS 16 and three other operating systems developed for the Apple II family of computers (DOS, ProDOS 8, and Apple II Pascal), as well as two developed for the Apple III (SOS and Apple III Pascal).

If you have written programs for one of the other systems or are planning to write programs concurrently for ProDOS 16 and another system, this appendix may help you see what changes will be necessary to transfer your program from one system to another. If you are converting files from one system to another, this appendix may help you understand why some conversions may be more successful than others.

The first section gives a brief history. The next two sections give general comparisons of the other operating systems to ProDOS 16, in terms of file compatibility and operational similarity.

History

DOS

DOS stands for *Disk Operating System.* It is Apple's first operating system; before DOS, the firmware **monitor program** controlled program execution and input/output.

DOS was developed for the Apple II computer. It provided the first capability for storage and retrieval of various types of files on disk (the Disk II); the System Monitor allowed input/output (of binary data) to cassette tape only.

The latest version of DOS is DOS 3.3. It uses a 16-sector disk format, like ProDOS Earlier versions use a 13-sector format that cannot be read by ProDOS.

SOS

SOS is the operating system developed for the Apple III computer. Its name is an acronym for *Sophisticated Operating System*, reflecting its increased capabilities over DOS. On the other hand, SOS requires far more memory space than either DOS or ProDOS 8 (below), which makes it impractical on computers with less than 256K of RAM.

ProDOS 8

ProDOS 8 (for *Professional Disk Operating System*) was developed for the newer members of the Apple II family of computers. It requires at east 64K of RAM memory, and can run on the Apple IIe, Apple IIc, and 64K Apple II Plus.

ProDOS 8 brings some of the advanced features of SOS to the Apple II family, without requiring as much memory as SOS does. Its commands are essentially a subset of the SOS commands.

The latest version of ProDOS 8 developed specifically for the Apple IIe and IIc is ProDOS 8 (1.1.1). An even more recent version, developed for the Cortland but compatible with the IIe and IIc, is ProDOS 8 (1.2).

> **Note:** Prior to development of ProDOS 16, ProDOS 8 was called simply *ProDOS*.

ProDOS 16

ProDOS 16 is an extended revision of ProDOS 8, developed specifically for the Cortland (it will not run on other Apple II's). The "16" refers to the 16-bit internal registers in the Cortland 65816 microprocessor.

ProDOS 16 permits access to the entire 16 Mb addressable memory space of the Cortland (ProDOS 8 is restricted to addressing 64K) and it has more "SOS-like" features than ProDOS 8 has. It also has some new features, not present in SOS, that ease program development.

There are two versions of ProDOS 16. Version 1.0 is an interim system, consisting of a ProDOS 8 core surrounded by a "ProDOS 16-like" user interface. Version 2.0 is the complete implementation of the ProDOS 16 design.

Pascal

The Pascal operating system for the Apple II is modified and extended from UCSD Pascal, developed at the University of California at San Diego. The latest version, written for the Apple IIe/IIc and 64K Apple II Plus, is Pascal 1.3. It also runs on a Cortland.

Another Pascal, for the Apple III, is a modified version of Apple II Pascal. It uses SOS as its operating system.

File compatibility

ProDOS 16, ProDOS 8, and SOS all use a hierarchical file system with the same format and organization. Every file on one system's disk can be read by either of the other systems. DOS and Pascal use significantly different formats.

The other systems compare to ProDOS 16 as follows:

ProDOS 8: ProDOS 16 and ProDOS 8 have identical file system organizations and recognize the same file types, with these exceptions:

ProDOS 8 does not recognize the file types $B3, $B4, $B5, $B6; these file types are specific to ProDOS 16.

SOS: The SOS file types that are recognized by ProDOS 16 are directory files, text files, and binary files. These three types are adequate for transferring programs and data between SOS and ProDOS 16.

DOS: DOS does not have a hierarchical file system. ProDOS 16 cannot directly read DOS files (but see "Reading DOS 3.3 and Apple II Pascal Disks," below).

Pascal: Apple II Pascal does not have a hierarchical file system. ProDOS 16 cannot directly read Apple II Pascal files (but see "Reading DOS 3.3 and Apple II Pascal Disks," below).

Apple III Pascal uses the SOS file system. Therefore ProDOS 16 can read Apple III Pascal directory files, text files, and binary files***binary files?***.

Reading DOS 3.3 and Apple II Pascal disks

Both DOS 3.3 and ProDOS 8 140K flexible disks are formatted using the same 16-sector layout. As a consequence, the ProDOS 16 `READ_BLOCK` and `WRITE_BLOCK` calls are able to access DOS 3.3 disks too. These calls know nothing about the organization of files on either type of disk.

When using `READ_BLOCK` and `WRITE_BLOCK`, you specify a 512-byte block on the disk. When using `RWTS` (the DOS 3.3 counterpart to `READ_BLOCK` and `WRITE_BLOCK`), you specify the track and sector of a 256-byte chunk of data, as explained in the *DOS Programmer's Manual.* To use `READ_BLOCK` and `WRITE_BLOCK` to access DOS 3.3 disks, you must know what 512-byte block corresponds to the track and sector you want.

Table B-1 shows how to determine a block number from a given track and sector. First multiply the track number by 8, then add the sector offset that corresponds to the sector number. The half of the block in which the sector resides is determined by the half-of-block line (1 is the first half; 2 is the second).

Table B-1. Tracks and sectors to blocks (140K disks)

Block number = (8*track number) + sector offset

Sector:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Sector offset:	0	7	6	6	5	5	4	4	3	3	2	2	1	1	0	7
Half of block:	1	1	2	1	2	1	2	1	2	1	2	1	2	1	2	2

Refer to the *DOS Programmer's Manual* for a description of the file organization of DOS 3.3 disks.

Operating system similarity

Input/Output

ProDOS 16 can perform I/O operations on disk files (block devices) only. Under ProDOS 16, therefore, the current application is responsible for knowing the protocol necessary to communicate with character devices (such as the console, printers, and communication ports).

The other systems compare to ProDOS 16 as follows:

ProDOS 8: Like ProDOS 16, ProDOS 8 performs I/O on block devices only.

SOS: SOS communicates with all devices, both character devices and block devices, by making appropriate *file* access calls (such as open, read write, close). Under SOS, writing to one device is essentially the same as writing to another.

DOS: DOS allows communication with one type of device only—the Disk II drive. DOS 3.3 uses a 16-sector disk format; earlier versions of DOS use a 13-sector format. 13-sector Disk II disks cannot be read directly by DOS 3.3, SOS, ProDOS 8, or ProDOS 16.

Pascal: Apple II and Apple III Pascal provide access to both block devices and character devices, through *File I/O*, *Block I/O*, and *Device I/O* calls to the volumes on the devices.

Filing calls

SOS, ProDOS 8, and ProDOS 16 filing calls are all closely related. Most of the calls are shared by all three systems; furthermore, their numbers are identical in ProDOS 8 and SOS (ProDOS 16 calls have a completely different numbering system from either ProDOS 8 or SOS).

The other systems compare to ProDOS 16 as follows:

ProDOS 8: The ProDOS 8 `ON_LINE` call corresponds to the ProDOS 16 `VOLUME` call. When given a device name, `VOLUME` returns the volume name for that device. When given a unit number (derived from the slot and drive numbers), `ON_LINE` returns the volume name.

The ProDOS 8 `RENAME` call corresponds to the ProDOS 16 `CHANGE_PATH` call, except that `RENAME` can change only the *last* name in a pathname.

SOS: Under SOS (unlike ProDOS 16), you must specify the file size when creating a file. File sizes are not automatically extended when needed.

TheSOS `GET_FILE_INFO` call returns the size of the file (the value of EOF). With ProDOS 16 you must first open the file and then use the `GET_EOF` call.

The SOS `VOLUME` call corresponds to the ProDOS 16 `VOLUME` call. When given a device name, `VOLUME` returns the volume name for that device.

The SOS calls `SET_MARK` and `SET_EOF` can use a displacement from the current position in the file. ProDOS 16 accepts only absolute positions in the file for these calls.

DOS: DOS calls distinguish between *sequential- access* and *random-access* text files. ProDOS 16 makes no such distinction, although the ProDOS 16 READ call in `NEWLINE` mode functions as a sequential-access read.

DOS uses `APPEND` and `POSITION` commands, roughly similar to ProDOS 16's `SET_MARK`, to set the current position in the file and to automatically extend the size of the file.

The `CLOSE` command in DOS can be given in immediate (from the keyboard) or deferred(in a program) mode. No ProDOS 16 commands may be given in immediate mode.

Pascal: Apple II Pascal distinguishes among *text files*, *data files*, and *code files*, each with different header formats; all ProDOS 16 files have identical header formats. The Pascal procedures `REWRITE` and `RESET` correspond to ProDOS 16's `CREATE` and `OPEN` calls. Pascal has more procedures for reading from and writing to files and devices than does ProDOS 16.

Because Apple III Pascal uses the SOS file system, its filing calls correspond directly to SOS calls.

Memory management

Under ProDOS 16, neither the operating system nor the application program perform memory management; allocation of memory is the responsibility of the Memory Manager, a Cortland ROM-based Tool. When an application needs space for its own use, it makes a direct request to the Memory Manager. When it makes a ProDOS 16 call that requires the allocation of memory space, ProDOS 16 makes the appropriate request to the Memory Manager. The Cortland Memory Manager is similar to the SOS memory manager, except that it is more sophisticated and is not considered part of the operating system.

The other systems compare to ProDOS 16 as follows:

ProDOS 8: A ProDOS 8 application is responsible for its own memory management. It must find free memory, and then allocate it by marking it off in the ProDOS 8 Global Page's memory bit map. ProDOS 8 protects allocated areas by refusing to write to any pages that are marked on the bit map. Thus it prevents the user from destroying protected memory areas (as long as all allocated memory is properly marked off; and all data is brought into memory using ProDOS 8 calls).

SOS: SOS has a fairly sophisticated Memory Manager that is part of the operating system itself. An application requests memory from SOS, either by location or by the amount needed. If the request can be satisfied, SOS grants it. That portion of memory is then the sole responsibility of the requestor until it is released.

DOS: DOS performs no memory management. Each application under DOS is completely responsible for its own memory allocation and use.

Pascal: Apple II Pacal uses a simple memory managemênt system that controls the loading and unloading of code and data segments and tracks the size of the stack and heap.

Apple III Pascal uses SOS for memory management.

Interrupts

ProDOS 16 does not have any built-in interrupt-generating device drivers. Interrupt handling routines are therefore installed into ProDOS 16 separately, using the `ALLOC_INTERRUPT` call. When an interrupt occurs, ProDOS 16 polls the handling routines in succession until one of them claims the interrupt.

The other systems compare to ProDOS 16 as follows:

ProDOS 8: ProDOS 8 handles interrupts identically to ProDOS 16, except that it allows fewer installed handlers (4 vs. 16).

SOS: In SOS, any device capable of generating an interrupt must have a device driver capable of handling the interrupt; the device driver and its interrupt handler are inseparable and are considered to be part of SOS. In addition, SOS assigns a distinct interrupt priority to each device in the system.

DOS: DOS does not support interrupts.

Pascal: Apple II Pascal versions 1.2 and 1.3 support interrupts; earlier versions of Apple II Pascal do not.
Apple III Pascal uses the SOS interrupt system.

Appendix C

The ProDOS 16 Exerciser

information not yet available

Appendix D

System Loader Technical Data

This appendix assembles some specific technical details on the System Loader. For more information, see the referenced publications.

Object module format

The System Loader can load only code and data segments that conform to Cortland **Object Module Format.** Object Module format is described in detail in *Cortland Programmer's Worrkshop Reference.*

File types

File types for load files and other OMF-related files are listed below. For a complete list of Apple II file types, see Table A-2 in Appendix A.

File type	**Description**
$B0	Source file (*aux_type* defines language)
$B1	Object file
$B2	Library file
$B3	Application file
$B4	Run-time library file
$B5	Shell load file
$B6 - $BF	*Reserved for system use. Currently-defined types include:*
$B6	Permanent inititialization file
$B7	Temporary initialization file
$B8	New desk accessory
$B9	Classic desk accessory

Segment kinds

Whereas files are classified by type, segments are classified by **kind.** Each segment has a kind designation in the `KIND` field of its header. Each bit in the `KIND` field describes some attribute of the segment; different combinations of these attributes yield different values for the segment kind.

The `KIND` field is one byte long. Figure D-1 shows its format.

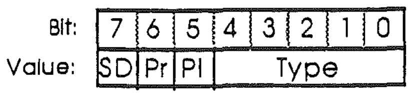


Figure D-1. Segment kind format

where	**SD** (bit 7) =	Static/Dynamic	(0 = static; 1 = dynamic)
	Pr (bit 6) =	Private	(0 = no; 1 = yes)
	PI (bit 5) =	Position-Independent	(0 = no; 1 = yes)

and **TYPE** (bits 0-4) describes one of the following classifications of the segment:

Value of TYPE	**Description**
$00	Code Segment
$01	Data Segment
$02	Jump Table Segment
$04	Pathname Segment
$08	Library Dictionary Segment
$10	Initialization segment
$11	Absolute Bank Segment
$12	Direct Page/Stack Segment

Segment attributes (bits 5-7) can be combined with particular types (bits 0-4) to yield different values for `KIND`. For example, a *dynamic* Initialization Segment has `KIND` = $90.

Record codes

Load segments, like all OMF segments, are made up of **records**. Each type of record has a code number and a name. For a complete list of record types, see *Cortland Programmer's Workshop Reference*. The only record types recognized by the System Loader are these:

Record Code	**Name**	**Description**
$E2	`RELOC`	intrasegment relocation record (in relocation dictionary)
$E3	`INTERSEG`	intersegment relocation record (in relocation dictionary)
$F1	`DS`	zero-fill record
$F2	`LCONST`	long-constant record (the actual code and data for each segment)
$00	`END`	the end of the segment

If the loader encounters any other type of record in a load segment, it returns error $110A.

Load-file numbers

Load files processed by the Cortland Programmer's Workshop Linker at any one time are numbered consecutively from 1. Load file 1 is called the **initial load file.** All other files are considered to be run-time libraries.

A load-file number of 0 in a Jump Table segment or a Pathname segment indicates the end of the segment.

Load-segment numbers

In each load file created by the linker, segments are numbered consecutively by their position in the load file, starting at 1. Load segment 1 must be static. The loader determines a segment's number by counting its position from the beginning of the load file. As a check, the loader also looks at the segment number in the segment's header.

Segment headers

The first part of every Object Module Format segment is a **segment header**; it contains 16 fields that give the name, size, and other important information about the segment.

Restrictions on segment header values

Because OMF supports capabilities that are more general than the System Loader's needs, the System Loader permits load files to have only a subset of all possible OMF characteristics. The loader does this by restricting the values of several segment header fields:

`NUMSEX:`	must be 0
`NUMLEN:`	must be 4
`BANKSIZE:`	must be less than or equal to $10 000
`ALIGN:`	must be less than or equal to $10 000

If the System Loader finds any other values in any of the above fields, it returns error $110B ("Segment is Foreign"). The restrictions on `BANKSIZE` and `ALIGN` are also enforced by the CPW Linker.

Page-aligned and bank-aligned segments

In OMF, the values of `BANKSIZE` and `ALIGN` may be any multiple of 2. But because the Memory Manager supports only two types of alignment (page- and bank-alignment) and one bank size (64K), the System Loader converts `BANKSIZE` and `ALIGN` to a limited set of values, as follows:

1. If `ALIGN` = 0, the block is not aligned to any memory boundary, *except that*:
   - if 0<`BANKSIZE`<=$100, the block is page-aligned
   - if 100<`BANKSIZE`<=$10 000, the block is bank-aligned

2. If 0<`ALIGN`<=$100, the block is page-aligned, *except that*:
   - if 100<`BANKSIZE`<=$10 000, the block is bank-aligned
3. if 1000<`ALIGN`<=$10 000, the block is bank-aligned

Entry point and global variables

There is only one entry point needed for all System Loader calls (actually, all tool calls). It is to the Cortland tool dispatcher, at the bottom of bank $E1 (address $E1 00 00). Although the System Loader maintains memory space with a table of loader functions in other parts of memory, locations in those areas are not supported. Please make all System Loader calls with a JSL to $E1 00 00, as explained in Chapter 18.

The following variables are of global significance. They are defined at the system level, meaning that any application that needs to know their values may access them. However, only `USERID` is important to most applications, and it should be accessed only through proper calls to the System Loader. The other variables are needed by controlling programs only, and should not be used by applications.

`SEGTBL`	Absolute address of the Memory Segment Table
`JMPTBL`	Absolute address of the Jump Table List
`PATHTBL`	Absolute address of the Pathname Table
`USERID`	UserID of the current application

UserID format

The UserID Manager is discussed in Chapter 5, and fully explained in *Cortland Toolbox Reference*. Only the format of the UserID number, needed as a parameter for System Loader calls, is shown here.

There is a 2-byte UserID associated with every allocated memory block. It is divided into three fields: **Main ID**, **Aux ID**, and **Type ID**. The Main ID is the unique number assigned to the owner of the block by the UserID Manager; every allocated block has a nonzero value in its Main ID field. The Aux ID is a user-assignable identification; it is ignored by the System Loader, ProDOS 16, and the UserID Manager. The Type ID gives the general class of software to which the block belongs.

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	Type ID				Aux ID				Main ID							

Figure D-2. UserID format

The Main ID can have any value from $01 to $FF (0 is reserved).

The Aux ID can have any value from $00 to $0F.

Type ID values are defined as follows:

0	Memory Manager
1	Application
2	Controlliing Program
3	ProDOS 8 and ProDOS 16
4	Toolset
5	Desk accessory
6	Run-time library
7	System Loader
8-F	(undefined)

Appendix E

ASCII Tables

Char	Dec	Hex	Binary	Char	Dec	Hex	Binary
nul	0	0	00000000	@	64	40	01000000
soh	1	1	00000001	A	65	41	01000001
stx	2	2	00000010	B	66	42	01000010
etx	3	3	00000011	C	67	43	01000011
eot	4	4	00000100	D	68	44	01000100
enq	5	5	00000101	E	69	45	01000101
ack	6	6	00000110	F	70	46	01000110
bel	7	7	00000111	G	71	47	01000111
bs	8	8	00001000	H	72	48	01001000
ht	9	9	00001001	I	73	49	01001001
lf	10	A	00001010	J	74	4A	01001010
vt	11	B	00001011	K	75	4B	01001011
ff	12	C	00001100	L	76	4C	01001100
cr	13	D	00001101	M	77	4D	01001101
so	14	E	00001110	N	78	4E	01001110
si	15	F	00001111	O	79	4F	01001111
dle	16	10	00010000	P	80	50	01010000
dc1	17	11	00010001	Q	81	51	01010001
dc2	18	12	00010010	R	82	52	01010010
dc3	19	13	00010011	S	83	53	01010011
dc4	20	14	00010100	T	84	54	01010100
nak	21	15	00010101	U	85	55	01010101
syn	22	16	00010110	V	86	56	01010110
etb	23	17	00010111	W	87	57	01010111
can	24	18	00011000	X	88	58	01011000
em	25	19	00011001	Y	89	59	01011001
sub	26	1A	00011010	Z	90	5A	01011010
esc	27	1B	00011011	[	91	5B	01011011
fs	28	1C	00011100	\	92	5C	01011100
gs	29	1D	00011101	]	93	5D	01011101
rs	30	1E	00011110	^	94	5E	01011110
us	31	1F	00011111	_	95	5F	01011111
sp	32	20	00100000	`	96	60	01100000
!	33	21	00100001	a	97	61	01100001
"	34	22	00100010	b	98	62	01100010
#	35	23	00100011	c	99	63	01100011
$	36	24	00100100	d	100	64	01100100
%	37	25	00100101	e	101	65	01100101
&	38	26	00100110	f	102	66	01100110
'	39	27	00100111	g	103	67	01100111
(	40	28	00101000	h	104	68	01101000
)	41	29	00101001	i	105	69	01101001
*	42	2A	00101010	j	106	6A	01101010
+	43	2B	00101011	k	107	6B	01101011
,	44	2C	00101100	l	108	6C	01101100
-	45	2D	00101101	m	109	6D	01101101
.	46	2E	00101110	n	110	6E	01101110
/	47	2F	00101111	o	111	6F	01101111
0	48	30	00110000	p	112	70	01110000
1	49	31	00110001	q	113	71	01110001
2	50	32	00110010	r	114	72	01110010
3	51	33	00110011	s	115	73	01110011
4	52	34	00110100	t	116	74	01110100
5	53	35	00110101	u	117	75	01110101
6	54	36	00110110	v	118	76	01110110
7	55	37	00110111	w	119	77	01110111
8	56	38	00111000	x	120	78	01111000
9	57	39	00111001	y	121	79	01111001
:	58	3A	00111010	z	122	7A	01111010
;	59	3B	00111011	{	123	7B	01111011
<	60	3C	00111100	|	124	7C	01111100
=	61	3D	00111101	}	125	7D	01111101
>	62	3E	00111110	~	126	7E	01111110
?	63	3F	00111111	del	127	7F	01111111

Char	Dec	Hex	Binary	Char	Dec	Hex	Binary
Ä	128	80	10000000	¿	192	C0	11000000
Å	129	81	10000001	¡	193	C1	11000001
Ç	130	82	10000010	¬	194	C2	11000010
É	131	83	10000011	√	195	C3	11000011
Ñ	132	84	10000100	ƒ	196	C4	11000100
Ö	133	85	10000101	≈	197	C5	11000101
Ü	134	86	10000110	∆	198	C6	11000110
á	135	87	10000111	«	199	C7	11000111
à	136	88	10001000	»	200	C8	11001000
â	137	89	10001001	…	201	C9	11001001
ä	138	8A	10001010		202	CA	11001010
ã	139	8B	10001011	À	203	CB	11001011
å	140	8C	10001100	Ã	204	CC	11001100
ç	141	8D	10001101	Õ	205	CD	11001101
é	142	8E	10001110	Œ	206	CE	11001110
è	143	8F	10001111	œ	207	CF	11001111
ê	144	90	10010000	–	208	D0	11010000
ë	145	91	10010001	—	209	D1	11010001
í	146	92	10010010	“	210	D2	11010010
ì	147	93	10010011	”	211	D3	11010011
î	148	94	10010100	‘	212	D4	11010100
ï	149	95	10010101	’	213	D5	11010101
ñ	150	96	10010110	÷	214	D6	11010110
ó	151	97	10010111	◊	215	D7	11010111
ò	152	98	10011000	ÿ	216	D8	11011000
ô	153	99	10011001	Ÿ	217	D9	11011001
ö	154	9A	10011010	⁄	218	DA	11011010
õ	155	9B	10011011	¤	219	DB	11011011
ú	156	9C	10011100	‹	220	DC	11011100
ù	157	9D	10011101	›	221	DD	11011101
û	158	9E	10011110	ﬁ	222	DE	11011110
ü	159	9F	10011111	ﬂ	223	DF	11011111
†	160	A0	10100000	‡	224	E0	11100000
°	161	A1	10100001	·	225	E1	11100001
¢	162	A2	10100010	‚	226	E2	11100010
£	163	A3	10100011	„	227	E3	11100011
§	164	A4	10100100	‰	228	E4	11100100
•	165	A5	10100101	Â	229	E5	11100101
¶	166	A6	10100110	Ê	230	E6	11100110
ß	167	A7	10100111	Á	231	E7	11100111
®	168	A8	10101000	Ë	232	E8	11101000
©	169	A9	10101001	È	233	E9	11101001
™	170	AA	10101010	Í	234	EA	11101010
´	171	AB	10101011	Î	235	EB	11101011
¨	172	AC	10101100	Ï	236	EC	11101100
≠	173	AD	10101101	Ì	237	ED	11101101
Æ	174	AE	10101110	Ó	238	EE	11101110
Ø	175	AF	10101111	Ô	239	EF	11101111
∞	176	B0	10110000		240	F0	11110000
±	177	B1	10110001	Ò	241	F1	11110001
≤	178	B2	10110010	Ú	242	F2	11110010
≥	179	B3	10110011	Û	243	F3	11110011
¥	180	B4	10110100	Ù	244	F4	11110100
µ	181	B5	10110101	ı	245	F5	11110101
∂	182	B6	10110110	ˆ	246	F6	11110110
∑	183	B7	10110111	˜	247	F7	11110111
∏	184	B8	10111000	¯	248	F8	11111000
π	185	B9	10111001	˘	249	F9	11111001
∫	186	BA	10111010	˙	250	FA	11111010
ª	187	BB	10111011	˚	251	FB	11111011
º	188	BC	10111100		252	FC	11111100
Ω	189	BD	10111101	˝	253	FD	11111101
æ	190	BE	10111110		254	FE	11111110
ø	191	BF	10111111	ˇ	255	FF	11111111

Appendix F

ProDOS 8, ProDOS 16 (1.0), and ProDOS 16 (2.0)

[**Writer's note:** this appendix will appear only in the drafts of this manual that describe version 1.0 of ProDOS 16. The final draft, which is expected to be describing version 2.0, will refer to version 1.0 only in Appendix B.]

ProDOS 8 is the operating system developed specifically for the Apple II family of computers. Most users of the 64K Apple II Plus, Apple IIc and Apple IIe computers have ProDOS 8 as an operating system. ProDOS 8 makes use of the capabilities of the 6502 and 65C02 microprocessors used in those machines and, unlike its predecessor DOS 3.3, supports interrupts and can access several different types of disk storage devices.

The Cortland is the latest computer in the Apple II family. It has a 65816 microprocessor, compatible with—but much more powerful than—the 6502 processors used on earlier Apple II computers. To exploit this greater power, Apple is developing an operating system that is more powerful than, but similar to, ProDOS 8. That operating system is ProDOS 16 (version 2.0). ProDOS 16 (2.0) will be the principal Cortland operating system when it is complete.

However, ProDOS 16 (2.0) will probably not be finished before the first shipments of the Cortland, and the Cortland needs a functioning operating system in the interim. Several key components of Cortland system software rely on ProDOS 16 (2.0)'s capabilities. Apple has therefore introduced ProDOS 16 (version 1.0) as a temporary operating system, to allow the Cortland to function as designed and to give developers and early users a system that "feels" and acts very much like ProDOS 16 (2.0). ProDOS 16 (1.0) implements the exact call structure and many of the other features of ProDOS 16 (2.0), but will be available approximately 6 months sooner. Developers can write software compatible with ProDOS 16 (2.0) and test it under ProDOS 16 (1.0).

How does ProDOS 16 (1.0) work?

ProDOS 16 (1.0) functions as a shell around ProDOS 8. It has a user-interface layer that mimics the external appearance and system call structure of ProDOS 16 (2.0), but the actual operating system beneath the shell is ProDOS 8. Users, applications, and system software can therefore make calls as if they were calling ProDOS 16 (2.0), even though their calls are carried out by ProDOS 8.

In spite of the new system call structure, ProDOS 16 (1.0) is for the most part restricted to functions that are available under ProDOS 8. Those parts of ProDOS 16 (2.0) that will involve fundamental revisions or extensions to ProDOS 8 are not available in ProDOS 16 (1.0). Nevertheless, there are new ProDOS 16 (1.0) features and calls; they are mentioned in the next subsection and documented where appropriate in the following chapters.

What new features does ProDOS 16 (1.0) have?

ProDOS 16 (2.0) was designed to take advantage of certain Cortland capabilities and to provide additional programming convenience over ProDOS 8. ProDOS 16 (1.0) implements most of those design features. For example:

- You can make ProDOS 16 (1.0) system calls from anywhere in memory, using parameter blocks located anywhere in memory. By comparison, ProDOS 8 calls and lists must be in the lowest 64K bytes of memory
- You can make I/O data transfers under ProDOS 16 (1.0) to or from anywhere in memory. ProDOS 8 can perform I/O only with the lowest 64K bytes of memory.
- ProDOS 16 (1.0) allows limited use of named devices. With ProDOS 8 you must refer to a device by its volume name or its slot and drive numbers.
- ProDOS 16 (1.0) supports up to eight pathname prefixes; ProDOS 8 supports only one.

The following operating system calls, not recognized by ProDOS 8, are part of ProDOS 16 (1.0):

`CLEAR_BACKUP_BIT`	(clears one of a file's access bits)
`CHANGE_PATH`	(changes the pathname of a file within a volume)
`SET_LEVEL`	(sets the system file level)
`GET_LEVEL`	(returns the system file level)
`GET_DEV_NUM`	(returns the reference number for a named device)
`GET_PATHNAME`	(returns the pathname of the current application)
`GET_BOOT_VOL`	(returns the name of the volume that contains ProDOS 16)
`GET_VERSION`	(returns the current ProDOS 16 version)

These and all other ProDOS 16 calls are described in detail in Chapters 9 through 13.

What additional features will ProDOS 16 (2.0) have?

Because ProDOS 16 (1.0) functions with a core of ProDOS 8, certain features that will be part of ProDOS 16 (2.0) could not be included in ProDOS 16 (1.0). Some of them are the following:

- ProDOS 8 allows a maximum of 8 files to be open simultaneously. ProDOS 16 (1.0) has the same restriction, but ProDOS 16 (2.0) will allow an unlimited number of open files.
- ProDOS 8 allows a maximum of 14 devices on line at a time. ProDOS 16 (1.0) has the same restriction, but ProDOS 16 (2.0) will allow any number of online devices.
- ProDOS 8 supports only one block device I/O protocol. ProDOS 16 (1.0) supports the same protocol, but ProDOS 16 (2.0) will support at least three separate deviceI/O protocols.
- ProDOS 8 does not support named devices. ProDOS 16 (1.0) supports named devices only in the `VOLUME` and `GET_DEV_NUM` calls. ProDOS 16 (2.0) will have more extensive support for named devices.
- Neither ProDOS 8 nor ProDOS 16 (1.0) prompts the user to mount a needed volume. ProDOS 16 (2.0) will have a volume mount function.

The following operating system calls, not recognized by ProDOS 16 (1.0), will be part of ProDOS 16 (2.0):

`GET_ENTRY`	(returns an ASCII string with a file's directory information)
`WRITE_PROTECT`	(determines the write-protect status of a volume)
`GET_DIB`	(returns a device information block)

Glossary

absolute code: Program code that must be loaded at a specific address in memory, and never moved.

access byte: An attribute of a ProDOS 16 file that determines what types of operations, such as reading or writing, may be performed on the file.

accumulator: The register in the microprocessor where most computations are performed.

address: A number that specifies the location of a single **byte** of memory. Addresses can be given as decimal integers or as hexadecimal integers. The Cortland has addresses ranging from 0 to 16,777,215 (in decimal) or from $00 00 00 to $FF FF FF (in hexadecimal). The letter *x* in an address stands for all possible values for that digit. For example, $D*xxx* means any or all of addresses from $D000 through $DFFF.

application program (or **application**): A program that performs a specific task useful to the computer user, such as word processing, data base management, or graphics. Compare **controlling program**. ProDOS 16 applications are **file type** $B3.

ASCII: Acronym for *American Standard Code for Information Interchange*. A code in which the numbers from 0 to 127 stand for text characters. ASCII code is used for representing text inside a computer and for transmitting text between computers or between a computer and a peripheral device.

assembler: A program that produces object files from source files written in assembly language.

backup bit: a bit in a file's **access byte** that tells backup programs whether the file has been altered since the last time it was backed up.

bank: A 64K (65,536-byte) portion of the Cortland internal memory. An individual bank is specified by the value of one of the 65816 microprocessor's bank registers.

bank-switched memory: On Apple II-series computers, that part of the **Language Card** memory in which two 4K-portions of memory share the same address range ($D000–$DFFF).

binary file: A file of absolute code that is ProDOS 8 file type $06. The System Loader will not load binary files.

bit: A contraction of *binary digit*. The smallest unit of information that a computer can hold. The value of a bit (1 or 0) represents a simple two-way choice, such as yes or no or on or off.

bit map: A set of bits that represents the positions and states of a corresponding set of items. See **global page bitmap** or **volume bitmap**.

block: (1) A unit of data storage or transfer, typically 512 bytes. (2) a contiguous, page-aligned region of computer memory of arbitrary size, allocated by the Memory Manager. Also called a *memory block.*

block device: A piece of equipment (hardware) that transfers data to or from a computer in multiples of one block (512 bytes) of characters at a time. Disk drives are block devices.

boot: Another way to say **start up.** A computer boots by loading a program into memory from an external storage medium such as a disk. *Boot* is short for *bootstrap load.*

buffer: A region of memory where information can be stored by one program or device and then read at a different rate by another; for example, a ProDOS 16 I/O buffer.

byte: A unit of information consisting of 8 **bits.** A byte can take any value between 0 and 255 ($0 and $FF hexadecimal). The value can represent an instruction, number, character, or logical state.

call: (v) To request the execution of a subroutine, function, or procedure. (n) A request from the keyboard or from a program to execute a named function.

carry flag: A status bit in the microprocessor, used as an additional high-order bit with the accumulator bits in addition, subtraction, rotation, and shift operations.

character: Any symbol that has a widely understood meaning and thus can convey information. Most characters are represented in the computer as one-byte values.

character device: A piece of equipment (hardware) that transfers data to or from a computer as a stream of individual characters. Keyboards and printers are character devices.

close: to prevent further access to an open file. When a file is closed, its updated version is written to disk and all resources it needed when open (such as its I/O buffer) are released. The file must be opened before it can be accessed again.

compact: To rearrange allocated memory blocks in order to increase the amount of contiguous unallocated (free) memory. The Memory manager compacts memory when needed.

compiler: A program that produces object modules from source files written in a high-level language such as Pascal.

controlling program: A program that loads and runs other programs, without itself relinquishing control. A controlling program is responsible for shutting down its subprograms and freeing their memory space when they are finished. A shell, for example, may be a controlling program.

current application: The application program currently loaded and running. Every application program is identified by a UserID number; the current application is defined as that application whose UserID is the present value of the `USERID` global variable.

data block: a 512-byte portion of a ProDOS 16 **standard file** that consists of whatever kind of information the file may contain.

desk accessories: Small, special-purpose programs that are available to the user regardless of which application is running—such as the Control Panel, Calculator, Note Pad, and Alarm Clock

desktop: The visual interface between the computer and the user. In computers that support the desktop concept, the desktop consists of a menu bar at the top of the screen, and a gray area in which applications are opened as windows. The desktop interface was first developed for the Macintosh computer.

device: A piece of equipment (hardware) used in conjunction with a computer and under the computer's control. Also called a **peripheral device** because such equipment is often physically separate from but attached to the computer.

device driver: A program that manages the transfer of information between a computer and a peripheral device.

direct page: A page-aligned portion of bank $00 of Cortland memory, any part of which can be addressed efficiently because its high address byte is $00 and its middle address byte is fixed by the value of the 65816 processor's **direct register**.

directory file: One of the two principal categories of ProDOS 16 files. Directory files contain specifically formatted entries that contain the names and disk locations of other files. Compare **standard file**. Directory files are either **volume directories** or **subdirectories**.

disk operating system: an operating system whose principal function is to manage files and communication with one or more disk drives. **DOS** and **ProDOS** are two families of Apple II disk operating systems.

dispose: To permanently deallocate a memory block. The Memory Manager disposes of a memory block by removing its master pointer. Any handle to that pointer will then be invalid. Compare **purge**.

DOS: An Apple II disk operating system. *DOS* is an acronym for *Disk Operating System*.

dynamic segment: A segment that can be loaded and unloaded during execution as needed. Compare **static segment**.

e flag: A flag bit in the 65816 that determines whether the processor is in native mode or emulation mode.

emulation mode: The 8-bit state of the 65816 processor, in which it functions like a 6502 processor in all respects except clock speed.

EOF (end-of-file): The logical size of a ProDOS 16 file; it is the number of bytes that may be read from or written to the file.

error (or **error condition**): the state of a computer after it has detected a fault in one or more commands sent to it.

error code: a number or other symbol representing a type of error.

event: a notification to an application of some occurrance (such as an interrupt generated by a keypress) that the application may want to respond to.

event-driven: A kind of program that responds to user inputs in real time by repeatedly testing for events posted by interrupt routines. An event-driven program does nothing until it detects an event such as a keypress.

external device: See **device.**

fatal error: an error serious enough that the computer must halt execution.

file control block (FCB): a data structure set up in memory by ProDOS 16 to keep track of all open files.

file entry or **file directory entry:** The part of a ProDOS 16 directory or subdirectory that describes and points to another file. The file so described is considered to be "in" or "under" that directory.

file level: see **system file level**

filename: The string of characters that identifies a particular file within its directory. ProDOS 16 filenames may be up to 15 characters long. Compare **pathname.**

file type: An attribute in a ProDOS 16 file's directory entry that characterizes the contents of the file and indicates how the file may be used. On disk, file types are stored as numbers; in a directory listing, they are often displayed as three-character mnemonic codes.

filing calls: Operating system calls that manipulate files. In ProDOS 16, filing calls are subdivided into *file housekeeping calls* (described in Chapter 9) and *file access calls* (described in Chapter 10).

finder: A program that performs file and disk utilities (formatting, copying, renaming, and so on) and also starts applications at the request of the user.

firmware: Programs stored permanently in the computer's read-only memory (ROM). They can be executed at any time but cannot be modified or erased.

flush: to update an open file (write any updated information to disk) without closing it.

global page: Under ProDOS 8, 256 bytes of data at a fixed location in memory, containing useful system information (such as a list of active devices) available to any application.

global page bitmap: A portion of the ProDOS 8 global page that keeps track of memory use in the computer. Applications under ProDOS 8 are responsible for marking and clearing parts of the bitmap that correspond to memory they have allocated or freed.

guest file system: a file system, other than ProDOS 16's, whose files can be read by ProDOS 16.

handle: See **memory handle**

hexadecimal: The base-16 system of numbers, using the ten digits 0 through 9 and the six letters A through F. Hexadecimal numbers can be converted easily and directly to binary form. In Apple manuals hexadecimal numbers are usually preceded by a dollar sign ($).

high-order: The most significant part of a numerical quantity. In normal representation, the *high-order bit* of a binary value is in the leftmost position; likewise, the *high-order byte* of a binary **word** or **long word** quantity consists of the leftmost eight bits.

Human Interface Guidelines: A set of software development guidelines developed by Apple Computer to support the **desktop** concept and to promote uniform user interfaces in Apple II and Macintosh applications.

image: A representation of the contents of memory. A code image consists of machine-language instructions or data that may be loaded unchanged into memory.

index block: A 512-byte part of a ProDOS 16 **standard file** that consists entirely of pointers to other parts (**data blocks**) of the file.

initial load file: The first file of a program to be loaded into memory. It contains the program's main segment and the load file tables (Jump Table segment and Pathname segment) needed to load dynamic segments and run-time libraries.

initialization segment: A segment in an initial load file that is loaded and executed first, to perform any initialization that the program may require.

input/output: the transferral of information between a computer's memory and peripheral **devices.**

interrupt: a temporary suspension in the execution of a program that allows the computer to perform some other task, typically in response to a signal from a device or source external to the computer.

interrupt handler: a program, associated with a particular external device, that executes whenever that device sends an interrupt signal to the computer. The interrupt handler performs its tasks during the interrupt, then returns control to the computer so it may resume program execution.

interrupt vector table: A table maintained in memory by ProDOS 16 that contains the addresses of all currently active (allocated) interrupt handlers.

INTERSEG record: A part of a relocation dictionary. It contains relocation information for external (intersegment) references.

I/O: See **input/output.**

Jump Table: A table contructed in memory by the System Loader from all Jump Table segments encountered during a load. The Jump Table contains all references to dynamic segments that may be called during execution of the program.

Jump Table directory: a master list in memory, containing pointers to all segments that make up the Jump Table.

Jump Table segment: A segment in a load file that contains all references to dynamic segments that may be called during execution of that load file. The Jump Table segment is created by the linker.

K: Kilobyte. 1024 (2^{10}) bytes.

kernel: The central part of an operating system. ProDOS 16 is the kernel of the Cortland operating system.

key block: The first block in any ProDOS 16 file.

kind: see **segment kind.**

Language Card: Originally, a peripheral card for the Apple II that expanded its memory capacity from 48K to 64K. It is now a general term, denoting those parts of memory with addresses between $D000 and $FFFF on any Apple II-family computer. See also **bank-switched memory.**

level: see **system file level**

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.

linker: The 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.

load file: The output of the linker. Load files contain memory images that the System Loader can load into memory.

load segment: A segment in a load file.

lock: To prevent a memory block from being moved or purged. A block may be locked or unlocked by the Memory Manager, or by an application through a call to the System Loader.

long word: A double-length word. For the Cortland, a long word is 32 bits (4 bytes) long.

low-order: The least significant part of a numerical quantity. In normal representation, the *low-order bit* of a binary number is in the rightmost position; likewise, the *low-order byte* of a binary **word** or **long word** quantity consists of the rightmost eight bits.

m flag: A flag in the 65816 processor that determines whether the accumulator is 8 bits wide or 16 bits wide.

macro: a single predefined assembly-language pseudo-instruction that an assembler replaces with several actual instructions. Macros are almost like higher-level instructions that can be used inside assembly-language programs, making them easier to write.

main segment: The first segment in the initial load file of a program (unless the file also has an initialization segment). It is loaded first and never removed from memory until the program terminates.

MARK: The current position in an open file. It is the point in the file at which the next read or write operation will occur.

master index block: The key block in a ProDOS 16 **tree file**, the largest organization of a **standard file** that ProDOS 16 can support. The master index block consists solely of pointers to one or more **index blocks**.

master pointer: A pointer to a memory block; it is kept by the Memory Manager. Each allocated memory block has a master pointer, but the block is normally accessed through its memory handle (which points to the master pointer), rather than through the master pointer itself.

Mb: Megabyte. 1,048,576 (2^{20}) bytes.

memory handle: The identifying number of a particular block of memory. It is a pointer to the master pointer to the memory block. A handle rather than a simple pointer is needed to reference a movable memory block; that way the handle will always be the same though the value of the pointer may change as the block is moved around.

Memory Manager: A program that manages memory use in the Cortland. The Memory Manager allocates and deallocates memory **blocks** to hold program segments or data.

memory block: see **block** (2).

MLI: Machine Language Interface—the part of ProDOS 8 that processes operating system calls.

monitor: see **video monitor**.

monitor program: a firmware program of Apple II-family computers, used for operating the computer at the machine-language level.

move: To change the location of a memory block. The Memory Manager may move blocks to consolidate memory space.

movable: A memory block attribute, indicating that the Memory Manager is free to move the block . A block is made movable or unmovable through Memory Manager calls.

native mode: the 16-bit operating state of the 65816 processor.

newline: a file-reading mode in which each character read from the file is compared to a specified character (called the **newline character**); if there is a match, the read is terminated. Newline mode is typically used to read individual lines of text, with the newline character defined as a carriage return.

nibble: a unit of information consisting of one-half of a **byte**, or 4 **bits**. A nibble can take on any value between 0 and 15 ($0 and $F hexadecimal).

NIL: Pointing to a value of 0. A memory handle is NIL if the address it points to is filled with zeroes. Handles to purged memory blocks are NIL.

object file: The output from an assembler or compiler, and the input to a linker.

object module: An object file in Object Module Format.

object module format: The general format used in object files, library files, and load files.

OMF File: Any file in object module format.

open: To allow access to a file. A file may not be read from or written to until is is open.

operating system: A program that organizes the actions of the various parts of the computer and its peripheral devices. See also **disk operating system.**

page: A portion of memory 256 bytes long and beginning at an address that is an even multiple of 256. Memory blocks whose starting addresses are an even multiple of 256 are said to be *page-aligned.*

parameter: A value passed to or from a function.

parameter block: A set of contiguous memory locations, set up by a calling program to pass parameters to and receive results from an operating system function that it calls. Every call to ProDOS 16 must include a pointer to a properly constructed parameter block.

partial pathname: the part of a pathname following a **prefix.** A partial pathname does not begin with a slash because it has no volume directory name.

pathname: the complete name by which a file is specified. It is a sequence of **filenames** separated by slashes, starting with the filename of the volume directory and following the path through any subdirectories that the operating system must follow to locate the file. A complete pathname always begins with a slash, because volume directory names always begin with a slash.

Pathname segment: A segment in a load file that contains the cross-references between load files referenced by number (in the Jump Table segment) and their pathnames (listed in the file directory). The Pathname segment is created by the linker.

Pathname Table: A table constructed in memory from all individual Pathname segments encountered during a load. It contains the cross-references between load files referenced by number (in the Jump Table) and their pathnames (listed in the file directory).

pointer: An item of information consisting of the memory address of some other item.

position-independent code: Code that is written specifically so that its execution is unaffected by its position in memory. It can be moved without needing to be relocated.

prefix: A portion of a pathname starting with a volume name and ending with a subdirectory name. A prefix and a **partial pathname** together constitute a complete **pathname.**

prefix designator: A code used to represent a particular prefix. Under ProDOS 16, there are eight prefix designators, each consisting of a number followed by a slash: 0/, 1/,...,8/.

ProDOS: A family of disk operating systems developed for the Apple II family of computers. *ProDOS* stands for *Professional Disk Operating System*, and includes both ProDOS 8 and ProDOS 16. All ProDOS operating systems have identical file formats.

ProDOS 8: A disk operating system developed for the 64K Apple II Plus, Apple IIe, and Apple IIc computers. It functions in 8-bit mode, compatible with 6502-series microprocessors. It also runs on the Cortland when the 65816 processor is in 6502 **emulation mode.**

ProDOS 16: A disk operating system, functionally similar to ProDOS 8 but more powerful, developed for 65816 **native mode** operation on the Cortland.

purge: To temporarily deallocate a memory block. The memory menager purges a block by setting its master pointer to NIL (0). All handles to the pointer are still valid, so the block can be reconstructed quickly. Compare **dispose.**

purge level: An attribute of a memory block that sets its priority for purging. A purge level of 0 means that the block is unpurgeable.

purgeable: A memory block attribute, indicating that the Memory Manager may purge the block if it needs additional memory space. Purgeable blocks have different **purge levels**, or priorities for purging; these levels are set by Memory Manager calls.

queue: A list in which entries are added at one end and removed at the other, causing entries to be removed in first-in, first-out (FIFO) order. Compare **stack.**

quit return stack: A stack maintained in memory by ProDOS 16. It contains a list of programs that have terminated but wish to return when the presently executing program is finished.

random-access device: See **block device.**

reentrant: A characteristic of certain types of software. A program is reentrant if it reinitializes its variables and makes no assumptions about machine state each time it gains control. Programs that are reentrant can be restarted from a **dormant** state in memory; they also can be executed even while they have been halted during operation by an **interrupt** signal. Under certain conditions, the **Scheduler** manages execution of programs that are not reentrant.

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.

relocatable segment: A load segment that can be loaded into any part of memory. A relocatable load segment contains a code **image** followed by a **relocation dictionary**.

relocation dictionary: A part of a relocatable load segment that contains relocation information necessary to modify the code-image immediately preceding it. When the code-image part of the segment is loaded into memory, the relocation dictionary is processed to recalculate the values of location-dependent addresses and operands.

restart: to reactivate a **dormant** program in the computer's memory. The System Loader can restart dormant programs if all their segments are still in memory. If any critical part of a dormant program has been purged by the Memory Manager, the program must be reloaded from disk instead of restarted.

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.

sapling file: An organizational form of a ProDOS 16 **standard file.** A sapling file consists of a single **index block** and up to 256 **data blocks.**

Scheduler: a firmware program that manages requests to execute **interrupted** software that is not **reentrant.** If, for example, an interrupt handler wishes to make ProDOS 16 calls, it must do so through the Scheduler because ProDOS 16 is not reentrant. Applications need not use the Scheduler because ProDOS 16 is not in an interrupted state when it processes applications' system calls.

sector: a division of a **track** on a disk. When a disk is formatted, its surface is divided into tracks and sectors.

seedling file: An organizational form of a ProDOS 16 **standard file.** A seedling file consists of a single **data block.**

segment: An individual component of an OMF file. Each file contains one or more segments.

segment kind: a numerical designation used to classify a segment in object module format. It is the value of the `KIND` field in the segment's header.

sequential-access device: See **character device.**

source file: An ASCII file consisting of instructions written in a particular language, such as Pascal or assembly language. An assembler or compiler converts source files into object files.

sparse file: A variation of the organizational forms of ProDOS 16 **standard files.** A sparse file may be either a **sapling file** or a **tree file**; what makes it sparse is the fact that its logical size (defined by its **EOF**) is greater than its actual size on disk. This occurs when one or more **data blocks** contain nothing but zeros. Those data blocks are considered to be part of the file, but they are not actually allocated on disk until nonzero data is written to them.

stack: A list in which entries are added (pushed) and removed (pulled) at one end only (the top of the stack), causing them to be removed in last-in, first-out (LIFO) order. Compare **queue.**

standard file: One of the two principal categories of ProDOS 16 files. Standard files contain whatever data they were created to hold; they have no predefined internal format. Compare **directory file.**

start up: To get the system running.. It involves loading system software from disk, and then loading and running an application. Also called **boot.**

static segment: A segment that is loaded only at program boot time, and is not unloaded during execution. Compare **dynamic segment**.

storage type: An attribute of a ProDOS 16 file that describes the file's organizational form (such as directory file, seedling file, or sapling file).

switcher: a controlling program that rapidly transfers execution among several applications.

System Death Manager: A firmware program that processes fatal errors by displaying a message on the screen and halting execution.

system: A coordinated collection of interrelated and interacting parts organized to perform some function or achieve some purpose—for example, a computer system comprising a processor, keyboard, monitor, disk drive, and software.

system file level: A number between $00 and $FF associated with each open ProDOS 16 file. Every time a file is opened, the current value of the system file level is assigned to it. If the system file level is changed (by a `SET_LEVEL` call), all subsequently opened files will have the new level assigned to them. By manipulating the system file level, a controlling program can easily close or flush files opened by its subprograms.

System Loader: The program that manages the loading of program and data segments into the Cortland memory. The System Loader works closely with ProDOS 16 and the Memory Manager.

system program: A self-booting **application**. It is either 1) a ProDOS 16 application (file type $B3), whose filename has the extension .SYS16; or 2) a ProDOS 8 application (file type $FF), whose filename has the extension .SYSTEM.

system software: The components of a computer system that support application programs by managing system resources such as memory and I/O devices.

subdirectory: A ProDOS 16 directory file that is not the volume directory.

tool: see **Tool Set**

Tool Set: a related group of (usually firmware) routines, available to applications and system software, that perform necessary functions or provide programming convenience. The Memory Manager, the System Loader, and Quickdraw II are Tool Sets.

toolbox: The sum of all Tool Sets available on the Cortland. It includes both ROM-based and RAM-based Tool Sets.

track: A series of concentric circles on a disk. When a disk is formatted, its surface is divided into tracks and **sectors**.

tree file: An organizational form of a ProDOS 16 **standard file**. A tree file consists of a single **master index block**, up to 127 **index blocks**, and up to 32,512 **data blocks**.

unload: To remove a load segment from memory. To unload a segment, the System Loader does not actually "unload" anything; it calls the Memory Manager to either **purge** or **dispose** of the memory block in which the code segment resides. The loader then modifies the Memory Segment Table to reflect the fact that the segment is no longer in memory.

UserID: an identification number that specifies the owner of every memory block allocated by the Memory Manager. UserID's are assigned by the UserID Manager.

UserID Manager: A Tool Set that is responsible for assigning UserID's to every block of memory allocated by the Memory Manager.

video monitor: a display device that receives video signals by direct connection only.

version: A number indicating the release edition of a particular piece of software. Version numbers for most system software (such as ProDOS 16 and the System Loader) are available through function calls.

volume: An object that stores data; the source or destination of information. A volume has a name and a volume directory with the same name. Volumes typically inhabit **devices;** a device such as a floppy disk drive may contain one of any number of volumes (disks).

volume bitmap: A portion of every ProDOS 16-formatted disk that keeps track of free disk space.

volume control block (VCB): A data structure set up in memory by ProDOS 16 to keep track of all volumes/devices connected to the computer.

volume directory: A ProDOS 16 directory file that is the principal directory of a volume. It has the same name as the volume. The pathname of every file on the volume starts with the volume directory name.

word: A group of bits that is treated as a unit. For the Cortland, a word is 16 bits (2 bytes) long.

zero page: The first page (256 bytes) of memory in Apple II computers. Since the high-order byte of any address in this part of memory is zero, only a single byte is needed to specify a zero-page address. Zero-page locations are therefore more quickly and compactly addressed than any other parts of memory. Compare **direct address.**