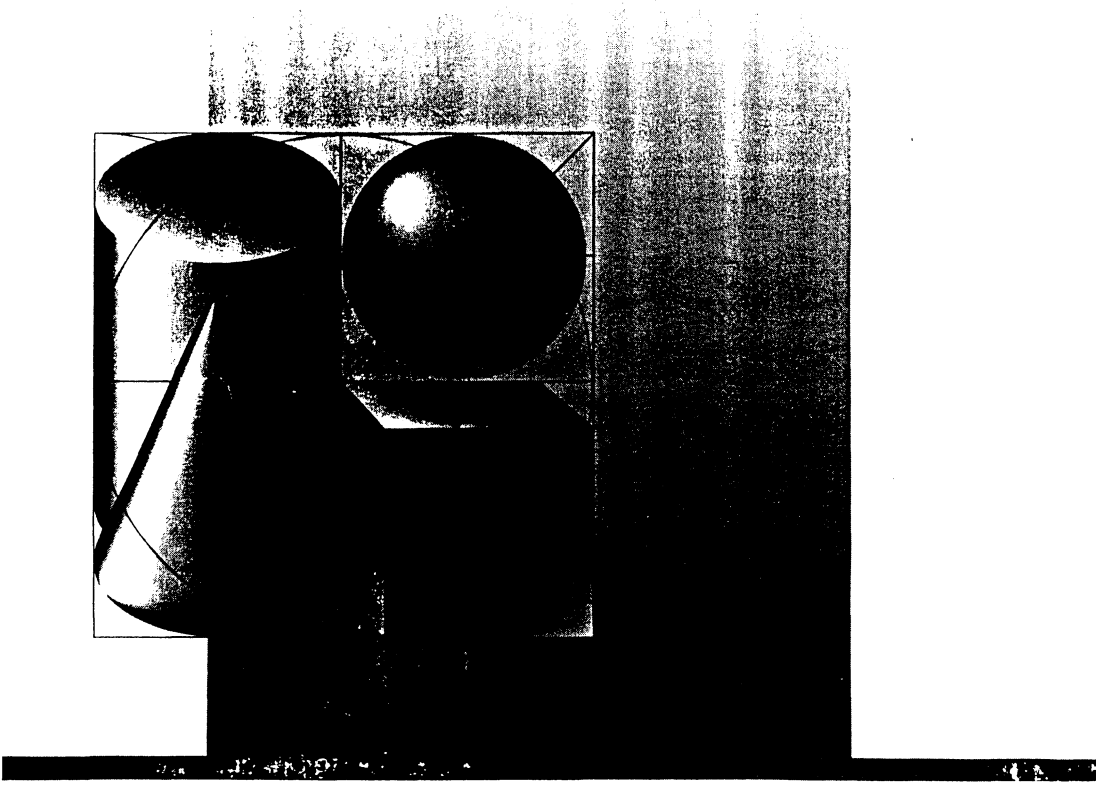
AppleShare® Programmer's Guide for the Apple® II

Beta Draft

APDA™ # A2G0051/A



Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010
TLX 171-576

To reorder products, please call:
Apple Programmers and Developers Association
1-800-282-APDA

AppleShare® Programmer's Guide for the Apple II

NOTICE

The information in this document reflects the current state of the product. Every effort has been made to verify the accuracy of this information; however, it is subject to change. Preliminary Notes are released in this form to provide the development community with essential information in order to work on compatible products.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014-6299
(408) 996-1010

Simultaneously published in the United States and Canada.

Contents

Figures and Tables

1 Application Development

2 Programming Guidelines

3 Calls to AppleTalk Protocols

4 The AppleShare File System Translator (FST)

Appendix A Result Codes

Preface

THIS PROGRAMMERS GUIDE is written for application program developers for the Apple® IIGS® and Apple IIe® computer who want to do either of the following:

- develop new network-specific applications for the Apple IIGS and Apple IIe computer
- modify existing application programs to implement AppleTalk® protocols on the Apple II workstations.

You can develop either ProDOS 8- or GS/OS-based applications for use with the AppleTalk network system. ProDOS 8 applications have the advantage of working with either of the Apple II workstations (IIe or IIGS), while GS/OS applications will be able to use the more advanced features of the Apple IIGS workstation. An Apple II workstation is an Apple IIe or Apple IIGS with a workstation card installed.

You Should Know...

You should have a working knowledge of the Apple II, the operating systems developed specifically for the Apple II family of computers, ProDOS 8 and GS/OS. Only ProDOS 8 (version 1.5 or later) and GS/OS are supported. This note also assumes you are familiar with AppleTalk protocols (described in *Inside AppleTalk*) and the LocalTalk™ cable system.

Application Compatibility

The Apple II workstation is intended to support applications that are compatible with ProDOS, written in the following development environments:

- Assembler
- BASIC
- Pascal (ProDOS only)
- C (ProDOS only)

◆ *Note:* If you have an existing program, that is neither ProDOS 8 based nor OS/OS based, you should first convert it to ProDOS 8 or GS/OS to enable it to run on the network. DOS 3.3 RWTS calls, related hardware accesses, or both must be eliminated from all programs. Only standard entry points will be supported.

Most Apple II applications have little need to know that they may be in a network environment, and most ProDOS 8 and GS/OS, and their operating system calls that can be made to local volumes can also be made to network server volumes. However, there are situations in which current programming practices will not function properly in a network environment. Chapter 2 provides programming guidelines and examples of network situations to consider in developing your application.

◆ *Note:* Apple IIGS System Software 4.0 (the first release of GS/OS) does not support file service, but does support all other network layers (i.e. LAP through ASP). System Software 5.0 supersedes the AppleShare IIGS workstation software.

Where to Go for More Information

The following is a list of Apple Computer reference materials for Apple IIGS computers and the AppleTalk network system that you might find helpful. These documents are related to the Apple II computer:

- *Apple IIGS Firmware Reference Manual* describes routines stored in ROM (except for BASIC and the Toolbox). The manual includes interrupt routines and low-level I/O for serial ports, disk, and DeskTop Bus.
- *BASIC Programming with ProDOS* covers file and program conversion from DOS 3.3 to ProDOS.
- GS/OS Reference Manual, Volumes 1 and 2 (Volume 1 is Calls, Volume 2 is Device Drivers) describes the new operating system.
- *ProDOS Technical Reference Manual* describes the ProDOS operating system for assembly language programmers.
- *Programmer's Introduction to the Apple IIGS* introduces programming for the Apple IIGS for the desktop environment using the IIGS tools.
- *Technical Introduction to the Apple IIGS* describes the IIGS and its differences from the IIe, the programming environments, the toolbox, and an introduction for hardware designers and Apple Programmer's Workshop (APW) users.
- *Apple IIe Technical Reference Manual.*

These documents are related to AppleTalk:

- *Inside AppleTalk* describes the theory of the AppleTalk network, including specifications for the AppleTalk protocols.
- *LocalTalk Cable System Owner's Guide* shows you how to set up a network, and how to add to and change the network once you've set it up, using LocalTalk cables and connectors for the AppleTalk network system.
- *Software Applications in a Shared Environment* (Preliminary Note) describes programming guidelines for developing applications to function properly in a shared environment (such as network file servers). Although this note was developed for applications using AppleShare™ on the Macintosh® computer, some of the information is relevant to the Apple IIGS.
- *AppleTalk Network User's Guide for the Apple IIGS* Included with System 5.0.

Most of these items are available through. APDA is an excellent source of technical information for anyone interested in developing Apple-compatible products. Membership in the association allows you to purchase Apple technical documentation, programming tools, and utilities. For information on membership fees, available products, and prices, please contact

APDA
Apple Computer, Inc.
20525 Mariani Avenue, Mailstop 33-G
Cupertino, CA 95014-6299
(800) 282-APDA (800-282-2732)
Fax: 408-562-3971
Telex: 171-576
AppleLink: APDA

Chapter 1 **Application Development**

THIS CHAPTER provides an overview of the AppleTalk network system and describes the AppleTalk functions implemented on the Apple II workstation. Refer to Chapter 2 for programming guidelines, call formats, and protocol parameter structures, and Chapter 3 for a detailed description of calls to each of the AppleTalk protocols. ■

The AppleTalk network system

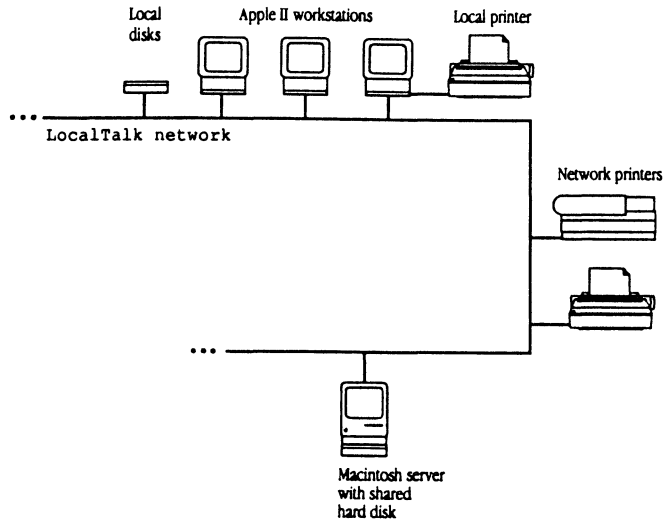
As the number of products from Apple Computer, Inc., continues to grow, so does the desire for a simple way to take advantage of the economies and features each product has to offer. The solution is the AppleTalk network system, in which Apple II workstations and a Macintosh server can share file and printer resources, thus avoiding costly hardware duplication.

An AppleTalk network system has three components: a cable system that links devices; software that supports the network; and optional services that networked devices share, such as LaserWriter printers and AppleShare file servers. AppleTalk has its own protocol architecture that can be run over different physical media. Currently, the LocalTalk Cable System is the only physical medium supported in the Apple II family of computers.

Figure 1-1 shows how Apple II workstations and a Macintosh server are connected to form the network. Hard disks can be attached directly to the server, allowing workstations to share applications and data. Other peripherals, such as a LaserWriter® or ImageWriter® II with an LocalTalk option card, are attached directly to the network. Local printers and other devices can also be connected to individual workstations for use only by that workstation.

The LocalTalk cable system uses a bus topology. The bus acts as the common medium routed through all devices attached to the network. The advantages of the bus are that network devices are attached at any convenient point, and failure of one network device does not cause the entire network to fail. Each device (such as a workstation or printer) connected to the network obtains its own unique address when the device powers up and makes a logical connection to the network.

■ **Figure 1-1** The AppleTalk Network System with Apple II Workstations



AppleTalk on the Apple II workstation

The Apple II workstation is designed to provide similar AppleTalk functionality to perform network functions as is built into the Macintosh computer. This design allows the Apple II workstation to

- Starting up over the network (network booting)
- access servers on the network
- print to a LaserWriter or ImageWriter II on an AppleTalk network system

Applications run on Apple II workstations can use their own high-level protocols interacting with low-level AppleTalk protocols to communicate with any network device. An application can make several types of calls: printing calls, calls for booting over the network, filing calls for file service on the network, AppleTalk network calls, and diagnostic or "housekeeping" calls.

The Apple II workstation processes the calls and data through the operating system, translating operating system filing calls to the appropriate AppleTalk protocol calls. When the translation is complete, the firmware on the workstation formats the data into packets and prepares to transmit it over the network to the server.

The Apple II workstation currently implements the following AppleTalk protocols in ROM or RAM:

- Link Access Protocol (LAP)
- Datagram Delivery Protocol (DDP)
- Name Binding Protocol (NBP)
- Zone Information Protocol (ZIP)
- AppleTalk Transaction Protocol (ATP)
- Printer Access Protocol (PAP)
- AppleTalk Session Protocol (ASP)
- Routing Table Maintenance Protocol (RTMP)
- Echo Protocol (EP)

Chapter 3 provides detailed information on making calls to each of these protocols. In addition to the AppleTalk layers just described, the Apple II workstation supports the following:

- the Remote Printer Manager (RPM) for transparent printer access, which emulates Super Serial Card (SSC) serial drivers for the serial port
- special calls to provide a timer interrupt
- code to allow an Apple II workstation to boot directly over the network from any server that supports AppleTalk Filing Protocol (AFP), version 2.0
- ProDOS Filing Interface (PFI), which allows transparent ProDOS file access by translating ProDOS filing calls which have AFP servers as destinations into AFP calls

- *Note:* Features of AFP that are not available through operating system calls can be accessed by making AFP calls directly through ASP.

- AppleShare FST for making calls from GS/OS on the Apple IIGS workstation

  - *Note:* Making AFP calls directly through ASP is discouraged under GS/OS. Most AFP calls are available through normal GS/OS calls, or FST-Specific calls to the AppleShare FST. If these calls do not provide the functionality you need, please contact Apple II Developer Technical Support for more information.

Requirements

In order to boot GS/OS over the network, all servers in a zone should be updated with the GS/OS booting software. As with GS/OS in general, all machines that wish to boot GS/OS over the network must have version 01 ROMs or newer.

Downloading the code

When AppleTalk is activated and the startup is set to AppleTalk, the AppleTalk ROMs will start the boot process. First it will look for an entity with type "Apple //gs" ; the first object that responds will be used. It issues an ATP request with the machine type (1) in the first user byte and the block number of the image in the second and third user bytes (low order first, starting with block 0). Blocks are 512 bytes each and placed in memory starting at $800 in bank 0. If the first ATP user byte in the response from the server is non-zero then that is the last block in the image and it may be shorter than 512 bytes.

Because the retry counts and intervals in ROM prior to version 3 are too short when large numbers of machines are trying to boot, this first image will be one block or less in length to try to prevent timeout errors. This block will contain code that delays (about 5 seconds) to allow other machines booting up a chance to find the server before network traffic gets too heavy.

After the delay, a lookup for the operating system's image is performed. Then the first few blocks (about 2K) of this second image is loaded by the code in the first image using ATP requests. The block number and end-of-image flag work the same; the machine ID is 3. After these few blocks have been read in, they receive control. This code is known as "Fizzy." Fizzy is responsible for displaying a user-friendly message and indicating progress while downloading the rest of the image.

Starting up the OS

The image contains patches and additions to the protocol stack through ASP, PFI, a logon program, and an FST stub. After protocol layers are installed and initialized, the logon program runs allowing the user to log on to a file server. The FST stub containing the routines ReadInFile, GetBootName, and GetFSTName is left at $2000 and the file /Volume/System/Start.GS.OS is read in and executed at $6800. Start GS/OS contains the GLoader and GQuit routines that will call the FST stub to load in the AppleShare FST from the System/FST directory on the boot volume. Once the FST is read in, the rest of the operating system will be loaded and executed.

At this time drivers, FSTs, setup files, DAs, etc. are loaded from folders in the System directory on the boot volume. Therefore, these files will be shared by all users who boot GS/OS from that server.

The AppleTalk drivers will find out from PFI which volumes were mounted and create Device Information Blocks for them so that the volumes the user selected during boot remain connected.

After the OS is loaded, it will first look for the file START in the System directory on the boot volume. The Start program will load any permanent or temporary init files and desk accessories found in the user's setup folder ("*/Users/*UserName*/Setup"), check for mail, open the user's ATINIT file, set their default printer, and launch their startup application. Note that the System/Start program will be run by GS/OS whenever an application quits and there is no program to quit to and there are no other programs waiting to be restarted (i.e. when ProDOS 16 would have displayed the "Start Next Program" menu).

GS/OS <–> ProDOS 8 switching

ProDOS 8 will be loaded from the server on demand. PFI (ProDOS Filing Interface) will have to be informed of any volumes mounted or unmounted while GS/OS was active so that ProDOS 8 will have an accurate view of the world (PFI will actually maintain session and volume information and will share this information with the AppleTalk drivers). After ProDOS 8 is loaded and initialized, PFI must be patched into the $BF00 vector.

When returning to GS/OS, the operating system restarts from RAM. When the drivers are reinitialized, they will have to find out from PFI if any volumes have been mounted or unmounted and update the Device Information Blocks appropriately.

Trivia: AppleShare is the only foreign file system that GS/OS can boot from and still support ProDOS 8.

User Interface

To boot over the network, the first thing you need to do is set up the control panel properly. If you have version 01 ROMs, you need to set slot 7 to AppleTalk, and set the startup slot to 7. If you connect the drop box to the printer port, then slot 1 should be set to "Your Card" (slot 2 can be set to either "Your Card" or "Modem Port"). If you connect the drop box to the modem port, then slot 1 should be set to "Printer Port" and slot 2 should be set to "Your Card."

If you have version 03 ROMs, set slot 1 to "AppleTalk" if the drop box is connected to the printer port, or set slot 2 to "AppleTalk" if the drop box is connected to the modem port. Set "Startup:" to "AppleTalk".

- *Note:* Some older applications (such as Aristotle) require you to set slot 7 to "AppleTalk" as well.

As you power on (or reboot), you will see some dots displayed near the upper left corner of the screen; these dots are generated by the ROM and the first stage boot code to let you know that something is happening. At this point, the first stage of the boot code is being read from a server.

Shortly, the second stage boot code (known as "Fizzy") will have been loaded, and put up the screen shown in *Figure 1-2.*

The server name is displayed near the middle of the screen. A "spinner" (a line that rotates in 45° increments) is displayed between the "Starting up over the network" message and the server name; it indicates progress during the boot process by turning 45° as each block is read in. The thermometer at the bottom of the screen is filled in proportionately to the amount of boot information read in (it will be completely filled in as the last block is read in). A typical screen about 2/3 through the boot process is shown in *Figure 1-3.*

If the connection with the server is lost (i.e. a request times out), the server name, spinner, and the insides of the thermometer will be erased (reverting back to *Figure 1-2*) and there will be a lookup for another server. If a server is found, booting will start over; otherwise the screen will be cleared and control will return to the ROM to look for another server for the first stage boot code.

Once the AppleTalk protocols have been loaded and initialized, the Logon program will be run. If you have multiple zones or multiple servers in your zone, you will see the screen in *Figure 1-4.* If there are no routers (and hence no zones), the "Current zone:..." string will not be displayed and "Change zones: Esc" will become "Cancel: Esc" . If you press ESC at this point, an attempt will be made to find a router and let you choose from a list of zones (as in *Figure 1-5*). Once you have selected a zone or if there are no zones, you will be returned to the screen in *Figure 1-4.* Pressing ESC from the zone selection screen will pick your current zone and return to *Figure 1-4.*

To select from either the zone list or the server list, you may use the up and down arrow keys to move the highlighted bar up and down through the list. If there are more items below the bottom of the window, the word "More" is displayed along the bottom line of the window. Pressing Return will select the highlighted name from the list.

Once you have selected a file server (or if there are no zones and only one file server), you will be presented with the screen shown in *Figure 1-6.* You must choose whether you will log in as a guest (" <Any User>") or as a registered user. Pressing the up and down arrows will move through the choices. Pressing Return selects the highlighted choice. Pressing ESC returns you to *Figure 1-4.*

If you selected "Log on as a Registered User" , you will be presented with the screen shown in *Figure 1-7.* You must enter your user name and password. If either is incorrect, a message will be displayed and you will be asked to try again.

Once you have successfully logged on to the server, you will be presented with a list of volumes on the server as shown in *Figure 1-8*. If there were no zones, only one server, and only one volume on the server, this list will not be displayed and the only volume will be automatically mounted. Volumes with a check mark next to them will be mounted when you press Return. Use the up and down arrow keys to move through the list of volumes; the left arrow key will remove the check mark next to the selected volume; the right arrow key will put a check mark next to the selected volume. Note that the user volume (the volume with the Users folder and the folders for all of the users) is automatically checked and cannot be unchecked. This volume will become your boot volume.

Once you have selected any additional volumes, the boot process will continue. Setup files, desk accessories, file system translators, drivers, etc. will be loaded from the user volume. Eventually, the startup application on the user volume will be run. If you have installed the network booting software normally, this will be the "*/System/Start*" file and the process will continue as described below.

The startup application will first load any custom setup files and desk accessories found in your user folder (" */Users/*_Your Name_*/Setup*"). Note that these are loaded in addition to the system-wide files loaded at boot time from the usual places in the System folder. Next, your mail folder (" */Users/*_Your Name_*/Mail*") will be checked; if it is a non-empty folder, you will be told that you have mail waiting (see *Figure 1-10*). Your default printer will be set to the printer named in your ATINIT file (as set up in AppleShare Admin). Next, prefix 0 will be set to the prefix in the ATINIT file (set up in AppleShare Admin). Lastly, the user's startup application named in the ATINIT file (set up by AppleShare Admin) will be launched.

If the user's startup program quits, control will return to the AppleShare Startup program (described in the next section).

AppleShare startup (and Quick Logoff)

The file '*/System/Start*' is the AppleShare Startup program. If the user's startup application quits, control is returned to the AppleShare Startup program and the screen shown in *Figure 1-9* will be displayed. From here, you have four options: log off from all file servers, re-run the startup application, and reboot.

Selecting "Log off from file servers" will log you off from all file servers and then you (or another user) will be allowed to log on again (starting with *Figure 1-4* or *Figure 1-6* as appropriate). After logging on, the new user's startup application will be launched. Note that the operating system is not reloaded, and no custom desk accessories or setup files are loaded for the new user. Typically, a student would select this option at the end of a class and the student in the next class can log on without having to completely reboot.

Selecting "Return to startup application" will check for mail and re-run your startup application as if you had just logged on (custom desk accessories and setup files will not be reloaded). Select this option if you accidently quit from your startup application and want to run it again.

Selecting "Shutdown" will log you off from all file servers, eject all disks, and reboot the machine. This function is similar to the Restart option from the Finder's "Shut Down" command. This option should be used when you want to completely restart the computer, such as when you have loaded custom desk accessories or setup files and you don't want to have them installed for the next user.

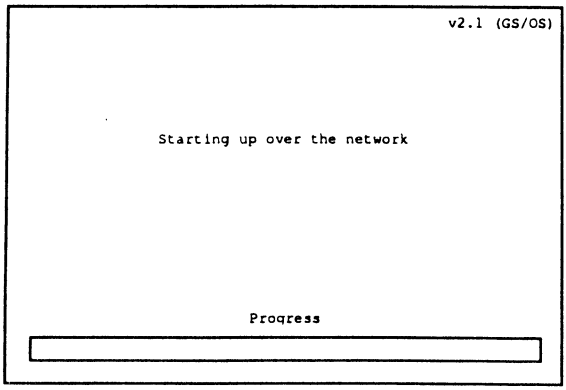
If you install the Quick Logoff update, the menu in Figure 1-9 is skipped and AppleShare acts as if the first option was selected.

The Aristotle patch

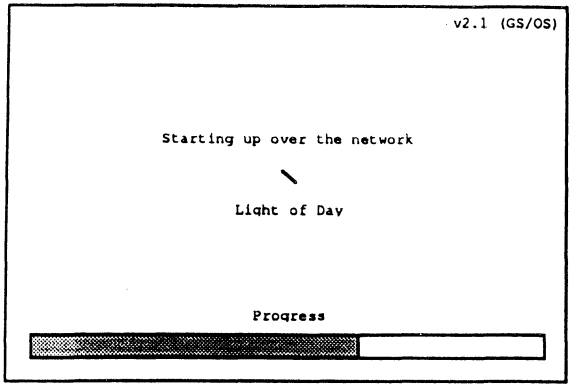
Aristotle is Apple's classroom management software for the Apple II. When a user quits from Aristotle, it reboots the machine. Originally, this was done so that students would not have to run a separate Logoff program when they were done using the machine. This also means that Aristotle, as shipped, cannot make use of the quick logoff feature.

In order to allow Aristotle to take advantage of the quick logoff feature, we have included an update that will modify Aristotle to determine which machine it is running on before trying to reboot. If it is running on an Apple //e, it will reboot as usual. On an Apple IIGS, it will instead do a ProDOS 8 QUIT call to return control to the AppleShare Startup application.

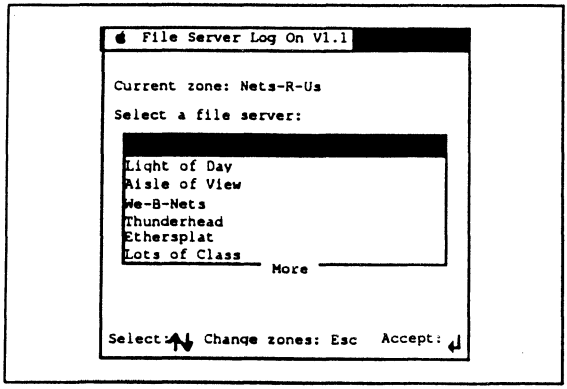
■ **Figure 1-2** Initial Startup Screen



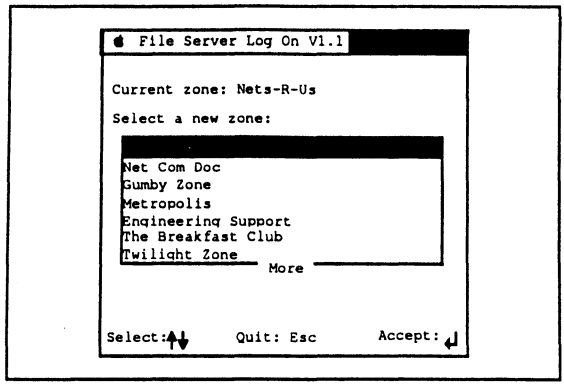
■ **Figure 1-3** Loading Startup Code



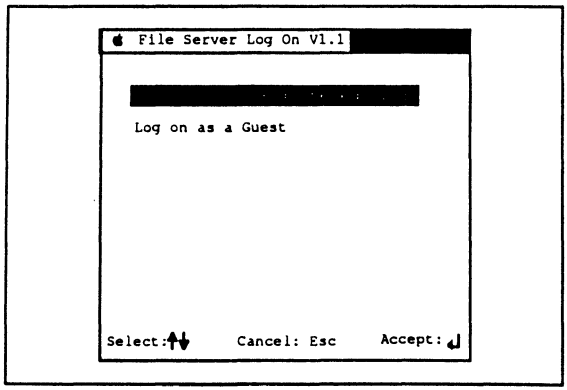
■ **Figure 1-4** File Server List



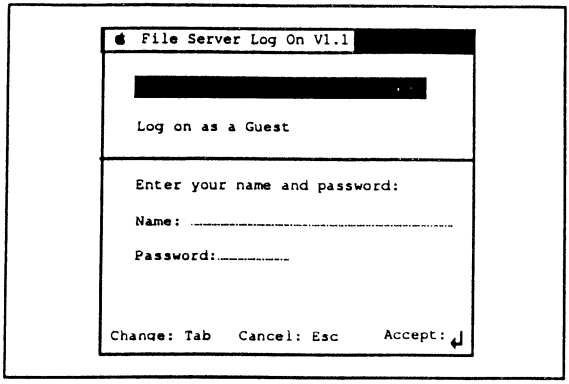
■ **Figure 1-5** Zone List



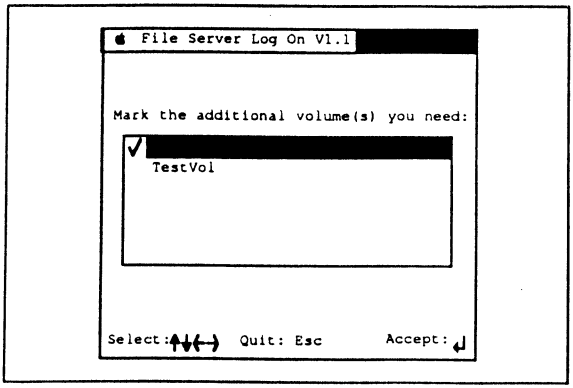
■ **Figure 1-6** Logging On



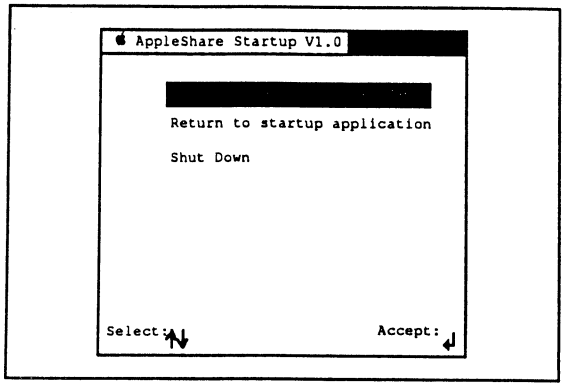
■ **Figure 1-7** Entering Your Name and Password



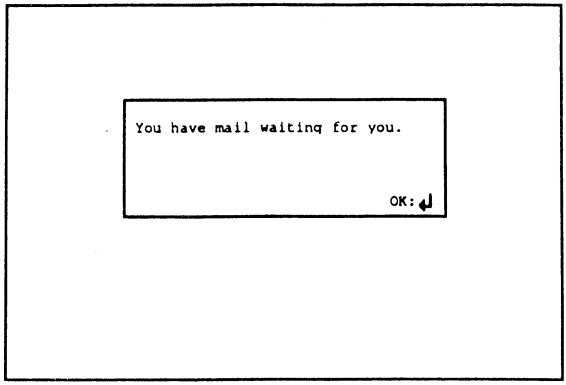
■ **Figure 1-8** Selecting Volume



■ **Figure 1-9** AppleShare Startup Menu



■ **Figure 1-10** "You have mail" dialog



File access

Users can access file functions over the network (such as running programs or saving data files) as if they were using a local disk drive attached directly to their workstation.

◆ *Note:* Only ProDOS 8 and GS/OS applications are supported on the network. Other operating systems, such as DOS 3.3, are not supported.

An important feature of the Apple IIGS workstation is that it does not require the use of memory banks 0 and 1 because most ProDOS 8 applications use all (or almost all) of those banks, including important "reserved" areas within those banks.

Network applications may take advantage of multi-user file access, byte-range locking, security restrictions, and other advanced features available with this full file server implementation. There are some new ProDOS calls that provide access to these features, described more fully in Chapter 3. For example, you can use the Byte Range Lock call to prevent other users from accessing the range of data that you are in the process of manipulating.

The AppleTalk protocols reside in the Apple IIGS firmware and RAM. While ProDOS 8 is active, the protocols are accessible through the same entry point at which ProDOS 8 MLI (Machine Language Interface) calls are made. The MLI is the portion of ProDOS 8 that receives, validates, and executes Operating System commands. Calls to the MLI include housekeeping calls, filing calls, memory calls, and interrupt handling calls. Refer to Chapter 2 for more information on entry points. (While GS/OS is active, you should use GS/OS to make Operating System calls, or RAMDispatch to make AppleTalk calls for other AppleTalk protocols.) Many of these functions are available as FST-Specific calls. Use them instead.

◆ *Note:* There is no file service in System Software 4.0.

Printing over the network

The AppleTalk software contains a module called the Remote Print Manager (RPM). RPM allows transparent printing to remote printers through the SSC entry points to AppleTalk. This emulation occurs by having the firmware appear to be a Super Serial Card (SSC). Both BASIC and Pascal entry points are supported through vectors. (Refer to "Entry Points" in Chapter 2.) These vectors allow the emulator (RPM) to capture the calls that an application makes to send data.

Using RPM, the workstation can print to any device supported by the Chooser (control panel on the Apple IIGS) that registers its name as a printer on the network, including

- ImageWriter II printers with AppleTalk adapter cards installed
- LaserWriters with the ImageWriter II emulator installed

The workstation can print either directly or, if connected to the server, through the print spooler.

The printing task begins when you issue a `PR#n` command, where n is the slot RPM is using. When the call is made, the emulator attempts to open a connection to the device selected from the Chooser. After the connection is opened, the application program sends print data out through the SSC entry points. Print data is sent to the COUT hooks normally directed to the SSC. (Refer to the *Apple IIGS Firmware Reference Manual* or the *Apple IIe Technical Reference Manual* for more information.)

The Apple II workstation captures the stream of characters to be printed. RPM buffers the data, translates the print commands to the equivalent PAP call formats, and sends it to the selected network printer or print spooler.

It is not necessary to call PAP directly to accomplish printing. Printers are selected using the Chooser on the Apple IIe or the Control Panel on the Apple IIGS. (You must select a printer with the Chooser/Control Panel after you first boot from a local drive.)

There are three ways to print over the network:

- The Print Manager. (Apple IIGS only)
- The Remote Print Manager (RPM)
- The Printer Access Protocol (PAP)

To identify the slot RPM is using (to give the user a more readable option, such as "Network Printer," instead of "Slot n"), follow these steps:

1. Make the AppleTalk GetInfo call ($02, see chapter 3)
2. If the call returns an error, AppleTalk is not present or not installed correctly, and there is no network printer.
3. If the "Completion Rtn Return" address is of the form $0000CnXX, where $1 \leq n \leq 7$, then RPM is using slot n and the entry point is $Cn00. Be Sure to test all four bytes.

◆ *Note:* If this call is made from ProDOS 8, the bank byte (third byte) will always be zero.

4. Otherwise, the RPM slot is unknown (and might not be present). Ask the user for the slot.

Chapter 2 **Programming Guidelines**

THIS CHAPTER describes guidelines for creating or modifying an application program for an Apple II workstation that will run on an AppleTalk network system. These functions create many new opportunities and challenges for the Apple II family application programmer. To address this, you need to understand some of the implications of programming for a shared environment. This chapter also discusses how to write your own protocols, as well as formats and conventions. Chapter 3 provides a description and parameter list for each call.

This chapter explains the changes and additions made to the AppleTalk® protocol stack for the Apple II workstation. It is assumed that the reader has read and has access to *Inside AppleTalk* ■

Programming for the shared environment

In the network environment, there are four categories of application development:

- **Single-user** (private data) applications that allow only one user at a time to make changes to a file
- **Multi-user** (shared data) applications that allow two or more users to concurrently make changes to the same file, and correctly coordinates those changes
- **Single-launch** applications that allow only one user at a time to launch and use a single copy of the application
- **Multi-launch** applications that allow two or more users at a time to launch and use a single copy of the application

Single-user and *multi-user* describe data file-sharing modes, and *single-launch* and *multi-launch* describe the launching characteristics, or program file-sharing modes, of the application.

Most applications for the Apple II workstation are accessed by a single user on a single computer. These programs take advantage of the simplicity of the single computer environment, and are often written with the following assumptions.

- Access to applications and data is private.
- Read/Write block calls are frequently made to "private" disk areas.
- File-naming and directory conventions are often fixed.
- Temporary files are often used with fixed names.
- Interrupts are frequently locked out or not considered.
- Printing sometimes uses direct hardware access, and often assumes slot 1 or slot 2.
- Copy protection schemes often depend on specific hardware.

While these assumptions are valid for a single-user environment, they do not apply to a network environment where several workstations may simultaneously access the same application program, data files, and resources on the file server.

▲ **Warning** Writing an application program to run in a network environment using the assumptions used in a single-user environment just given may result in data loss. ▲

General programming guidelines

To create or modify programs to run effectively on the network, you should take into consideration the following general guidelines for programming in a network environment.

- ProDOS 8 and GS/OS utilize a hierarchical file structure. As with the Macintosh server, you must use calls that support this file structure.
- All files residing on a server volume that supports AFP will have two forks: a data fork and a resource fork (even if the resource fork is designated as empty, such as in an MS-DOS application). An application should not write to itself, to its data fork, or to its resource fork.
- An application should *not* close a file while making changes to its contents. For example, an editor may fail that follows this sequence: opens, reads, and closes a file, allows a user to edit data in memory, and then opens, writes, and closes the file. It is important to follow this sequence: open a file, read the file, edit, write to the file, and then close the file. This will prevent other users from modifying the file while you are editing it, (and prevent you from over-writing their changes when you save the file).
- An application should inform the user what access was granted to the document during the open process. An application should allow the user to specify the access desired (such as read-only when the user wishes to view only, but not edit a file).
- An application must be intelligent about the way it manages temporary files. Do not use fixed names; instead, generate a random name or suffix combined with the time or network address.
- Program segmentation should be kept to a minimum.
- Don't assume that, because a file exists, you can open it.

For a more complete explanation of each item, refer to the Preliminary Note, *Software Applications in a Shared Environment*. The sections that follow provides more detailed information on entry points, program compatibility, interrupts, and so forth.

Entry points

The AppleTalk protocols reside in the Apple II workstation firmware and RAM, and are accessible through the same entry point through which ProDOS 8 MLI calls are made. The entry points for making calls to the IIGS firmware include the following:

- The MLI entry point at $BF00 under ProDOS 8, which accommodates all ProDOS 8 MLI calls and all network protocol layer calls. It must be in bank 0 to retain compatibility with the Apple IIe workstation.

- The SSC (Super Serial Card) entry points at $CnXX (n = RPM slot), which is used for printing to remote devices. This entry point emulates the Super Serial Card BASIC and Pascal 1.1 entry points.
- RAMDispatch, which is used under GS/OS for full native mode access in applications.
- File system calls should be made from GS/OS. FST-Specific calls should be used instead of their PFI counter parts. See Chapter 3.

Under ProDOS 8, the PFI entry point of the Apple II workstation is inserted into the MLI vector at $BF00 by the AppleTalk setup during the boot process, thus causing all ProDOS calls to be captured by the Apple II workstation firmware. This process allows PFI to decide whether or not a call is to be a local call or a server AFP call and handle it accordingly. If PFI requires a server AFP call, the appropriate packet or packets are created and sent through ASP to the proper server, resulting in transparent file access to remote files.

◆ *Note:* The $BF page of memory (addresses $BF00 through $BFFF) contains the system global variables. The Apple IIGS maintains certain of these variables for local operating system and assumes them to be in "normal condition." Use the ProDOS 8 ONLINE call for this information. See the *ProDOS 8 Technical Reference Manual* for more detail.

An application program can use the RamDispatch entry point to call AppleTalk directly in full native mode on the Apple IIGS The application can be running code in any bank and still use this call. *Table 2-1* describes the RamDispatch entry point. You can use this entry point while under ProDOS 8 or GS/OS; however, this is the *only* entry point to use while GS/OS is active.

Table 2-1 RamDispatch entry point

Entry Point	Routine	Description
RamDispatch	$E11014	This vector is the direct entry point into the commanddispatcher. It should be called in full native mode with the X (low) and Y (high) registers pointing to the parameter list to be dispatched.

Refer to "Accessing AppleTalk Protocals Directly" in this chapter for more complete information on entry points.

The AppleShare FST is the implementation of AppleShare for GS/OS. It is meant to supersede AppleShare IIGS, the implementation of AppleShare for ProDOS 16. Since ProDOS 16 made calls to ProDOS 8 to get its work done, it patched the ProDOS 8 MLI to intercept calls bound for the network. In this way, both ProDOS 8 and ProDOS 16 can use network volumes. GS/OS is completely separate from ProDOS 8. The ProDOS 8 MLI will still be patched to intercept network calls while ProDOS 8 is running. When GS/OS is running, GS/OS will make calls directly to the AppleTalk routines via the AppleShare FST, instead of calling ProDOS 8 to make the AppleTalk calls.

Program compatibility

An application designed using the firmware calls should result in full compatibility with the current implementation of AppleTalk on the Apple II workstation, as long as the application:

- follows ProDOS 8 conventions
- places all code and buffers in bank 0

Other implementations may be unable to manipulate data outside of bank 0. Therefore, code that makes use of any data bank other than 0 in the Apple II workstation may not be compatible with AppleTalk implementations for the Apple IIe computer.

In order for AppleTalk protocols to work properly on an Apple II workstation, you must use GS/OS to boot; GS/OS files are relocatable files that must use the System Loader. After the system boots up, you can run ProDOS 8 applications. You can also make calls directly to the AppleTalk dispatcher when running in full native mode.

Because of the structure of the system interface, any application or language that uses the ProDOS MLI properly should be compatible (excluding READ_BLOCK and WRITE_BLOCK calls). GS/OS applications that use GS/OS and the FST -specific calls, instead of PFI calls, should also be compatible.

Overlays

You should avoid program overlays whenever possible. If you must use overlays, they should be a minimum of 512 bytes and preferably no more than 4K bytes (1 to 8 blocks) to minimize the number of overlays required. Use READ (not READ_BLOCK) to get overlays. Segment overlays carefully to minimize swapping and reduce network traffic.

As in the single-user environment, overlays cause delays. If your application uses a memory expansion card, you can achieve better performance by downloading overlays to the memory expansion card at initialization time, for later use. Doing this prevents the overlays from possibly slowing down the network. Never write data or configuration information into an overlay (or main program), as this will create problems in a multi-user or multi-launch environment.

Writing into programs

To avoid conflicts among simultaneous users, multi-launch programs must never write data (such as configuration information) back into themselves. You must create a unique name or store the information in a unique, known location, such as a subdirectory named after the user (or the user's directory on the server). A suggestion for creating a unique name is to append the network number and the node to the filename.

Network ProDOS READ and WRITE calls

The network server operating system manages disk access on the server volumes. For this reason, you should always use the READ call and the WRITE call; *never* use the READ_BLOCK call and the WRITE_BLOCK call. The server will not accept READ_BLOCK and WRITE_BLOCK calls and will return a network error ($88). Those calls may be still used for local disk access, but are not recommended.

Unique filenames for temporary files

Names of temporary files must now be checked to prevent duplicate filename problems. If your application program creates temporary files or saves files using default filenames, you must provide a way to add a random suffix or to give the user the opportunity to create a unique filename. Otherwise, Apple II workstation users will save or write to the same files, resulting in lost data or crashed applications. However, this activity will not crash the server.

Memory-resident data files

To prevent changes to a file by other users, you must first determine whether or not data is kept resident in memory.

For programs that load entire data files into memory, the application can check the modification date and the modification time in order to distinguish between files. By checking this combination, the program knows if the file has been changed since the initial read. The application can then prompt the user either to save the data under a different file name or to overwrite the data in the existing file.

For programs that do not keep data resident in memory, you can prevent data changes by other users while you are manipulating data by doing the following:

- using the special open-with-deny mode, or class 1 open in GS/OS
- byte-range locking the entire file

ProDOS 8 Compatibility on the IIe and IIGS

This section describes areas which could cause an application to run under the AppleShare Apple IIe workstation software, but fail under the Apple IIGS workstation software.

- If code is running in auxiliary memory in emulation mode (e.g., ProDOS 8 programs that run code from auxiliary memory), make sure $0100 in auxiliary memory is set to the normal stack pointer and $0101 in auxiliary memory is set to the auxiliary (alternate) stack pointer. (See page 93 of the *Apple IIe Technical Reference Manual*.)

- Make sure ProDOS 8 calls are not made from auxiliary memory; Apple has never recommended doing this, and it is not supported.
- Make sure interrupts are enabled when making ProDOS 8 calls.
- Make sure interrupts are not disabled for long periods of time, nor for a high percentage of time. Whenever interrupts are disabled, there is a chance that an AppleTalk packet will be missed (which could cause AppleShare volumes to be unmounted). The more interrupts are disabled, the more likely that packets will be missed. This risk is inherent for any application that disables interrupts (directly or indirectly), therefore, interrupts should be disabled with discretion and only when absolutely necessary.
- Make sure programs get the completion routine return address from the GetInfo call when they are started.
- Make sure to identify AppleTalk by calling GetInfo and checking for an invalid call number error (which means AppleTalk is not present). Do not use the ATLK signature bytes for identification. See Apple II AppleTalk Technical Note #1, Identifying AppleTalk.
- ProDOS 8 invisible bit is not respected. The invisible bit in the ProDOS 8 access byte was defined after the release of the Apple IIe Workstation Card, so the ProDOS Filing Interface present on the card treats this bit as reserved.

Working with network directories

Network volume directories cannot be manipulated in the same way as directories on local ProDOS volumes. This section describes these reasons and tells how ProDOS 8 and GS/OS applications on an Apple IIGS workstation can properly process network directories. The example routines included in this section work with both local and network volumes; separate routines are not required for local versus network volumes.

Directory and volume name locations

For ProDOS 8, use the pathname at $280 to determine your pathname (refer to Section 5.15 of the *ProDOS 8 Technical Reference Manual* for more information). In GS/OS, prefix 1 is set to the name of the application's directory.

▲ **Warning** Do not hard-code pathnames, directory names, volume names, or their slot/drive locations. ▲

Launching over the network

The following sample program shows how to determine if your ProDOS 8 application was launched over the network.

```
                longa       off
                longi       off
                absaddr     on
                65c02       on
                verbose     on
                Keep        NetLaunch

NetLaunch       Start
mli             equ     $BF00
Lastdev         equ     $Bf30
AtCall          equ     $42
Read_Blk        equ     $80

*****************************************
* NetLaunch checks to see if the last   *
* device accessed was a network volume. *
* An application can run this routine   *
* at the beginning of an application    *
* to see if it has been launched from   *
* a network volume.                     *
*---------------------------------------*
* Inputs:                               *
*     None                              *
*---------------------------------------*
* Output:    *
* Carry Clear = Application launched    *
*           from a local volume.        *
* Carry Set =   Application launched    *
*           from a network volume.      *
*****************************************

CheckNetLaunch  anop
                lda     Lastdev         ;before accessing any disk,
                                        ;get the last device
                sta     UnitNum         ;accessed and store it for the
                                        ;Block Read call
                jsr     mli             ;do the block read
                dc      i1'Read_Block'  ;ProDOS Command
                dc      a'ReadBlock'    ;our parameter list
                cmp     #$88            ;did you get a network error?
                beq     CNL2            ;if not then this is not a network
                                        ;volume
                clc                     ;indicate that the volume is local
                rts                     ;and return
```

```
CNL2            anop
                sec                     ;the volume is on the network
                rts                     ;return

ReadBlock       dc      h'03'
UnitNum         ds      1
                dc      a'BlockBuff'
                dc      a'0'

BlockBuff       ds      512
                end
```

User directories

Because users may configure their workstation differently (such as installing a printer card in different slots, using a different network or server printer), an individual user directory is created by the server's Admin program each time a new user is established. The Admin program creates a directory of the same name as the user's name in the directory USERS on the user volume of the server, and assigns a startup application and printer to each user.

User directories can also be used to modify your application program configurations, allowing each user to configure their own printer, prefix, and so on. To protect your configuration file, your application should either

- create a directory inside the individual user's directory, or
- use the directory SETUP

Both of these directories are located inside the individual user's directory. Your application should first check for the configuration file in the directory you selected; if there is no configuration file in that directory, then return to the directory from which the application was launched. Whenever you make changes to the configuration, store the new information in the directory inside the individual user's directory so that it will be available the next time the application is launched. Using the FIUserPrefix call in ProDOS 8 returns the path to the individual user's directory (refer to Chapter 3 for more detailed information).

Whenever the user executes the Logon program and selects a server, the Logon program automatically mounts that user's volume on the server. The user volume is then available to all of the applications that need individual configuration information for each user.

From GS/OS, you can use the "@" prefix. If your application was launched from a fileserver volume, and a user volume is mounted, it will be set to the user's directory. Otherwise, it will be set to the application's directory.

◆ *Note:* There is no way to determine conclusively, on a per user basis, who is a Apple II user and who is not. Users may also correspond to Macintosh or other AppleShare workstations.

Cataloging ProDOS directories

ProDOS 8 applications often contain routines that catalog or process directories by following the two pointer fields at the top of each directory block. These pointers are links in a chain that connect all the blocks that make up the directory. In a local environment, it has been a common practice to issue Read_Block commands using these pointer values. Additionally, some applications have used the file_count byte of the directory header to keep track of the number of items they are dealing with. In an AppleTalk network, both these practices are unacceptable.

Unlike local ProDOS volumes, file servers are not block devices. An attempt to read a block from a network volume generates a network error ($88). In addition, the directory file_count byte returned by a network server can easily mislead an application. In an environment where multiple users may be creating and deleting files, this file count byte can be made invalid the moment after it is read.

Rather then relying on illegal block reads and unreliable file counts, set up your application to process directories as follows:

1. Open the directory.
2. Issue a READ command with a byte request parameter of 512 bytes. This first READ request gives you the directory's header block. The 24th hex byte in this block contains the number of entries_per_block for the entire directory.
3. Process the entries in that block, and then ask ProDOS for another block's worth of data.

Repeat this process until the ProDOS read command responds with an EOF error. That error indicates you have processed all the entries in that directory.

The programming examples that follow in this section illustrate this technique for processing directories in ProDOS 8.

For GS/OS, always use GetDirEntry to read the contents of a directory.

Searching and deleting from ProDOS directories

When local ProDOS 8 is asked to delete an entry from a directory, it stores $00 in the target entry's Storage_Type/Name_Length byte and updates the Volume Bit Map to release the blocks held by the entry. The deletion of the entry does not remove it from the directory where it resides, but merely marks it as a deleted entry whose space is now available. No reordering of the remaining entries occurs. Because deleting has never caused a restructuring of directories, applications in a single-user environment have been able to safely search and delete multiple entries "on the fly."

You cannot use this approach in a network environment, since network server software does not maintain its directory structures in the same manner as the local operating system. When an entry is deleted from a network directory, the entry name is removed. Entries below it "bubble up" to fill in the gap. An application *must* account for the possibility of this reordering as it deletes multiple files in order for its search routine to see the entries that moved to directory blocks already searched.

Applications can use the following methods to safely delete multiple directory entries, regardless of whether the directory is on a local volume or on a network volume.

- Use the ProDOS 8 Set_Mark function to place the file-location marker at the beginning of the directory after each delete.
- Create a list of items to delete as you search a directory.

The first method assures that the program accounts for all entries as it searches the directory. If your goal is to delete all entries in the directory, this method is easy to implement. However, it might be slow when the program deletes entries from a large directory on a local volume. In such a case, that application must search past a growing number of previously deleted entries as it scans down the list for the next item to delete.

In the second method, you can delete each entry in your list when the end of the directory is reached. If the list fills with entries before the search is completed, close the directory and delete the files in the list, and then reopen the directory and continue searching. By reopening the directory, the application again starts reading from the first entry to assure that no entry is missed.

▲ **Warning** Some applications already use an algorithm similar to the second method by creating a list of indexes into the directory that point to entries to be deleted. However, this method fails in a network environment because the indexes are not updated as the network directory is reordered after each delete. ▲

Under GS/OS, it is more efficient to use the second method. Then use GetDirEntry with base = displacement = 0 to see if there are more entries left. If so, repeat the process. Due to buffering, entries return by GetDirEntry may not be updated immediately after changes are made.

In order to delete entries from ProDOS 8 directories, the netcat.c program must be modified by adding a ProDOS 8 destroy command.

Recursion and network directories

In order for an application to traverse directories of local or network volumes and list ProDOS directories recursively, the routines must not issue Read_Block commands and must use EOF instead of relying on the file count.

▲ **Warning** Because directories in a network environment are subject to change at any time by users, do not recursively catalog and process network directories while the network is being accessed by multiple users. ▲

An example of an application that would be able to use recursive processing is an administrator's utility; such an application might show the organization of files and directories on a server volume. The following ProDOS 8 sample program shows how to traverse and catalog network or local ProDOS volumes.

```
/*
        netcat.c
        An example of how to recursively or non-recursively catalog
        network or local ProDOS volumes without the use of file_counts
        or Read_Block commands.
*/

#include "stdio.h"

#define         max_path_len            65
#define         max_name_len            16
#define         max_list                5
#define         oneblock                512
#define         txt                     0x04
#define         false                   0
#define         true                    !false

/* the following structure is used to control the reading of directory
blocks */
struct infoblk
{
  char          init;           /* indicates "state" of file being read */
  char          refnum;         /* this should be set after open */
  char          entrylen;       /* length of each directory entry */
  int           entryptr;       /* points to current entry within block */
  char          epb;            /* entries per "oneblock" block */
  char          *rbuf;          /* points to "oneblock" byte area of */
                                /* memory for reads */
  char          blockents;      /* number of entries we have scanned in */
                                /* this block */
  long lastmark;                /* need this so we can reco < \dir */
                                /* position if we trash block */
};

struct entryinfo
{
        char    name[max_name_len],     /* name of the directory item */
        access;                         /* its access byte */
};
```

```
char    gpath[max_path_len],  /* pass the path with this global */
                              /* string */
                glist[max_list][max_name_len],
                              /* global list used when deleting */
                slash[2]={1,'/'},
                openbuf[1024+256],
                              /* buffer for when ProDOS opens a file */
                rbuf[oneblock];       /* our read buffer */

int CatDir(path,ftype,recflag)
char    *path,                /* path on which to begin search */
        ftype,                /* type of entry to return, all types */
                              /* if equals 0 */
        recflag;              /* flag to enable/disable recursion */
{
        char    result,
                              dirflag,      /* subdirectory flag */
                              pathmark;     /* legth byte of current path */
                                            /* before going recursive */
                int     operr;
                struct infoblk Myib;
                struct entryinfo thisent;

        /* store length byte of path we are about to search */
        pathmark=*path;
        /* 0 indicates this directory newly opened */
        Myib.init=0;
        /* point to global read buffer */
        Myib.rbuf=&rbuf[0];
        result=1; /* assume file is found and not eof */
        operr=PDosOpen(path,&openbuf[0],&Myib.refnum);
                                            /* open the directory */
        if (!operr) /* if we have accsess and network pathname ok */
        {
                while (result==1)     /* while not eof keep calling */
                                      /* GetNextEntry */
                {
                        /* return next directory item */
                        result=GetNextEntry(ftype,&thisent,&dirflag,&Myib);
                        if ((result==1) || (result==2))
                                      /* found and not eof OR found and eof */
                        {
                                POutputs(path);
                                printf("/");   /*display the path and "/" */
                                POutput(&thisent.name[0]);
                                      /* and display item returned */
                                /* if item is subdirectory, and we are */
                                /* allowed to go recursive */
                                if ((dirflag) && (recflag))
                                {
```

```
                                        /* close parent and process child */
                                        PDosClose(Myib.refnum);
                                        /* append "/" to path */
                                        Pappend(path,&slash);
                                        /* append subdir name to the pathname */
                                        /* we've been in */
                                        Pappend(path,&thisent.name[0]);
                                        /* catalog subdirectory, go recursive */
                                        CatDir(path,ftype,recflag);
                                        /* we're back form recursive call, */
                                        /* restore path to what it was */
                                        * path=pathmark;
                                        /* restore read buffer, do only a */
                                        /* partial init */
                                        Myib.init=1;
                                        /* reopen parent */
                                        PDosOpen(path,&openbuf[0],
&Myib.refnum);
                                }
                        }
                } /* end while loop */
        }
        if (!operr) PDosClose(Myib.refnum);
        return operr;
}

int GetNextEntry(ftype,entryrec,dirflag,info)
                                        /* returns next directory item */
char    ftype,                          /* type of item to search for */
                        *dirflag;       /* set if item returned is a */
                                        /* subdirectory */
struct infoblk *info;
struct entryinfo *entryrec;             /* return matched item in an entry */
                                        /* record     */
{
        char storage_type,found;
        char err;
        int  xferred;

        err=0;                          /* assume no error */
        if (info->init==0)              /* has not yet been read */
        {
                info->init=2;           /* fully initialized */
                err=PDosRead(info->refnum,info->rbuf, oneblock, &xferred);
                info->entrylen=info->rbuf[0x23];
                                        /* pull info out of buffer */
                info->epb=info->rbuf[0x24];         /* entries per block */
                info->entryptr=info->entrylen+4;
                                        /* point to first entry */
                /* start with block entry 2 (dir header name is entry #1) */
                info->blockents=2;
        }
```

```
if (info->init==1)    /* read buf was trashed by a recursive */
                      /* call, restore it */
{
       info->init=2;  /* now we will again be fully */
                      /* initialized */
       PDosSetMark(info->refnum,info->lastmark-oneblock);
                      /* restore mark */
       err=PDosRead(info->refnum,info->rbuf, oneblock, &xferred);
}
found=false;
while ((! err) && (!found))  /* loop til we get an item */
{
       if (info->rbuf[info->entryptr])
                      /* storage_type byte, a type exists if !0 */
       {
              /* high nibble is the storage type  */
              storage_type=info->rbuf[info->entryptr]&0xF0;
              if (storage_type==0xD0)
                     /* set dirflag if this item is a subdirectory */
                     *dirflag=1;
              else
                     *dirflag=0;
              /* get length of item name from low nibble */
              info->rbuf[info->entryptr]&=0x0F;
              /* now check for our entry type or all entry types */
              if (info->rbuf[info->entryptr+0x10]==
                     ftype || ftype==0)
              {
                     /* copy entry to the string */
                     Pstrcpy(&entryrec->name[0],&info->rbuf
                            [info->entryptr]);
                     entryrec->access=info->rbuf
                            [info->entryptr+0x1E];
                     found=true;
              }
       }
       if (info->blockents==info->epb)   /* we need a new block */
       {
              err=PDosRead(info->refnum,info->rbuf,
                     oneblock,&xferred);
              info->blockents=1;
              info->entryptr=4;
       }
       else
       {
              info->entryptr+=info->entrylen;
                                   /* move to next entry */
              info->blockents++;
       }
```

```
        }       /* end while */
        /* find current mark and pass back to info struct */
        PDosGetMark(info->refnum,&info->lastmark);
        if ((found) && (err!=0x4C)) return  1;
                                        /* file found and not eof */
        if ((found) && (err==0x4C)) return  2;
                                        /* file found and eof */
        if (!found) return 3;           /* if file not found then eof also true */
}
main()
{
        char    error;

        error=PDosGetPrefix(&gpath[0]);
        if (!error)
        {
                printf("current prefix :");
                gpath[0]--; /* ditch trailing "/" at end of prefix */
                POutput(&gpath[0]);
                CatDir(&gpath,0,0);     /* send path, all file types, */
                                        /* disable recursion */
                /* to allow recursion, send a '1' as last parameter of */
                /* CatDir */
        }
}
```

Accessing AppleTalk protocols directly

This section describes the implementation details of writing your own protocol specific to the Apple IIGS and accessing AppleTalk protocols directly. You'll need to know the locations of some fixed addresses related to AppleTalk, as well as how to install your protocol, protect your code, and implement an interface to ProDOS.

Entry points

To write your own protocol for the Apple IIGS, you need to know the locations of some fixed addresses related to AppleTalk protocols. These fixed addresses are related to the following entry points:

- $Cn00 Interface (making calls through BASIC and Pascal entry points)
- Super Serial Card emulation
- A unique protocol

The routines related to each of these entry points are described in the tables given next.

Making calls through BASIC and Pascal

The entry points listed in *Table 2-2* allow you to take control from users who are making calls through the BASIC and Pascal entry points. Install a JML to your routine at the address specified for the call so that your protocol is called whenever an action is required. You will be called in native mode with 8-bit M and X.

- **Table 2-2** $C700 interface-related entry points on the Apple IIGS

Entry Point	Routine	Description
BASIC	$E11004	All BASIC calls are routed through this entry point. The following conditions are set: Carry=clear for output, set for input Overflow=set for init If you are providing your own BASIC vector, as RPM does, you must set up CSWL and KSWL just as any interface code does.

(continued) ➨

■ **Table 2-2** ***(continued)*** $C700 interface-related entry points on the Apple IIGS

Entry Point	Routine	Description
Pascal	$E11008	This routine is called whenever someone makes a call through the standard Pascal interface. The low nibble of the Y register contains the Pascal command that is being called. These numbers are as follows: $01 for Status $02 for Write $03 for Read $04 for Init *Note:* This routine is already set up in RPM; you don't need to make any changes unless you want to change RPM.

Serial card emulation

The next group of entry points listed in *Table 2-3* are related to serial-card emulation, and use the built-in serial firmware to perform the necessary conversions. These conversions are generally only useful if you plan to replace RPM with your own code. They are all called in native 8-bit M and X.

■ **Table 2-3** Serial card emulation entry points on the Apple IIGS

Entry Point	Routine	Description
SerStatus	$E11026	Firmware calls this routine to find out the status for input and output. The byte returned must look as if it were an actual status byte from the SCC.
SerWrite	$E1102A	Firmware calls this routine to output a character. This will only be called when you have given the go-ahead via SerStatus.
SerRead	$E1102E	Firmware calls this to read 1 byte for you after SerStatus gives its permission.

Unique protocol

The final group of routines listed in *Table 2-4* are related primarily to writing your own protocol.

■ **Table 2-4** Unique protocol entry points on the Apple IIGS

Entry Point	Routine	Description
RamGoComp	$E1100C	This routine allows you to call a completion routine the properway when writing a protocol (routines in bank 0 called in emulation mode; all other banks full native mode). It should be called in 16-bit M,X native mode with the address of the routine to be called at locations $84—$87.
SoftReset	$E11010	This vector is actually part of a chain of routines to be called when control reset has been hit. This chain gives you the opportunity to reinitialize your code before the application regains control. The section on writing your own protocol provides the necessary details on how to install and make use of this vector.(Each AppleTalk protocol is installed in the reset chain in order to initialize itself on reset.)
RamDispatch	$E11014	This vector is the direct-entry point into the command dispatcher. It should be called in full native mode with the X (low) and Y (high) registers pointing to the parameter list to be dispatched. (This is the safe way to call AppleTalk protocol with a parameter list not in bank 0. Parameter lists for calls made through the PFI can be in banks other than 0.)
RamForbid	$E11018	This vector disables packet and timer interrupts without physically disabling interrupts. Using this vector is a safe way to protect a portion of code from being entered during an interrupt. (For a list of which AppleTalk calls can be executed after RamForbid has been called, refer to *Table 2-5*)
RamPermit	$E1101C	This vector reenables packet and timer interrupts. You must call this vector after a RamForbid.
ProEntry	$E11020	This 2-byte address of routine in page zero is called if a command is not dispatched by code at ProDOS 8 vector. (This address is generally the entry point to ProDOS 8 itself, to which $BF00 points.)

(continued) ➨

■ **Table 2-4** ***(continued)*** Unique protocol entry points on the Apple IIGS

Entry Point	Routine	Description
ProDOS	$E11022	This address is for the routine to be called when ProDOS 8 calls have been rerouted through the $C7AE vector in the interface code. Doing this allows you to trap calls that are headed to ProDOS 8 and to do whatever you need to do. The details of using this vector can be found in the section on the ProDOS 8 interface.
CmdTable	$E1D600	This routine is the beginning of the command table. This table holds 256 four-byte entries and extends to $E1D9FF. Since this table is in the language-card area of bank $E1, it is essential that, whenever you need to access it directly, you save the state of the language-card, and then force in language card bank 2. Then you must restore the original language card state.
TickCount	$E1DA00	The 2-byte value is the current number of ticks that have expired since AppleTalk protocols were initialized. This count is not reset to 0 if the RESET key is pressed and AppleTalk protocols reinitialized. Since this value is in the language-card area of bank $E1, it is essential that, whenever you need to access it directly, you save the state of the language card, and then force in language-card bank 2. You must then restore the original language-card state.
Priority Vector	$E1103A	This vector is used during load of SYSTEM.SETUP files to ensure proper order of load.
PFI Vector	$E1103E	This vector is the address of code to get called on all MLI calls. *Note:* If this address is 0, all calls go directly to ProDOS, except for Command $42.

Installing a unique protocol

To write your own protocol, you must first install the command into the command list.

To do this, you must take the command number you are installing (or replacing) and multiply it by 4 to get an offset into the command table. This offset should then be added to $E1D600 (the start of the command table), which gives you the address to install a vector to your code. This vector consists of a 3-byte pointer, followed by 1 byte for the amount of zeropage you would like saved when you are called. Zeropage is saved starting from $80; therefore you should use from $80 on up for your own use.

◆ *Note:* Since this table is in the language card area of bank $E1, it is essential that whenever you need to access it directly that you save the state of the language card, then force in language card bank 2. Then you must restore the original language card state.

For example, if you were replacing command $01, you would store the address of your routine at $E1D604-$E1D606, and the amount of zeropage you would like saved at $E1D607.

The reset chain

After you install your command, install your code into the soft reset chain. The mechanism of calling routines in the chain ensures that the routines first installed are called first.

In order to maintain this order, you must do the following to install your code:

1. Take the code that is currently in that vector and save it somewhere within your program; you will need to call it later.
2. Install a JML to your own reset routine. When reset is pressed, your routine is called.

Before executing any code, however, you must allow the routine installed prior to your code to be executed (this is the code that you saved when you installed your code). Do this by doing a JSL to that code, which returns to you when it is done. Only then can you execute your own reset code. That code should be called in full native mode. Your reset code should also preserve the state of the machine when exiting.

Interrupts and protecting your code

It is sometimes necessary to guarantee that you are not interrupted during a critical section of your program (a routine that might be called during an interrupt). It's possible to execute an SEI (Set Interrupt) to disable interrupts physically. If you do so, SEI should be executed only for short periods; otherwise, AppleTalk incoming packets may be missed completely.

▲ **Caution** To maintain AppleTalk performance on the Apple IIGS, however, you should not execute an SEI. AppleTalk protocols require interrupts to be enabled to function normally. ▲

In order to alleviate this problem, Apple provides two RAM-based routines (RamForbid and RamPermit) that disallow AppleTalk-related interrupts without physically turning off interrupts:

- RamForbid basically increments an internal flag that causes LAP to buffer any packets, without dispatching them while that flag is set.
- RamPermit decrements that flag; when it reaches 0, sockets are again dispatched.

These are particularly useful in routines that may be called by completion routines or socket listeners. Any packets that might have been buffered are dispatched, providing a safe but effective mechanism to protect your code. These routines do not lock out any system interrupts.

▲ **Warning** If you make a call to the AppleTalk firmware with interrupts turned off, the Apple IIGS will hang. ▲

The Apple IIGS contains its own interrupt handler. Applications should not mask out interrupts in general, especially when making an operating system call that results in a network call. The Apple IIGS always buffers packets (if it has buffers remaining to be processed). Any packets that might have been buffered are forwarded on demand, as required. For additional information, refer to "Restrictions" later in this chapter.

Using completion routines

A completion routine is used only for asynchronous calls, and is a routine that is called under an interrupt. When a completion routine gets control, RamForbid will have already been called, so you cannot get interrupted in a completion routine.

◆ *Note:* A non-zero completion address will not be called on a synchronous call.

When a completion routine is called, the databank register is set to E1, and is called in full native mode (16-bit accumulator and 16-bit X and Y registers), unless the routine you are calling is in bank 0. If your code is in bank 0, then the code gets called in emulation mode (8-bit). If the routine is in any other bank, it will be called in native mode. Direct page (D register) will be set to $0000. Locations $80-$83 contain a pointer to the parameter block for the call.

Any zero pages or anything you use should be preserved, because they are basically an interrupt routine. The Apple II workstation takes care of switching to emulation mode and switching stacks for you.

You must exit completion routines, socket listeners, and protocol handlers via a jump (rather than an RTS) to the address returned in the Completion Routine Return field of the GetInfo call ($02). To exit properly, you should call the GetInfo command just once at the beginning of your program to get the address. When writing the completion routine, you need to call that vector (from the GetInfo call) with a JML (on the Apple IIGS) or a JMP (on the Apple IIe) to that address.

Restrictions

There are some restrictions concerning the use of completion routines and interrupts and the time that certain types of calls may be made. For example, going to the Control Panel creates an interrupt. These restrictions are as follows:

- *The code should not be called during an interrupt routine for a device other than AppleTalk itself.* Since the firmware does not lock out interrupts when it is called, an interrupt from another device could happen while a call is in progress. If a call to the AppleTalk firmware is executed during this time, it would result in one call being executed on top of another call. The firmware cannot handle such a situation. The firmware itself will not cause an interrupt once it has begun processing a call.

◆ *Note:* This restriction does not apply if you follow guidelines for the Task Scheduler in the IIGS Toolbox.

- *Never enable an interrupt from within an interrupt routine, such as a completion routine, socket listener, or protocol handler.* Doing so could cause a second pending interrupt posted by the Apple II workstation to occur, and could result in the same problems as mentioned in the restrictions just given.
- Do not disable interrupts for a long period of time when using calls that make use of completion routines, socket listeners, or protocol handlers. These calls need interrupts in order to complete, and packets may be lost because the packet buffers may overflow.
- Certain AppleTalk calls cannot be executed in synchronous mode from a completion routine, socket listener, or protocol handler (any time that RamForbid has been previously called). Because interrupts must be locked out during a completion routine, and because asynchronous calls need interrupts to complete, even when executed synchronously, a conflict will occur.

The packet dispatcher automatically calls RamForbid when it issues packets for completion routine, protocol handlers, socket listeners, and so forth. The above restriction also applies if your application calls RamForbid for any reason. *Table 2-5* lists which AppleTalk calls can be executed after RamForbid has been called.

■ **Table 2-5** Issuing AppleTalk calls protected by RamForbid on the Apple IIGS

OK anytime	OK anytime if asynchronous, but only OK in synchronous if RamForbid not yet called	Only OK if RamForbid not yet called
GetInfo ($02)	InstallTimer ($04)	FILogin ($2B)
GetGlobal ($03)	RegisterName ($0E)	FILoginCont ($2C)
RemoveTimer ($05)	LookupName ($10)	FILogout ($2D)
Boot ($06)	ConfirmName ($11)	FIMountVol ($2E)
LAPWrite ($07)	SendATPReq ($12)	FIAccess ($32)
ReadBuffer ($08)	GetATPReq ($16)	
AttachProt ($09)	SendATPResp ($17)	
RemoveProt ($10)	GetMyZone ($1A)	
OpenSocket ($0B)	GetZoneList ($1B)	
CloseSocket ($0C)	SPGetStatus ($1D)	
SendDatagram ($0D)	SPOpenSession ($1E)	
RemoveName ($0F)	SPCloseSession ($1F)	
CancelATPReq ($13)	SPCommand ($20)	
OpenATPSocket ($14)	SPWrite ($21)	
CloseATPSocket ($15)	PAPStatus ($22)	
RelATPCB ($19)	PAPOpen ($23)	
SPGetParms ($1C)	PAPClose ($24)	
PMSetPrinter ($27)	PAPRead ($25)	
FIUserPrefix ($2A)	PAPWrite ($26)	
FIListSessions ($2F)		
FITimeZone ($30)		
FIGetSrcPath ($31)		
FINaming ($33)		
ConvertTime ($34)		
FISetBuffer ($36)		

Formats and conventions

This section describes the formats and conventions used in making the two basic types of calls: synchronous and asynchronous.

Synchronous calls are calls that complete while you wait. The caller's program makes a call to the entry point, and simply waits for control to return to the application program with a result code.

Asynchronous calls are calls that do not complete immediately, even though control is returned to the calling program immediately. The workstation can continue with another task while an asynchronous call is in process of completing. You can make multiple asynchronous calls; however, the order of completion may be different than the order in which you made the calls. The number of pending asynchronous calls may be limited.

When an asynchronous call is made, the parameter list becomes the "property" of the network until the call completes. Your program supplies the address of a completion routine in the call; when the Apple II workstation completes the call, it interrupts your program and causes a jump to the completion routine. The firmware initially returns an $FF in low byte of the Result Code field in the parameter list to indicate that it is in the process of completing. When the call is complete, the Result Code fields change to the final status.

Asynchronous calls versus synchronous calls

The programmer should never make a synchronous only call with the async flag set (bit 7 = 1). Although some synchronous only calls can be made with the async flag set, the results can be unpredictable. In most cases, the call will complete with no detectable side effects, but others will hang or crash.

The format of the calls for ProDOS 8 is shown in *Table 2-6*.

■ **Table 2-6** Call format for ProDOS 8

Call	Description	
Call Format	JSR	$BF00
	DB	COMMAND ($42 for AppleTalk calls)
	DW	PARAMETER LIST ADDRESS
	BCS	ERROR ROUTINE
On Exit	A = ERROR CODE	($00 = No Error, $88 = Result Code contains error)
		CARRY SET = ERROR
		CARRY CLR = NO ERROR

Table 2-7 shows the format for calling AppleTalk protocols directly in full native mode on an Apple IIGS workstation.

■ **Table 2-7** Non-FST call format for GS/OS

Call	Description	
RamDispatch	$E11014	(Pointed to by vector at $BF00)
Call Format	LDX	#parmlist
	LDY	#^parmlist
	JSL	>RamDispatch
On Exit	A = ERROR CODE	($00 = No Error, $88 = Result Code contains error)
		CARRY SET = ERROR
		CARRY CLR = NO ERROR

Note: This format allows the Apple IIGS workstation to have code and data in banks other than 0. However, you can use this format from ProDOS 8 as well on an Apple IIGS workstation.

Parameter list format

For all AppleTalk calls with the above formats, the first 4 bytes of all parameter lists are the same.

The first byte is the Async Flag field. This flag indicates whether the call should be executed in asynchronous mode or synchronous mode.

The value supplied in the Async Flag indicates whether such a call is to be executed synchronously or asynchronously.

- If MSB (bit 7) is set, the call executes asynchronously.
- For calls that cannot be executed asynchronously (those without completion routine pointers), MSB must be set to 0; the call then executes synchronously.

Calls that may be executed asynchronously contain a Completion Routine Pointer in the parameter list after the Result Code field. If the Completion Routine Pointer field contains a value other than 0, the field is interpreted as an address to be called when the command being executed completes.

▲ **Warning** For calls listed as synchronous only, this pointer must be 0. Also, do not modify the parameter list of a call made asynchronously, since the parameter list belongs to AppleTalk protocols until the call is completed. ▲

The second byte is the AppleTalk Command field. All non-ProDOS 8 calls to the protocol layers are made using the MLI command $42 in the command byte of the parameter list. There is an AppleTalk Command code in the second byte of the parameter lists of these calls specifying the actual command that is to be executed.

The next two bytes contain a Result Code field, in which the actual error is returned. The following section describes how errors are returned.

How errors are returned

If an asynchronous call was made with a completion routine specified, the firmware transfers control to the completion routine when the call completes, usually under an interrupt. If no completion routine was specified, the caller must check the result code field periodically and take the required action, including freeing the parameter list memory if necessary.

When an error occurs during a network or other non-operating system call (Command $42 for ProDOS 8), a single standard error code is returned in the accumulator (Network Error = $88). The parameter lists for all of these calls contain a 2-byte Result Code field in which the actual error is returned. When there is no error, both the accumulator and the Result Code field contains a value of 0 (successful).

The Result Code field contains both a "level" indicator and the actual result code. The high byte of the Result Code field contains a value indicating the protocol layer called, except for special errors returned for any layer ($0101, $0102, and $0104); the low byte contains the actual error code. Values of $C0 through $DF in the high byte are reserved for additional errors to be returned by code that the user adds to the dispatcher. For asynchronous calls, the low byte of the Result Code field is set to $FF (busy) while the call is executing; the high byte contains a number indicating the protocol layer called. It is important to check both fields. All calls at all levels may encounter the system-level error conditions listed in *Table 2-8*. (These calls are in the range of $01xx.) Actual error codes for each call are listed in Chapter 3.

Table 2-8 General result codes

Result Code	Description
$0101	Invalid command
$0102	Heap/memory management error
$0104	Sync/Async call error

For example, if a DDPCloseSocket call is executed specifying a Socket Number that is not open, the call completes with the carry set, the Network Error ($88) code in the accumulator and a Socket Not Open error in the Result Code field of the parameter list. This technique provides separation between operating system errors and those errors returned by network layers, and also provides space for a larger number of error codes.

◆ *Note:* It is important that an application program maps errors properly and interprets them for the user in the most simple and accurate language. An application program should inform them of the relationship of any recent action to the error. For example, if a user attempts to open a file for which he does not have access, do not return I/O Error or the Network Error. Such error messages are not specific and could represent any number of problems.

Conventions

All address values in the AppleTalk parameter lists are 4 bytes to allow for larger address spaces in future development. All multibyte values are in low-byte to high-byte order, except as noted. Each item in a parameter list needs the following information:

- Relative offset from the beginning of the list (Position)
- Parameter Name
- Parameter Size
- Parameter Value
- Optional Comments

Table 2-9 lists possible entries in the parameter Size field. If the parameter Size is not one of these values, the actual length in bytes will be given (for example, 6 bytes).

- **Table 2-9** Entries in the parameter size field

Parameter Size	Description
Byte	1 byte
Word	2 bytes, in low-to-high order
<Word>	2 bytes, in high-to-low order (reverse order)
Long	4 bytes, in low-to-high order
Var	Variable

The Value field indicates whether the caller supplies the parameter or the parameter is returned by the call. The field also indicates the value of caller-supplied constants. *Table 2-10* lists possible entries in the Value field.

- **Table 2-10** Entries in the Value Field

Value	Description
Constant	Caller-supplied constant
--->	Caller-supplied parameter
<---	Parameter returned by call
<-->	Parameter supplied by caller, returned by call, or both
x	Reserved field used only by call

▲ **Warning** Tampering with Reserved fields may result in your application not working properly, because these fields may be used to hold temporary values while the command is being executed. ▲

The following example shows how parameter lists will look in Chapter 3.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$06
$02	Result Code	Word	<---

Chapter 3 **Calls to AppleTalk Protocols**

THIS CHAPTER describes the calls to the various protocols, with a description and parameter listing for each call. The following AppleTalk protocols are currently implemented on the Apple IIGS workstation:

- LocalTalk Link Access Protocol (LAP; also known as LLAP)
- Datagram Delivery Protocol (DDP)
- Name Binding Protocol (NBP)
- AppleTalk Transaction Protocol (ATP)
- a subset of the Zone Information Protocol (ZIP)
- the workstation side of the Printer Access Protocol (PAP)
- the workstation side of the AppleTalk Session Protocol (ASP)

For a more detailed description of each of these protocols, refer to *Inside AppleTalk*. This chapter also includes general housekeeping and support calls, along with result codes for the calls on each layer. ■

In addition, the Apple IIGS workstation contains

- the Remote Printer Manager (RPM) for transparent printer access
- serial drivers for the serial port
- the ProDOS Filing Interface (PFI), which allows transparent file access by translating non-local or network filing calls into AFP (AppleTalk Filing Protocol) calls. Features of AFP that are not available through ProDOS or GS/OS calls can be accessed by making AFP calls directly through ASP.
- AppleShare FST to provide access to AFP Servers from GS/OS.
- AppleTalk drivers for accessing file servers and RPM from GS/OS.
- a stub of the Routing Table Maintenance Protocol (RTMP)
- the Echo Protocol (EP)

◆ *Note:* RTMP and EP are built into the AppleTalk firmware on the Apple IIGS workstation, and your program does not need to make any calls or take any action to implement these protocols.

◆ *Note:* PFI, the AppleShare FST and AppleTalk drivers for GS/OS are not available in System Software 4.0.

There are also special routines to provide a timer interrupt. Boot code is provided to allow an Apple IIGS workstation to boot directly over the network from a server that supports boot service.

Identifying AppleTalk

To determine if an application has been launched over the network, refer to the NetLaunch code

Under ProDOS, to identify both AppleTalk and the slot with which is is associated for printing, refer to Apple II AppleTalk Technical note #4, *Printing Through the Firmware.*

To identify AppleTalk under ProDOS 8:

1. Issue an AppleTalk GetInfo call.
2. If there is no error result, AppleTalk is installed. See also 'Printing over the Network: in Chapter 1.

```
InfoParems      DB $00          ;Synchronous only
                DB $02          ;GetInfo call number
InfoResult      DS 13           ;<- results returned here
```

```
CheckTalk       JSR $BF00
                DB $42              ;$42 command # for AppleTalk calls
DW InfoParmas   ;Parameter list address
BCS NoATlk      ;handle the error
IsATlk.                             ;AppleTalk installed when here
NoATalk                             ;AppleTalk not installed when here
```

To identify AppleTalk protocols and AppleShare file system under System Software 5.0:

1. Set up the parameter block for a GS/OS GetFSTInfo call using fstNum = 1.
2. Issue the GetFSTInfo call.
3. If the fileSysID is $OD the AppleShare FST and AppleShare are present.
4. If a parameter out of range error (453) results, the AppleShare file system is not present.
5. Otherwise, if steps 3 and 4 are inconclusive, increment the fstNum and loop back to step 2.

To identify AppleTalk protocols, including LAP through PRI, but excluding the file system, under System Software 5.0:

1. Set up the parameter block for a GS/OS DInfo call using device number one.
2. Issue the DInfo call.
3. If the deviceID is $1D, the AppleTalk main driver and AppleTalk are present.
4. If a parameter out of range error ($53) results, the AppleTalk protocols are not present.
5. Otherwise, if steps 3 and 4 are inconclusive, increment the device number and loop back to step 2.

To identify AppleTalk protocols, including LAP through ASP, but excluding the File system, under System Software 4.0:

1. Issue an SPGetStatus call.
2. If the call returns without error, AppleTalk is present.

◆ *Note:* With the release of System Software 5.0, earlier versions not supported.

Miscellaneous calls

This section discusses general housekeeping and support calls that the application needs to make. These calls are listed in *Table 3-1*. The sections that follow describe each call, the parameter listing, and the result codes.

■ **Table 3-1** General housekeeping and support calls

Command Number	Name	Description
$01	Init	Initialize AppleTalk firmware
$02	GetInfo	Get information
$03	GetGlobal	Get global parameters
$04	InstallTimer	Install interval timer
$05	RemoveTimer	Remove interval timer
$06	Boot	Boot over network
$45	CancelTimer	Cancel InstallTimer

Init ($01)

This section provides background information on the Init call. The boot file makes an Init call, which causes the AppleTalk firmware to be initialized.

▲ **Warning** You should never make the Init call, because it is done for you in the AppleTalk setup files. Making an Init call disconnects all the RAM-based protocols installed by the startup file. ▲

The ATInit file can be found in the System:System.Setup directory of the local boot volume or in the Users:YourName:Setup directory of the AppleShare boot volume (where YourName is the User Name used to log on to the boot server). In all cases, ATInit will contain the three required data fields `UserName`, `PrinterFlags`, and `PrinterTuple` at the end of the file. Before those data fields, ATInit may also contain executable code or additional data fields. Since the three required data fields are directly before ATInit's end-of-file (EOF), you can find them relative to ATInit's EOF using the displacements listed in Table 3-1a.

■ **Table 3-1a** Offsets of required data fields

Displacement to ATInit EOF	Size	Field Name	Description
133	33 Bytes	`UserName`	A Pascal-type string containing the default User Name. It consists of a length byte followed by up to 31 bytes of ASCII data followed by a single unused byte. This field is always 33 bytes long.
100	Byte	`PrinterFlags`	This is the Flags field used by the Remote Print Manager's default network printer.
99	99 Bytes	`PrinterTuple`	This field specifies the name of the default network printer used by the Remote Print Manager. The `PrinterTuple` field is in standard Name Binding Protocol (NBP) format. This field is always 99 bytes long.

If the ATInit file is on an AppleShare server, it will have 6 additional data fields (`PathVolID`, `PathDirID`, `Path`, `PrefixVolID`, `PrefixDirID`, and `Prefix`) directly before the three required data fields. These fields can also be found relative to ATInit's EOF using the displacements listed in Table 3-1b.

■ **Table 3-1b** Offsets of optional data fields

Displacement to ATInit EOF	Size	Field Name	Description
275	Word	`PathVolID`	The Volume ID number of the user's AppleTalk startup application.
273	Long	`PathDirID`	The Directory ID number of the user's AppleTalk startup application.
269	65 Bytes	`Path`	The Pathname of the user's AppleTalk startup application.
204	Word	`PrefixVolID`	The Volume ID number of the user's AppleTalk default prefix.
202	Long	`PrefixDirID`	The Directory ID number of the user's AppleTalk default prefix.
198	65 Bytes	`Prefix`	The user's AppleTalk default prefix.

The displacements in Tables 1 and 2 can be used with the GS/OS `SetMark` call to move the file mark to the beginning of any of the above fields. The `SetMark` call's `base` field should be set to $0001 so the mark will be set equal to EOF minus the displacement.

When a hardware Reset occurs, whether caused by the use of the Reset key or by power-up, the Apple IIGS reinitializes itself through the reset chain and acquires a new node number. All sessions, sockets, and packets are lost.

The Super Serial Card ID bytes are also set as follows (for the slot being used by RPM):

$C705 = $38
$C707 = $18
$C70B = $01
$C70C = $31

In addition, the Logon program will announce mail if there is a folder called MAIL located in the Users folder with anything in it. The Logon program displays the message, "You have mail waiting."

The result codes returned for the Init call are the same as those for all system calls, as follows:

Result Code	Description
$0101	Invalid command
$0102	Heap/memory management error
$0104	Sync/Async call error

GetInfo ($02)

The GetInfo call returns some miscellaneous information that may be needed by applications. The parameter structure for the GetInfo call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$02
$02	Result Code	Word	<---
$04	Completion Rtn Return	Long	<---
$08	This-Net	<Word>	<---
$0A	A-Bridge	Byte	<---
$0B	Hardware ID	Byte	<---
$0C	ROM Version #	Word	<---
$0E	Node Number	Byte	<---

The Completion Rtn Return field tells the caller the address to jump to upon finishing completion routines, socket listeners, or protocol handlers. You must go to this address by means of a jump rather than an RTS. This-Net and A-Bridge return the current values for these fields, which are maintained by the RTMP stub. The Hardware ID field returns an ID for the Apple IIGS workstation; the value of this ID is currently $00. The ROM Version # field returns the ROM version number. On an Apple IIe, the hardware ID and ROM Version # fields have no meaning.

The result code returned for the GetInfo call is as follows.

Result Code	Description
$0104	Sync/async call error

GetGlobal ($03)

The GetGlobal call is used to retrieve the Global Parameter area from the firmware. The Global Parameter area contains the LAP and DDP header data that was extracted by those protocols. The parameter structure for the GetGlobal call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 Synchronous Only
$01	Command	Byte	$03
$02	Result Code	Word	<---
$04	Buffer Pointer	Long	--->

The data is returned in the buffer pointed to by the Buffer Pointer field and has the following format.

Position	Name	Size	Value
$00	LAP Destination Node #	Byte	<---
$01	LAP Source Node #	Byte	<---
$02	Lap Type	Byte	<---
$03	Hop Count/DL (MSB)	Byte	<---
$04	Datagram Length (LSB)	Byte	<---
$05	DDP Checksum	<Word>	<---
$07	Destination Network #	<Word>	<---
$09	Source Network #	<Word>	<---
$0B	Destination Node	Byte	<---
$0C	Source Node	Byte	<---
$0D	Destination Socket	Byte	<---
$0E	Source Socket	Byte	<---
$0F	DDP Type	Byte	<---
$10	Packet Length	Word	<---

The packet length is the length of the entire packet received, including all headers.

The Datagram Length, DDP Checksum, Destination Network #, and Source Network # fields are in high-byte to low-byte order. The data returned in this call is not valid after the ReadBuffer call has been executed with the Purge Flag set (see "Calls to the Link Access Protocol" later in this chapter). This call assumes that the caller's buffer is at least 18 bytes long.

The result codes returned for the GetGlobal call are the same as those for all system calls.

InstallTimer ($04)

The InstallTimer call allows an application to set a time interval and to have the firmware notify it when the interval expires. The parameter structure for the InstallTimer call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$04
$02	Result Code	Word	<---
$04	Completion Routine	Long	--->
$08	Running Tick Count	Word	<---
$0A	# Ticks to Complete	Word	<-->
$0C	Reserved	Long	x

When called synchronously, the command completes when the interval specified in the # Ticks to Complete count expires. When called asynchronously, the completion routine is called when the # Ticks to Complete count expires. The maximum value allowed for the # Ticks to Complete field is up to $FFFF; however, to be compatible with the IIe, the maximum value is $1FFF. The ticks are 1/4-second periods. The value in the Running Tick Count field contains the number of ticks since the Init call was made. An unlimited number of timers may be active at any given moment on the Apple IIGS. Only eight timers may be active at any given moment on the Apple IIe.

▲ **Warning** This parameter list *must not* be modified as long as the timer is active, except if it is being changed to a RemoveTimer parameter list. The Apple IIGS firmware uses this list to identify and track the timer; the list must be available for potential use as a Remove Timer parameter list. ▲

The InstallTimer call returns the result code for all system calls. Additionally, this call returns an Apple IIe specific result code when there are too many timers active.

Result Code	Description
$0105	Too many timers

RemoveTimer ($05)

The RemoveTimer call is used to cancel an asynchronous InstallTimer call before it completes without calling a completion routine. It uses the identical parameter list as the corresponding InstallTimer that is being removed. The parameter structure for the RemoveTimer call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$05
$02	Result Code	Word	<---
$04	Reserved	12 Bytes	x (the rest of the InstallTimer parameter list)

The Async Flag and Command bytes must be changed in the original parameter list used for the InstallTimer call.

The RemoveTimer call returns this result code, as well as the result code for all system calls.

Result Code	Description
$0103	No Timer Installed

Boot ($06)

The Boot call causes a network boot to take place. The parameter structure for the Boot call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$06
$02	Result Code	Word	<---

If no boot server is found, no errors are returned and the Apple IIGS workstation boots locally.

CancelTimer ($45)

The CancelTimer call is used to cancel an asynchronous InstallTimer call before it completes. The timer's completion routine will be called. It uses the identical parameter list as the corresponding InstallTimer that is being removed. The parameter structure for the CancelTimer call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$45
$02	Result Code	Word	<---
$04	Reserved	12 Bytes	x (the rest of the Install Timer parameter list)

The Async Flag and Command bytes must be changed in the original parameter list used for the InstallTimer call. If the timer routine has not been installed or had completed, a "No Timer Installed" error ($0103) is returned. If the timer is successfully canceled, the completion routine will receive a "Timer Canceled" error ($0106). This is important as a successful result for the CancelTimer call will return error $0106 instead of "No Error" ($0000).

The CancelTimer call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0103	No Timer Installed
$0106	Timer Canceled

Calls to the Link Access Protocol (LAP)

The Apple IIGS firmware provides the standard LocalTalk Link Access Protocol (LLAP). It also provides calls to read packet data from the receive buffers, and to attach or remove protocol handlers.

Table 3-2 lists the calls to the LAP layer. The sections that follow describe each call, the parameter listing, and the result codes.

■ **Table 3-2** LAP calls

Command Number	Name	Description
$07	LAPWrite	Write LAP packet
$08	ReadBuffer	Read data from buffer
$09	AttachProt	Attach protocol
$0A	RemoveProt	Remove protocol

The LAP firmware maintains a number of receive buffers in its own reserved RAM. When a packet is received, it is placed in the next available receive buffer. If all buffers are full, the packet is ignored and is lost.

Received packets are processed in the order they are received, with the buffers being handled as a circular queue. When a packet is found in the buffer, the LAP header is pulled off by the LAP layer, the LAP Type table is searched to determine the next protocol layer to call, and control is then passed to that layer. If the LAP Type is not found in the table, the packet is discarded. Headers and data are retrieved from the buffer by protocol layers using the ReadBuffer call.

LAPWrite ($07)

The LAPWrite call is used to send a LAP packet. The parameter structure for the LAPWrite call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$07
$02	Result Code	Word	<---
$04	Destination Node	Byte	--->
$05	LAP Type	Byte	--->
$06	Pointer to BDS	Long	--->

The LAP packet is sent to the node specified in the Destination Node field, with the LAP Type specified in the list. The data to be sent is determined by a Buffer Data Structure (BDS), indicated by the Pointer to BDS. The format of the BDS is shown here.

Position	Name	Size	Value
$00	First Buffer Length	Word	--->
$02	First Buffer Pointer	Long	--->
$06	Second Buffer Length	Word	--->
$08	Second Buffer Pointer	Long	--->
		|	|
		v	v
$xx	Last Buffer Length	Word	--->
$xx	Last Buffer Pointer	Long	--->
$xx	End-of-BDS Flag	Word	$FFFF

The LAPWrite call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0203	LAP data too large
$0204	Retry count exhausted
$0205	Illegal LAP type

ReadBuffer ($08)

The ReadBuffer call allows data to be retrieved from the current packet being processed, including headers for "client protocols". It enables protocol layers to extract only their own headers without disturbing the remainder of the packet. The parameter structure for the ReadBuffer call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$08
$02	Result Code	Word	<---
$04	Request Count	Word	--->
$06	Buffer Pointer	Long	--->
$0A	Purge Flag	Byte	--->
$0B	Amount Transferred	Word	<---

The first time the call is executed, the firmware starts at the beginning of the buffer and reads the number of bytes specified in the parameter list.

◆ *Note:* Because LAP and DDP both read the beginning of the buffer, you do not need to worry about those headers. Similarly, ATP and other protocols remove their headers before you see the data.

Bytes read are placed into the destination buffer specified in the list. The firmware maintains a Location Pointer that is left pointing to the next byte after the last one that was read. The next time the call is executed, it begins with the byte to which the Location Pointer is pointing.

Request Count specifies how many bytes to transfer. If the Request Count is greater than the amount of data remaining, only the actual amount remaining is transferred, and an error is returned. The Amount Transferred field contains the size of the data (in bytes) actually transferred.

A non-zero value in the Purge Flag field causes the buffer to be purged after the transfer of data. When the buffer is purged, the packet data is no longer available. If the call is executed with the Purge Flag set and the Request Count is less than the amount remaining in the buffer, the buffer is purged and an error is returned, indicating that not all of the data has been transferred.

◆ *Note:* LAP protocol handlers and socket listeners must always execute this call with the Purge Flag set before completing; if they do not, the next packet will never be processed.

The ReadBuffer call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0201	No packet in buffer
$0202	End of buffer
$0209	Data lost in purge

AttachProt ($09)

The AttachProt call allows a LAP protocol handler to be attached. The parameter structure for the AttachProt call is listed below.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$09
$02	Result Code	Word	<---
$04	Protocol Type	Byte	--->
$05	Protocol Address	Long	--->

The Protocol Type and Protocol Address fields specify the LAP type and address of the handler. These values are installed in a LAP Type table. When a packet is received, the firmware's interrupt handler searches the LAP Type table and calls the appropriate protocol handler by using the address specified in this call. LAP Types may be in the range from $01 to $7F.

It is not necessary to install a Protocol Handler in order to *send* a LAP packet with a LAP Type that has not been installed. However, if a packet is *received* that contains a LAP type not found in the LAP type table, the packet is discarded. The firmware provides a catch-all type ($FF) to receive packets containing Lap types that are not installed. Using $FF as the Protocol Type in the AttachProt call causes all such packets to be sent to the routine specified in the Protocol Address field. The protocol handler terminates by doing a jump to the Completion Routine Return address that comes from the GetInfo call.

◆ *Note:* The machine states for entry and exit for a LAP handler are the same as for a completion routine.

The result codes returned for the AttachProt call are as follows:

Result Code	Description
$0205	Illegal LAP type
$0206	Duplicate LAP type
$0207	Too many protocols

RemoveProt ($0A)

The RemoveProt call removes the protocol handler of the type specified in the Protocol Type field. The parameter structure for the RemoveProt call is listed below.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$0A
$02	Result Code	Word	<---
$04	Protocol Type	Byte	--->

LAP Types that can be removed are in the range from $01 to $7F. The catch-all type ($FF) can also be removed.

The RemoveProt call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0205	Illegal LAP type
$0208	Type not found

Calls to the Datagram Delivery Protocol (DDP)

The Datagram Delivery Protocol (DDP) for the Apple IIGS firmware provides the standard AppleTalk Datagram Delivery Protocol. *Table 3-3* lists the calls to the DDP layer. The sections that follow describe each call, the parameter listing, and the result codes.

■ **Table 3-3** DDP calls

Command Number	Name	Description
$0B	OpenSocket	Open DDP socket
$0C	CloseSocket	Close DDP socket
$0D	SendDatagram	Send datagram

OpenSocket ($0B)

The OpenSocket call is used to open a DDP socket. The parameter structure for the OpenSocket call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$0B
$02	Result Code	Word	<---
$04	Socket Number	Byte	<-->
$05	Client Address	Long	--->

If the Socket Number field in the parameter list contains 0, a dynamic socket is opened and a socket number in the range of $80 to $FE is returned in the Socket Number field. If the Socket Number field is non-zero and is within the correct range for static sockets ($01 to $7F), DDP attempts to open it as a static socket. The Client Address is the address of the client's socket-listener routine and must be a valid address (not 0). This address is called by the firmware's interrupt handler when a packet is received for the socket being opened. Sockets 1, 2 and 4 are preinstalled and return an error if an attempt is made to open them without first closing the sockets.

◆ *Note:* Socket numbers $00 and $FF are not allowed.

The OpenSocket call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0301	Too many sockets open
$0303	Socket already open
$0304	Invalid socket type

CloseSocket ($0C)

The CloseSocket call provides the means to close a DDP socket. The parameter structure for the CloseSocket call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$0C
$02	Result Code	Word	<---
$04	Socket Number	Byte	--->

The Socket Number specifies the socket to be closed. The CloseSocket call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0302	Socket not open
$0304	Invalid socket type

SendDatagram ($0D)

The SendDatagram call is used to send a datagram. The parameter structure for the SendDatagram call is listed as follows:

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$0D
$02	Result Code	Word	<---
$04	Checksum Flag	Byte	--->
$05	Destination Network	<Word>	--->
$07	Destination Node	Byte	--->
$08	Destination Socket	Byte	--->
$09	Source Socket	Byte	--->
$0A	DDP Type	Byte	--->
$0B	BDS Pointer	Long	--->

The Checksum Flag applies only to internet packets. If the flag is 0, no checksum is calculated and an internet packet has 0 in the checksum field of the DDP header. If the flag is non-zero, the DDP checksum is calculated only if the packet is an internet packet and is included in the long DDP header. If the size of DDP data (that is, the sum of the BDS buffer lengths) is greater than 586, an error of $0305 is returned.

◆ *Note:* A DDP socket must be opened in order to send a datagram.

The BDS Pointer points to a Buffer Data Structure (BDS), which in turn points to the DDP data to be sent. The format of the BDS is shown in the following list.

Position	Name	Size	Value
$00	Reserved	6 Bytes	--->
$06	First Buffer Length	Word	--->
$08	First Buffer Pointer	Long	--->
$0C	Second Buffer Length	Word	--->
$0E	Second Buffer Pointer	Long	--->
		|	|
		v	v
$xx	Last Buffer Length	Word	--->
$xx	Last Buffer Pointer	Long	--->
$xx	End-of-BDS Flag	Word	$FFFF

The SendDatagram call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0302	Socket not open
$0304	Invalid socket type
$0305	DDP data too large

Calls to the Name Binding Protocol (NBP)

The Name Binding Protocol (NBP) for the Apple IIGS firmware provides the standard AppleTalk Name Binding Protocol. This section discusses calls to the NBP layer, listed in Table 3-4. The sections that follow describe each call, the parameter listing, and the result codes.

■ **Table 3-4** NBP calls

Command Number	Name	Description
$0E	RegisterName	Register name
$0F	RemoveName	Remove name
$10	LookupName	Lookup name
$11	ConfirmName	Confirm name
$46	NBPKill	Cancel Asynchronous NBP call

RegisterName ($0E)

The RegisterName call allows a name to be registered on the network. The parameter structure for the RegisterName call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$0E
$02	Result Code	Word	<---
$04	Completion Routine Ptr	Long	--->
$08	Name Structure Pointer	Long	--->
$0C	Retry Interval	Byte	--->
$0D	Retry Count	Byte	--->
$0E	Reserved	Word	x
$10	Socket Number	=Byte	--->
$11	Check Flag	Byte	--->

The Socket Number being registered is required in the parameter list, along with a pointer to a structure containing the entity name (Name Structure Pointer). The caller supplies the Retry Count and Retry Interval. The interval is in 1/4-second periods. The Check Flag field specifies whether the network and internal tables should be checked for a duplicate name. A value of 0 means the network should be checked. A non-zero value prevents the check from occurring.

The Name Structure Pointer field points to a structure like the one in the following list. This data structure must remain until the RemoveName call removes the name.

Position	Name	Size	Value
$00	Reserved	9 Bytes	x
$09	Entity Name (NBP Format)	Variable Length	--->

An entity name is a character string consisting of three fields—object, type, and zone. Each field consists of a leading 1-byte string length, followed by up to 32 string bytes. The string length represents the number of bytes in the string. The three strings are concatenated without any intervening padding for a total length of up to 99 bytes. (For more information, refer to *Inside AppleTalk*.)

The RegisterName call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0401	Too many names
$0402	Name already exists
$0406	Invalid name format
$0407	Incorrect address
$0408	Too many NBP processes

RemoveName ($0F)

The RemoveName call removes the name pointed to by the Entity Name Pointer. The parameter structure for the RemoveName call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$0F
$02	Result Code	Word	<---
$04	Entity Name Pointer	Long	--->

The Entity Name Pointer points *directly at the name, not to a name structure* as in the RegisterName command.

The RemoveName call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0403	Name not found
$0406	Invalid name format

LookupName ($10)

The LookupName call performs a name lookup on the network or internet. The parameter structure for the LookupName call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$10
$02	Result Code	Word	<---
$04	Completion Routine Ptr	Long	--->
$08	Entity Name Pointer	Long	--->
$0C	Retry Interval	Byte	--->
$0D	Retry Count	Byte	--->
$0E	Reserved	Word	x
$10	Buffer Length	Word	--->
$12	Buffer Pointer	Long	--->
$16	Max # of Matches	Byte	--->
$17	Actual # of Matches	Byte	<---

The caller must supply a pointer to the buffer where the matches are to be placed (Buffer Pointer) and the length of the buffer (Buffer Length). The structure of the data returned in the buffer is as follows:

Position	Name	Size		
$00	First Network #	<Word>	}	
$02	First Node #	Byte	}	First internet address
$03	First Socket #	Byte	}	
04	First Enumerator	Byte		(refer to *Inside AppleTalk*)
$05	First Entity Name	Variable Length		

The second and subsequent internet addresses observe the same structure. In the following example, *xx* is the address of the byte following the last byte of the previous Entity Name.

Position	Name
xx+00	Next Internet Address (which includes the Network #, Node #, and Socket # of the next Internet)
+04	Next Enumerator Byte
+05	Next Entity Name

It is up to the caller to supply the values for the Max # of Matches, Retry Count, and Retry Interval fields. The Retry Interval is in 1/4-second periods. The value for the Actual # of Matches will be returned in the parameter list when the call completes.

The LookupName call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0401	Too many names
$0404	User's buffer full
$0406	Invalid name format
$0408	Too many NBP processes

ConfirmName ($11)

The ConfirmName call requires the name and address being confirmed. The parameter structure for the ConfirmName call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$11
$02	Result Code	Word	<---
$04	Completion Routine Ptr	Long	--->
$08	Entity Name Pointer	Long	--->
$0C	Retry Interval	Byte	--->
$0D	Retry Count	Byte	--->
$0E	Reserved	Word	x
$10	Network Number	<Word>	--->
$12	Node Number	Byte	--->
$13	Socket Number	Byte	--->
$14	Actual Socket Number	Byte	<---

The Retry Count and Retry Interval are required. The Retry Interval is in 1/4 second periods. The Actual Socket Number field returns the actual socket number found for the name. The ConfirmName call will not confirm a name in the caller's own node.

The ConfirmName call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0403	Name Not Found
$0406	nvalid name format
$0407	ncorrect address
$0408	Too many NBP processes

NBPKill ($46)

The NBPKill call is used to cancel an asynchronous NBP call before it completes. The parameter structure for the NBPKill call list listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$46
$02	Result Code	Word	<---
$04	ParamBlockPointer	Long	--->

ParamBlockPointer must point to the beginning of the parameter block that is currently being used by the asynchronous call that is to be canceled.

The cancelled NBP call will complete with error $0409.

The NBPKill call returns these result codes, as well as the result codes for all system calls.

Result	Description
$040A	ParamBlock Not Found

Calls to the AppleTalk Transaction Protocol (ATP)

The AppleTalk Transaction Protocol (ATP) for the Apple IIGS firmware provides full implementation of the ATP protocol, allowing use of all of the features specified in the AppleTalk ATP specification. Table 3-5 lists calls to the ATP layer; the following section describes each of these calls, and gives the parameter list and result codes for each call.

■ **Table 3-5** ATP calls

Command Number	Name	Description
$12	SendATPReq	Send ATP request
$13	CancelATPReq	Cancel ATP request
$14	OpenATPSocket	Open ATP responding socket
$15	CloseATPSocket	Close ATP responding socket
$16	GetATPReq	Get ATP request
$17	SendATPResp	Send ATP response
$18	AddATPResp	Add ATP response
$19	RelATPCB	Release responding control block

SendATPReq ($12)

The SendATPReq call is used to send an ATP request. The parameter structure for the SendATPReq call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$12
$02	Result Code	Word	<---
$04	Completion Routine Ptr	Long	--->
$08	Dynamic Socket Number	Byte	<---
$09	Destination Network	<Word>	--->
$0B	Destination Node	Byte	--->
$0C	Destination Socket	Byte	--->
$0D	ATP Transaction ID	<Word>	<---
$0F	Request Buffer Length	Word	--->
$11	Request Buffer Pointer	Long	--->
$15	User Bytes	4 Bytes	--->
$19	# of Response Buffers	Byte	--->
$1A	Response BDS pointer	Long	--->
$1E	ATP Flags	Byte	--->
$1F	Retry Interval	Byte	--->
$20	Retry Count	Byte	--->
$21	Current Bitmap	Byte	<--->
$22	# of Responses Received	Byte	<---
$23	Reserved	6 Bytes	x

The Dynamic Socket Number indicates the DDP socket that ATP is using. The Response BDS Pointer points to a Buffer Data Structure (BDS) for the response packets. Bit 5 in the ATP Flags field may be set to indicate an exactly-once transaction. The other bits are not used in this call. The caller must supply the values for the Retry Count and Retry Interval fields. The retry interval is in 1/4-second periods.

The User Bytes field specifies the ATP User Bytes to be sent in the request packet. You must set the Current Bitmap field with the proper value for the number of packets you're requesting. For instance, if you request packets 1 through 4, set the bitmap to $0F.

While the call is executing, the Result Code field contains $FF in the low byte. The ATP Transaction ID (TID) field is valid, and the Current Bitmap and # of Responses Received fields are updated throughout call execution. When the call completes, the # of Responses Received field indicates how many packets were received, and the Current Bitmap field indicates which packets were received.

The format of the Response BDS is as follows.

Position	Name	Size	Value
$00	1st Buffer Length	Word	--->
$02	1st Buffer Pointer	Long	--->
$06	1st Buffer User Bytes	4 Bytes	<---
$0A	1st Buffer Actual Length	Word	<---
$0C	2nd Buffer Length	Word	--->
$0E	2nd Buffer Pointer	Long	--->
$12	2nd Buffer User Bytes	4 Bytes	<---
$16	2nd Buffer Actual Length	Word	<---
		|	|
		v	v
$xx	Last Buffer Length	Word	--->
$xx	Last Buffer Pointer	Long	--->
$xx	Last Buffer User Bytes	4 Bytes	<---
$xx	Last Buffer Actual Length	Word	<---

Bits 0-14 of the Actual Length field contains the length of the data received for the buffer with which it is associated. If the data length is larger than the buffer supplied, the high bit of the Actual Length field is set to indicate the overflow, and bits 0 to 14 contains the length of the data actually transferred to the buffer. The User Bytes field contains the ATP User Bytes returned with the packet that was placed into the buffer.

The result codes returned for the SendATPReq call are listed here.

Result Code	Description
$0501	ATP data too large
$0504	Too many active ATP calls
$0507	ATP send request aborted
$0508	ATP send request failed, retry exceeded
$050B	Too many responses expected
$050C	Unable to open DDP socket

CancelATPReq ($13)

The CancelATPReq call is used to cancel an outstanding ATP request. The parameter structure of the CancelATPReq call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$13
$02	Result Code	Word	<---
$04	ATP Transaction ID	<Word>	--->

ATP Transaction ID must contain the identification of a request that is currently executing. The CancelATPReq call returns this result code, as well as the result codes for all system calls.

Result Code	Description
$0503	ATP control block not found

OpenATPSocket ($14)

The OpenATPReq call is used to open an ATP responding socket. The parameter structure of the OpenATPReq call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$14
$02	Result Code	Word	<---
$04	Socket Number	Byte	<-->

If the Socket Number field in the parameter list contains 0, a dynamic socket is opened and the number returned in the Socket Number field. If a socket number is supplied by the caller, it must be within the correct range for static sockets.

The OpenATPSocket call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$050A	Too many ATP sockets
$050C	Unable to open DDP socket

CloseATPSocket ($15)

The CloseATPSocket call closes the specified ATP responding socket. The parameter structure of the CloseATPSocket call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$15
$02	Result Code	Word	<---
$04	Socket Number	Byte	--->

The CloseATPSocket call returns this result code, as well as the result codes for all system calls.

Result Code	Description
$0502	Invalid ATP socket

GetATPReq ($16)

The GetATPReq call prepares the specified socket to receive a request. More than one GetATPReq may be outstanding on a socket. The parameter structure of the GetATPReq call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$16
$02	Result Code	Word	<---
$04	Completion Routine Ptr	Long	--->
$08	Response Socket #	Byte	--->
$09	Source Network Number	<Word>	<---
$0B	Source Node Number	Byte	<---
$0C	Source Socket Number	Byte	<---
$0D	ATP Transaction ID	<Word>	<---
$0F	Request Buffer Length	Word	--->
11	Request Buffer Pointer	Long	--->
$15	User Bytes	4 Bytes	<---
$19	Actual Request Length	Word	<---
$1B	ATP Flags	Byte	<---
$1C	Bitmap	Byte	<---
$1D	Reserved	Long	x

When the call completes, the parameter list contains the header data from the request. Bit 5 of the ATP Flags field specifies whether or not the request is an exactly-once transaction.

▲ **Warning** There is a 30-second time-out. With no further requests of the same TID after 30 seconds, the memory that was allocated for the Control Block is released (that is, memory that was created for this transaction when a request is received).

The GetATPReq call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0502	Invalid ATP socket
$0504	Too many active ATP calls. This result could will never be returned
$0509	Async call aborted, socket was closed

SendATPResp ($17)

The SendATPResp call is used to send a response to a received request.

◆ *Note:* This call does not complete until all packets have been sent and acknowledged.

The parameter structure for the SendATPResp call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$17
$02	Result Code	Word	<---
$04	Completion Routine Ptr	Long	--->
$08	Response Socket #	Byte	--->
$09	Dest Network Number	<Word>	--->
$0B	Dest Node Number	Byte	--->
$0C	Dest Socket Number	Byte	--->
$0D	ATP Transaction ID	<Word>	--->
$0F	# of Response Buffers	Byte	--->
$10	Total ATP Packets	Byte	--->
$11	Response BDS Pointer	Long	--->
$15	ATP Flags	Byte	--->
$16	Current Bitmap	Byte	<-->
$17	Add Routine Pointer	Long	--->
$1B	Add Bitmap	Byte	<---

◆ *Note:* It is the responsibility of the user to set the bitmap as indicated in *Inside AppleTalk* for the number of packets being sent before the user makes the call.

The Apple IIGS firmware must use the correct Response Socket # (obtained from an OpenATPSocket call), as well as the Destination Address and ATP Transaction ID (obtained from the GetATPReq call). The Total ATP Packets field tells ATP the total number of packets to be sent in the response.

The value in the # of Response Buffers field specifies how many of the Total ATP Packets are being sent with this call, and must match the number of packets you are sending.

The Response BDS field must contain room for the number of buffers specified in the Total ATP Packets field. The format of the Response BDS is the same as that for SendATPReq; however, the Actual Length field is not used, and the User Bytes are supplied by the caller rather than returned by the call. If any of the BDS buffers are too small to contain all the data in the packet assigned to it, the extra data for that packet only is lost, and no error is returned.

You must supply a 0 in the Add Routine Pointer and Add Bitmap fields. These fields are not currently used for the Apple IIGS workstation.

The SendATPResp call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0501	ATP data too large
$0502	Invalid ATP socket
$0503	ATP control block not found
$0505	No release received
$0509	Async call aborted, socket was closed
$050B	Too many responses expected
$050D	ATP Send Response was released

AddATPResp ($18)

This call is currently not implemented.

RelATPCB ($19)

The RelATPCB call is used by the responding node to release the control block and all buffers associated with a response to an exactly-once transaction. The control block and all held buffers are released. Parameter usage for the RelATPCB call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$19
$02	Result Code	Word	<---
$04	Response Socket #	Byte	--->
$05	Dest Network Number	<Word>	--->
$07	Dest Node Number	Byte	--->
$08	Dest Socket Number	Byte	--->
$09	ATP Transaction ID	<Word>	--->

The RelATPCB call returns this result code, as well as the result codes for all system calls.

Result Code	Description
$0503	ATP control block not found

Calls to the Zone Information Protocol (ZIP)

This section describes calls to the Zone Information Protocol (ZIP) layer on the Apple IIGS that may be needed for normal workstation use communicating with and through bridges. The other ZIP calls (described in *Inside AppleTalk*), not supported on the Apple IIGS, are used for taking down and bringing up a bridge, and would be part of special utilities used for such purposes.

Table 3-6 lists the ZIP calls supported. The sections that follow describe each call, the parameter listing, and the result codes.

■ **Table 3-6** ZIP calls

Command Number	Name	Description
$1A	GetMyZone	Get zone name for my zone
$1B	GetZoneList	List zones of all networks

GetMyZone ($1A)

The GetMyZone call returns the zone name of the network that the workstation is on. The parameter structure for the GetMyZone call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$1A
$02	Result Code	Word	<---
$04	Completion Routine Ptr	Long	--->
$08	Buffer Pointer	Long	--->
$0C	Retry Interval	Byte	--->
$0D	Retry Count	Byte	--->
$0E	Reserved	Word	x

This call assumes that the buffer is at least 33 bytes long. The Retry Interval is in 1/4-second periods. The GetMyZone call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0601	Network error
$0603	ZIP not found

GetZoneList ($1B)

The GetZoneList call returns the complete list of zones on the internet. The parameter structure of the GetZoneList call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$1B
$02	Result Code	Word	<---
$04	Completion Routine Ptr	Long	--->
$08	Buffer Length	Word	--->
$0A	Buffer Pointer	Long	--->
$0E	Bridge Node Number	Byte	--->
$0F	Start Index	Word	--->
$11	Retry Interval	Byte	--->
$12	Retry Count	Byte	--->
$13	# Zones Returned	Word	<---
$15	Reserved	Long	--->

The Bridge Node Number field specifies the bridge to which the call is to be directed. This number can be obtained through the GetInfo call. The user must provide the Bridge Node Number field in this call because more than one execution of the call may be required to get the complete zone list. Each execution of the call must be directed to the same bridge. The internal field A-Bridge on the Apple IIGS changes periodically if there is more than one bridge on the local network; therefore, the user must supply the node number of the bridge to which the call should be directed.

The Buffer Pointer points to a buffer where the list is to be placed. The buffer will be filled with as many zone names as will fit, starting with the entry specified by Start Index.

◆ *Note:* Start Index must start with 1; it cannot use 0.

The value in the # Zones Returned field indicates how many zone names have been returned in the buffer. The Retry Interval is in 1/4-second periods.

The GetZoneList call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0601	Network error
$0602	ZIP overflow
$0603	ZIP not found

Calls to the AppleTalk Session Protocol (ASP)

This section describes calls to the AppleTalk Session Protocol (ASP) layer on the Apple IIGS workstation. These ASP calls follow the ASP specification very closely.

Table 3-7 lists the ASP calls used by the Apple IIGS. The sections that follow provide a brief description of each call, as well as the parameter listing and the result codes for each.

■ **Table 3-7** ASP calls

Command number	Name	Description
$1C	SPGetParms	Get parameters
$1D	SPGetStatus	Get status
$1E	SPOpenSession	Open a session
$1F	SPCloseSession	Close a session
$20	SPCommand	Send a command
$21	SPWrite	Write multiple packets

The SPGetParms call returns implementation-dependent information regarding the allowable sizes of buffers. The maximum size of a command block and the maximum size of a reply or of write data are returned in the fields Max Command Size and Max Data Size.

The parameter structure for the SPGetParms call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 Synchronous Only
$01	Command	Byte	$1C
$02	Result Code	Word	<---
$04	Max Command Size	Word	<---
$06	Max Data Size	Word	<---

The result codes returned for the SPGetParms call are the same as those for all system calls.

SPGetStatus ($1D)

The SPGetStatus call is used by the workstation to obtain the current status of a known server. The parameter structure for the SPGetStatus call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$1D
$02	Result Code	Word	<---
$04	Completion Routine Ptr	Long	--->
$08	SLS Network Number	<Word>	--->
$0A	SLS Node	Byte	--->
$0B	SLS Socket	Byte	--->
$0C	Buffer Length	Word	--->
$0E	Buffer Address	Long	--->
$12	Length of Status Data	Word	<---

The Buffer Address field points to the status buffer, and the Length of Status Data field indicates its size. The SPGetStatus call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0701	Network error
$0702	Too many ASP calls
$0704	Size error.
$0706	No response from server

SPOpenSession ($1E)

The SPOpenSession call is used by the workstation to open a session with a known server. The parameter structure for the SPOpenSession call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$1E
$02	Result Code	Word	<---
$04	Completion Routine Ptr	Long	--->
$08	SLS Network Number	<Word>	--->
$0A	SLS Node	Byte	--->
$0B	SLS Socket	Byte	--->
$0C	Attention Routine Addr	Long	--->
$10	Session Reference #	Byte	<---

The Attention Routine Addr points to a routine to be called when an attention packet is received. The 2 bytes received in the attention packet are placed in the second 2 bytes of the address pointed to by the Attention Routine Addr. The Apple IIGS then jumps to the address 4 bytes beyond the specified address.

The 4 bytes pointed to by Attention Routine Addr include the following:

Position	Name	Size	Value
0	Session Reference #		<---
1	Atten Type		<---
2	Atten	Word	<--- (from server)

The Atten Type includes

$00 Normal

$40 Connection timeout

$80 Connection closed by server

The SPOpenSession call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0701	Network error
$0702	Too many ASP calls
$0706	No response from server
$0707	Bad version number
$0708	Too many sessions
$0709	Server busy

SPCloseSession ($1F)

The SPCloseSession call terminates an open session. The parameter structure for the SPCloseSession call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$1F
$02	Result Code	Word	<---
$04	Completion Routine Ptr	Long	--->
$08	Session Reference #	Byte	--->

The SPCloseSession call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0701	Network error
$0703	Invalid reference number

SPCommand ($20)

The SPCommand call is used by the Apple IIGS workstation on an open session to send a Command packet to a server. The parameter structure for the SPCommand call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$20
$02	Result Code	Word	<---
$04	Completion Routine Ptr	Long	--->
$08	Session Reference #	Byte	--->
$09	Command Block Length	Word	--->
$0B	Command Block Address	Long	--->
$0F	Reply Buffer Length	Word	--->
$11	Reply Buffer Address	Long	--->
$15	Command Result	Long	<---
$19	Reply Length	Word	<---

The inputs are the session reference number, a command block buffer, and a reply buffer. The Command Block Address field gives the address of the command block to be sent, and the Command Block Length field gives the number of bytes to be sent. The number of bytes sent cannot be greater than the maximum command size; otherwise, a size error is returned in the result code and no attempt made to send anything out over the network. The Reply Buffer Address field points to the reply buffer, and the value in the Reply Buffer Length field indicates its size.

Upon successful completion, the number of bytes of reply data returned is in the updated Reply Length field, and the Command Result field will have a 4-byte value provided by the server.

The SPCommand call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0701	Network error
$0702	Too many ASP calls
$0703	Invalid reference number
$0704	Size error
$0706	No response from server
$070A	Session closed

The system now uses SPCommand calls asynchronously. Applications that have AppleShare volumes mounted under System Software 5.0 and also make SPCommand calls themselves should now handle the "Too many ASP calls" error, $0702.

AppleShare uses a protocol called AppleTalk Session Protocol (ASP) to maintain a connection (session) with all servers that you are logged on to. All commands and data transfer to the server are sent using ASP.

The implementation of ASP on the Apple IIGS has a limit of one command outstanding (waiting to complete) per session. This means that if one command has been sent, its reply must be received before you can send the next command. Remember, the SPCommand call is used to send commands over a session. If you try to issue an SPCommand before another (asynchronous) SPCommand on the same session has completed, your call will return with a "Too many ASP calls" error, $0702.

Before System Software 5.0 on the Apple IIGS, no system software made asynchronous SPCommand calls, and therefore this error would only occur if the developer was making the asynchronous calls. As of System Software 5.0, the AppleShare FST uses asynchronous calls to help prevent the loss of a connection with servers and to assist the Finder in dynamically updating windows when a change is made to a network volume. Therefore, this error may be returned even though the developer is not making asynchronous calls.

The error is easy to handle if you are making synchronous SPCommand calls. Simply make the call, and if it completes with error $0702, loop back and make the call again until you can do so without error $0702. This technique forces your program to wait until ASP is free again to make the call.

If you are making asynchronous SPCommand calls, and you receive the $0702 error, you might want to install a short (i.e., 1/4 second) timer using the InstallTimer call, and make the SPCommand call again when the timer completes. Remember, the InstallTimer has to be asynchronous, since you are making it from the completion routine of an asynchronous call.

◆ *Note:* When using the AppleShare FST under GS/OS, there is little reason to make SPCommand calls yourself, since most of the calls you can make are available through the FST as normal file system calls or as FST-specific calls.

SPWrite ($21)

The SPWrite call is used by the Apple IIGS workstation during an open session to send a write packet to the server. The parameter structure for the SPWrite call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$21
$02	Result Code	Word	<---
$04	Completion Routine Ptr	Long	--->
$08	Session Reference #	Byte	--->
$09	Command Block Length	Word	--->
$0B	Command Block Address	Long	--->
$0F	Write Data Length	Word	--->
$11	Write Data Address	Long	--->
$15	Reply Buffer Length	Word	--->
$17	Reply Buffer Address	Long	--->
$1B	Command Result	Long	<---
$1F	Written Length	Word	<---
$21	Reply Length	Word	<---

The inputs are the same as for the SPCommand command described earlier, with the addition of the following: the Write Data Address field should contain the address of the data to be sent to the server, and the Write Data Length field gives the number of bytes.

The outputs are the same as the ASPCommand command above, with the addition of the following. Upon completion, the Written Length field will contain the number of bytes successfully sent to the server.

The SPWrite call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0701	Network error
$0702	Too many ASP calls
$0703	Invalid reference #
$0704	Size error
$0705	Buffer error
$0706	No response from server
$070A	Session closed

The SPWrite call also has a limit of one outstanding call per session. System software does not currently use asynchronous SPWrite calls, but looping until ASP returns something other than $0702 would be a good precaution for SPWrite, too.

◆ *Note:* When using the AppleShare FST under GS/OS, there is little reason to make SPWrite calls yourself, since packets can be sent through the FST as normal file system calls or as FST-specific calls.

Calls to the AppleTalk Filing Protocol (AFP)

To provide transparent access to AFP servers for the operating system, firmware on the Apple II workstation translates ProDOS 8 or GS/OS filing calls into a subset of AFP calls. The code that performs this, under ProDOS 8, is the ProDOS Filing Interface (PFI). Most AFP calls, under ProDOS 8, are supported indirectly through ProDOS MLI calls to the PFI. (For additional information, refer to the AFP specifications and "Calls to the ProDOS Filing Interface" given later in this chapter.)

Under GS/OS, the functionality of almost all AFP calls can be achieved using standard GS/OS calls or FST-Specific calls to the AppleShare FST.

Features of AFP that are not available through operating system calls can be accessed by making AFP calls directly through ASP. The following example demonstrates how to make AFP call GetVolParm (not available through ProDOS 8 and PFI) through AFP.

▲ Warning NEVER use this method to make AFP calls that will affect open files. If you do so, internal structures in the AppleShare FST and/or PFI may become corrupted and data loss may result. Always use the equivalent calls under the AppleShare FST or PFI. Under System Software 5.0 and later, you should use the FIGetSVersion call to determine the AFP version being used on that session before constructing an AFP packet. ▲

```
            longa       off
            longi       off
            absaddr     on
            65c02       on
            verbose     on
            Keep   AFPExample

AFPExample  Start
*****************************************
* AFPExample demonstrates how to make      *
* AFP calls using the ASP Protocol.   *
*****************************************

mli         equ    $BF00
AtCall equ  $42
ReplyBuff   equ    $800

            jsr    mli
            dc     i1'ATCall'
            dc     a'FIListSess' ;get the list of current
                                 ;sessions
            ldx    SessNum
            beq    NoSessions
            lda    ReplyBuff     ;use the first volume in the
                                 ;list as an example
            sta    SPCommand+8   ;get and save the Session No.
            lda    ReplyBuff+30  ;and the Volume ID
```

```
                sta     AFPPacket+2             ;
                lda     ReplyBuff+31            ;
                sta     AFPPacket+3             ;
                jsr     mli                     ;make the SPCommand Call
                dc      i1'ATCall'
                dc      a'SPCommand'
                bcc     NoError
                jmp     ErrorHandler
NoError         nop                             ;The result is in ReplyBuff as
                nop                             ;specified in Inside AppleTalk
                rts

NoSessions      nop                             ;ErrorHandling routines to
ErrorHandler    nop                             ;take care of unfortunate
                                                ;mishaps
                rts

SPCommand       anop
                dc      h'00'                   ;byte - sync flag
                dc      h'20'                   ;byte - Command
                dc      i'0'                    ;word - Result Code
                dc      i4'0'                   ;long - Completion Routine
                                                ;Pointer
                ds      1                       ;byte - Session Reference
                                        ;Number
                dc      i'6'                    ;word - Command Block Length
                dc      i4'AFPPacket'   ;long - Command Block Pointer
                dc      i'512'          ;word - Reply Buffer Length
                dc      i4'ReplyBuff'   ;long - Reply Buffer Address
                ds      4                       ;long - Command Result <-
                ds      2                       ;word - Reply Length <-

AFPPacket       anop
                dc      i1'17'          ;AFP Command for FPGetVolParms
                dc      h'00'                   ;Reserved
                ds      2                       ;Volume ID
                dc      h'0048'                 ;Bitmap to return Bytes Free
                                                ;and Modified Date

FIListSess      anop
                dc      h'00'                   ;byte - sync flag
                dc      h'2f'                   ;byte - Command
                dc      i'0'                    ;word - Result Code
                dc      i'512'          ;word - Reply Buffer Length
                dc      i4'ReplyBuff'   ;long - Reply Buffer Pointer
SessNum         ds      1                       ;byte - Number of Entries
                                        ;Returned
                end
```

Calls to the Printer Access Protocol (PAP)

It is not necessary to use the Printer Access Protocol (PAP) to print in a normal mode; the SSC entry point and Remote Print Manager (RPM) provide transparent ProDOS printing. However, PAP may be called directly for special purposes. Using PAP allows more control and flexibility, such as when you want to use the Chooser to communicate with a LaserWriter to see if it has an ImageWriter emulator installed.

This section describes calls to the PAP layer on the Apple IIGS, which provides a full implementation of the workstation side of PAP. These calls can be used for network printing, including spooling to servers that provide such capabilities.

◆ *Note:* If the For this section only, the Session Reference # field is called the Connection Reference # field to be compatible with the section on PAP in *Inside AppleTalk*.

Table 3-8 lists the calls to the PAP layer. The sections the follow provide a description of each call, as well as the parameter listing and the result codes for each.

■ **Table 3-8** PAP calls

Command number	Name	Description
$22	PAPStatus	Get server status
$23	PAPOpen	Open PAP session
$24	PAPClose	Close PAP session
$25	PAPRead	PAP read
$26	PAPWrite	PAP write
$27	PAPUnload	PAP unload

PAPStatus ($22)

The PAPStatus call is used to check the current status of a printer on the network. It is not necessary to be connected. You can use PAPStatus without using PAPOpen first, even if someone else is printing.

The parameter structure for the PAPStatus call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$22
$02	Result Code	Word	<---
$04	Completion Routine Ptr	Long	--->
$08	Printer Name Pointer	Long	--->
$0C	Status Buffer Pointer	Long	--->

The Printer Name Pointer field points to the NBP Entity Name. The status buffer must be at least 260 bytes. The result codes returned for the PAPStatus call are listed here.

Result Code	Description
$0804	Too many commands
$0805	Name not found
$0807	Network error
$0808	Server not responding
$080B	PAP in use

PAPOpen ($23)

The PAPOpen call is made by a workstation to open a connection with a printer (or any PAP server) on the network. On the open connection, the workstation may send data to the printer (via PAPWrite) or read data from the printer (via PAPRead).

◆ *Note:* If the target printer (or the PAP server) already has its maximum number of connections opened, then this PAPOpen does not complete until one of the existing connections is released (closed). You should always set the flow quantum to 1; the implementation on the Apple IIGS allows only 1 quantum from the other side.

The parameter structure for the PAPOpen call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$23
$02	Result Code	Word	<---
$04	Completion Routine Ptr	Long	--->
$08	Connection Reference #	Byte	<---
$09	Printer Name Pointer	Long	--->
$0D	Flow Quantum	Byte	<-->
$0E	Status Buffer Pointer	Long	--->

The buffer pointed to by Status Buffer Pointer should be at least 260 bytes.

The result codes returned for the PAPOpen call are as follows.

Result Code	Description
$0801	Too many sessions
$0803	Quantum error
$0804	Too many commands
$0805	Name not found
$0807	Network error

▲ **Warning** PAPOpen does not return a "Server Busy" error. It is up to a high-level routine to monitor the Result Code and keep track of time-out errors. If you want to time-out, this call should be made asynchronously. ▲

PAPClose ($24)

The PAPClose call is used by a workstation to close an open connection with a printer. The parameter structure for the PAPClose call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$24
$02	Result Code	Word	<---
$04	Completion Routine Ptr	Long	--->
$08	Session Reference #	Byte	--->

The result code returned for the PAPClose call is as follows.

Result Code	Description
$0802	Invalid reference number

PAPRead ($25)

The PAPRead call is used by a workstation to read data on a PAP connection (sent by the other end, either a printer or a server, using a PAPWrite). The Buffer Pointer should be at least 512 bytes times the flow quantum returned. The parameter structure for the PAPRead call is listed below.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$25
$02	Result Code	Word	<---
$04	Completion Routine Ptr	Long	--->
$08	Session Reference #	Byte	--->
$09	Buffer Length	Word	<---
$0B	Buffer Pointer	Long	--->
$0F	End-of-File Flag	Byte	<---

A non-zero value is returned to signal the end of file (note that both the LaserWriter and ImageWriter uses a value of non-zero).

The result codes returned for the PAPRead call are as follows.

Result Code	Description
$0802	Invalid reference number
$0804	Too many commands
$0806	Session closed
$0807	Network error$080B PAP in use

PAPWrite ($26)

The PAPWrite call is made by a workstation to write data on a PAP connection (that is read by the other end, either a printer or a server, using a PAPRead). Currently, the PAP implementation on the Apple IIGS does not allow you to write more than 512 bytes per call.

The parameter structure for the PAPWrite call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$26
$02	Result Code	Word	<---
$04	Completion Routine Ptr	Long	--->
$08	Connection Reference #	Byte	--->
$09	Data Length	Word	--->
$0B	Buffer Pointer	Long	--->
$0F	End-of-File Flag	Byte	$01

The result codes returned for the PAPWrite call are listed here.

Result Code	Description
$0802	Invalid reference number
$0804	Too many commands
$0806	Session closed
$0807	Network error
$080A	Buffer size error
$080B	PAP in use

PAPUnload ($27)

The PAPUnload call is used to close all connection with a server, whereas PAPClose closes only one connection. Parameter usage for the PAPUnload call is listed below.

Position	Name	Size	Value
$00	Async Flag	Byte	--->
$01	Command	Byte	$27
$02	Result Code	Word	<---
$04	Completion Routine Ptr	Long	--->

Calls to the Remote Print Manager (RPM) interface

The Apple II workstation firmware contains interface software called the Remote Print Manager (RPM). RPM allows transparent printing to remote printers on an AppleTalk network through the Super Serial Card (SSC) entry points and commands. RPM information is stored in the ATINIT file and is restored at boot time; it is not necessary to use the Chooser at every boot time.

The name of the printer to which output is to be directed is set by using the PMSetPrinter call. RPM uses PAP to send the data to be printed; the data to be printed is transferred to the firmware one character at a time through the SSC entry point. The characters are blocked, put into packets, then sent to the printer specified in the PMSetPrinter call.

Two time-outs are maintained by RPM. One time-out is used to flush the current block when there are small delays in the character stream being sent to RPM, as when a user is typing on the keyboard (usually 1/4 second). The second time-out is longer (usually 30 seconds), and indicates the end of the report. When this time-out expires, the last block is sent and the PAP connection closed. The maximum value for time-out is $1FFF for Apple IIe workstation compatibility.

Pascal Protocol Serial STATUS call returns incorrect results. When using the Workstation card, the Pascal STATUS call (normally used for printing) does not properly indicate whether the card is ready to receive characters. Applications should avoid this call, as the Pascal WRITE call in the firmware will perform this function automatically.

Table 3-9 lists the call to the Remote Print Manager (RPM) on the Apple IIGS workstation. The section that follows provides a description of the call, as well as the parameter list and the result codes.

From GS/OS, you should always use the operating system drivers. In System 5.0, the ".RPM" driver is used. In System 4.0, a generated driver using the RPM slot is used.

■ **Table 3-9** Calls to RPM

Command number	Name	Description
$28	PMSetPrinter	Set default printer
$47	PMCloseSession	Close an RPM session

PMSetPrinter ($28)

The PMSetPrinter call is used to determine where printed output is to be directed. The parameter structure for the PMSetPrinter call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$28
$02	Result Code	Word	<---
$04	Entity Name Pointer	Long	<--->
$08	Flags	Byte	<--->
$09	Flush Interval	Word	<--->
$0B	Timeout Interval	Word	<--->
$0D	Number of Buffers	Word	<--->

◆ *Note:* For the Number of Buffers field, you must use a value of at least 1 (of size 512 bytes). Normally this call allocates buffers for you and sets this parameter to 20. The more buffers you set, the faster the call will be (the tradeoff is the amount of memory used).

The name pointed to by the Entity Name Pointer field specifies the name of a network printer for RPM, and must be in standard NBP format. No attempt is made to verify that the given name exists on the network. The name is used only when the Flags field specifies a network printer. The Flags field contains the flags listed in Table 3-10.

■ **Table 3-10** Printer name flags

Value	Description
Bit 7	Network printer
Bit 6	Not used in the Apple IIGS
Bit 5	Postscript emulator
Bit 4	Reserved
Bit 3	Reserved
Bit 2	Reserved
Bit 1	Reserved
Bit 0	Return selected printer

When set, the Network Printer bit informs the firmware that the desired printer is on the network; its name is pointed to by the Entity Name Pointer. If the Postscript Emulator bit is set in conjunction with the Network Printer bit, RPM sends out the following code at the beginning of the first packet sent to the printer; this turns on the ImageWriter emulator in the LaserWriter.

```
DC C'%%IncludeProcSet IWEm 1 1'
DC H'0d'
DC C'_WBJ_ '
```

Bit 0 of the Flags field specifies that the call should return the name of the printer if it is set for RPM. If bit 0 is set when the call is made, all of the other bits must be clear. When the call is executed in this manner, the call completes with the proper bits set in the Flags field and the printer name is placed at the address specified by the Entity Name Pointer field.

▲ **Warning** When PMSetPrinter is called with Flags field bit 0 set, the buffer pointed to by the Entity Name Pointer field must be **100 bytes** long. ▲

The Flush Interval and Timeout Interval fields allow the caller to set the time values for these time-outs. They are both specified in 1/4-second increments; the Timeout Interval must be greater than the Flush Interval. If the Timeout Interval is set to zero (0), then the session will never time out and must be stopped via the PMCloseSession call.

The PMSetPrinter call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0901	Invalid flag byte
$0902	Invalid time values

PMCloseSession ($47)

The PMCloseSession call is used to close any outstanding RPM session. The parameter structure for the PMCloseSession call is listed here

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$28
$02	Result Code	Word	<---

The PMCloseSession call never returns an error.

ProDOS 8 AFP Translator

This section describes the differences between making ProDOS 8 calls and using the AFP Translator to make ProDOS 8 calls over the network. The AFP Translator code captures all ProDOS 8 MLI calls. Depending on the volume being accessed, the AFP Translator either routes the calls to the local ProDOS in the Apple II, or translates them into AFP calls and sends them over the network to an AFP file server. The AppleShare File System Translator performs the equivalent task under GS/OS.

Catalogs under ProDOS 8 are performed by opening and reading directories. However, AFP does not support reading directories directly. Instead, the AFP Translator must execute an AFP Enumerate call and create a "fake" directory for ProDOS 8 based on the information returned by AFP. When names are returned that contain invalid characters, the invalid characters are replaced by the question-mark character (?) in the fake directory block that is created. Also, when the name returned is longer than 15 characters, a question-mark character (?) is placed in the last position of the name to indicate that the name is actually longer than what has been returned. GS/OS applications should always use GetDirEntry to read directories.

ProDOS 8 AFP Translator Access Mode

Files on the file server may be opened and read by more than one workstation on the network. Only the first workstation to open the file is granted write access to a file (subsequent workstations have read-only access). If a second workstation tries to open the file and attempts a Write operation, the error Access Error ($4E) is returned.

If a file is open on more than one workstation and the workstation with the first open/write access closes the file, the access paths being used by the other workstations that have opened the file are not closed but retain their read-only access. The next workstation to open the file receives write access. If the files are opened with the Special Open Fork call, users can share write access.

Resource forks

Two forks are created for every file in an AFP 2.0 file server: a data fork and a resource fork. These forks are similar to the Macintosh Hierarchical File System, which has a Resource Manager and uses both forks. All normal ProDOS 8 calls are defaulted to the data fork by the AFP Translator.

- *Note:* Unless your application is smart enough to know how to manipulate and maintain resource maps in the resource fork, the application should only use the data fork for its data storage. If you want to create a multi-user or multi-launch version of your application, you cannot use resource files to store your data.

An applications programmer can access information from the resource fork by setting certain bits in the parameter count of related calls. The two ProDOS 8 calls that are affected are *GetFileInfo* and *Open*.

Differences in ProDOS 8 and AFP Translator Calls

There are specific differences in the way in which ProDOS calls are translated. This section describes the specific parameters affected for the following for ProDOS 8 commands. Refer to Chapter 4 in the *ProDOS 8 Technical Reference Manual* for detailed information on these calls.

- GetFileInfo (ProDOS 8 Command $C4)
- Open (ProDOS 8 Command $C8)

See the following section for specifics of these calls under GS/OS.

GetFileInfo

Differences in the way in which the GetFileInfo call is used for files and directories are described next.

For Files: The parameters that are affected by the selection of the data fork, the resource fork, or both, are the size of the file and the number of blocks that it uses. *Table 3-11* lists these parameters.

- **Table 3-11** File parameters for GetFileInfo command

To see	Set PCount to
The size of the data fork	$0A
The size of the resource fork	$8A
The combined size of both forks	$4A or $CA

For Directories: There is no difference from making ProDOS 8 calls in the usual manner. (Extra bits in the parameter count are ignored.) Table 3-12 lists the HFS file types that the file server converts into ProDOS 8 file types.

- **Table 3-12** File types

HFS Creator	HFS File Type	ProDOS File Type	Auxiliary File Type
(any)	TEXT	TXT ($04)	$0000
pdos	BINA	typeless ($00)	$0000
pdos	PSYS	SYS ($FF)	$0000
pdos	PS16	$B3	$0000
pdos	xxΔΔ	$xx	$0000 where xx is a 2-digit hex number and Δ is a space

If the files types in *Table 3-12* are not found, then the following is converted:

4 bytes	'p' ASCII $70	1-byte value	2-byte value

For example, the 4-byte value for a ProDOS binary file with an auxiliary file type of $800 would be $70 $06 $08 $00. The HFS creator should be 'pdos'. This conversion allows a Macintosh computer to create a file that can be transferred to a Macintosh server and have its ProDOS 8 file type set correctly.

Open

Differences in the way in which the Open call is used for files and directories are described as follows.

For Files: The Open call tells PFI what fork it wants to open. PFI remembers the type of fork, so that all of the other calls for the file that use the reference number do not have to know the fork type. *Table 3-13* lists these parameters.

- **Table 3-13** File parameters for Open command

To open	For call type, set PCount to either: ProDOS Open	Special Open fork
The data fork	$03	$04
The resource fork	$83	$84

If an open call is attempted on a file, an attempt will be made to open the file as read/write deny write. If this fails, an attempt will be made to open the file as read-only deny nothing. If this fails, an attempt will be made to open the file as write-only deny write. If this also fails, an access denied error ($4E) will be returned. This behavior is the same as for GS/OS and was done for compatibility with GS/OS

For Directories: When an Open call for a directory occurs, PFI looks at the high bits of the parameter count to find out what the user wants to include in the size of the files within that directory. *Table 3-14* lists these parameters. Three types of Catalog are possible using these features.

- **Table 3-14** Directory parameters for Open command

To find	For call type, set PCount to either: ProDOS Open	Special Open fork
The size of the data fork	$03	$04
The size of the resource fork	$83	$84
The combined size of both forks	$43 or $C3	$44 or $C4

Additional ProDOS MLI Calls

Two powerful features for creating multi-user applications are the Special Open Fork command and the Byte Range Lock command. These commands allow applications to preserve data integrity and control simultaneous access to files. *Table 3-15* lists two new ProDOS calls via the MLI for ProDOS 8 only.

■ **Table 3-15** New ProDOS calls

Command number	Name	Description
$43	Special Open Fork	Similar to ProDOS Open command, but with access specified by user. Use if your application allows data to be shared.
$44	Byte Range Lock	Used to lock out access to a portion of an open file. Use if your application allows multiple users to read and write to the same file at the same time.

Special Open Fork ($43)

The Special Open Fork call is similar to the normal Open command, with the exception of 1 additional byte that has been added to the end of the parameter list. This additional byte indicates the access that the user wants to have for the file. Applications may use the resource fork if they are designed either to run only on the network, or to run locally when the data file is on the network.

Under GS/OS, use the File System Translator call FST_Specific ($33), with command set to $0003 (Special Open Fork).

The parameter structure for the Special Open Fork call is listed here.

Position	Name	Size	Value
$00	PCount	Byte	$04 or $84
$01	Pathname Pointer	Word	--->
$03	I/O Buffer	Word	--->
$05	Reference Number	Byte	<---
$06	Access Mode	Byte	--->

The PCount field contains $04 if a data fork is to be opened, and contains $84 if a resource fork is to be opened. When set, bit 5 of the PCount field turns off buffering.

AppleTalk Filing Protocol (AFP) uses the Access Mode Byte. When you open a file over the network, you are given certain rights (privileges); these rights include

- Read/Write
- Deny Read /Deny Write

The Access Mode Byte is defined in *Table 3-16* (all reserved areas must be 0).

- **Table 3-16** Access mode byte

Bit number	Description
7	Reserved
6	Reserved
5	Deny write access to others
4	Deny read access to others
3	Reserved
2	Reserved
1	Request write access
0	Request read access

The errors returned for this command are the same as for the Open command, as follows:

Result Code	Description
$27	I/O error
$28	No device connected
$2E	Disk switched
$40	Invalid pathname syntax
$42	File Control Block table full
$44	Path not found
$45	Volume directory not found
$46	File not found
$4A	Version error
$4B	Unsupported storage type
$4E	Access not allowed
$4F	Buffer too small

Result Code	Description
$50	File is open
$52	Unsupported Volume type
$53	Invalid value in parameter list
$56	Bad buffer address
$58	Not a block device
$5A	Bitmap disk address is impossible

Byte Range Lock ($44)

The Byte Range Lock call is used to lock out access to a portion of an open file. Lock a range of bytes to ensure exclusive access to this area of the file. The lock keeps all other users from reading or writing within this area. This feature is very helpful in a multi-user application.

- ***Remember***: The procedure to follow is to lock the range, modify the range, and then unlock the range.

You must unlock an entire range that you have locked. You cannot unlock part of locked range. For Example, if you lock the range $00004180 - $00FF0080, then the only range that you can unlock is $00004180 - $00FF0080. You could not unlock the range $00004180 - $00005000.

Under GS/OS, use the File System Translator call FST_Specific ($33), with command set to $0002 (Byte Range Lock).

The following is the parameter usage for the Byte Range Lock call.

Position	Name	Size	Value
$00	PCount	Byte	$05
$01	Reference Number	Byte	--->
$02	Lock Flag	Byte	--->
$03	Offset in File	3 Bytes	--->
$06	Length of Lock	3 Bytes	--->
$09	Start of Range,	3 Bytes relative to beginning of file	<---

The Reference Number is the number given when a file is opened.

The function of bit 0 in the Lock Flag is to choose the function (Lock or Unlock). If bit 0 of the Lock Flag is set, then the range is unlocked. If bit 0 of the Lock Flag is clear, then the range is locked.

The function of bit 6 in the Lock Flag is to choose the direction of the Offset in File (before or after the selected reference point). If bit 6 of the Lock Flag is set, then the Offset in File is the length before the selected reference point. If bit 6 of the Lock Flag is clear, then the Offset in File is the length after the selected reference point.

The function of bit 7 in the Lock Flag is to choose the reference point for the lock (start of file or end of file). If bit 7 of the Lock Flag is set, then the Offset in File is relative to the end of the file. If bit 7 of the Lock Flag is clear, then the Offset in File is relative to the start of the file. You cannot unlock relative to the end of file because, between the time you lock and unlock a range, the EOF pointer could move.

The Offset in File is the distance away from the selected reference point. The Length of Lock is the amount to be protected by the lock. The value in the Start of Range field is returned to describe the location relative to the start of the file where the lock begins.

Byte Range Lock returns the position in the file of the locked range after every successful lock. It is the responsibility of the user to same this information (as well as the length of the lock) if the range is to be unlocked before closing the file.

The errors returned for this command are the same as for the ProDOS Open command, as follows:

Result Code	Description
$04	Invalid parm count
$4D	Position out of range indicates that the range you are attempting to lock overlaps a range that you have already locked.
$4E	Access denied indicates that the range you are attempting to lock overlaps a range that you have already locked.
$53	Invalid parameter may be returned for a lock length of zero (which is useless), or for an invalid lock flag (such as attempting to use a negative offset from the BOF, or unlocking relative to the EOF).

◆ *Note:* $4D, $4E, & $53 are not defined the same as in the rest of ProDOS 8.

Calls to the ProDOS Filing Interface (PFI)

This section describes calls to the ProDOS Filing Interface (PFI) on the Apple II workstation. PFI calls are utility calls for activity on server volumes that support AFP through the ProDOS 8 interface. PFI provides additional calls to handle logging in to, logging out of, and mounting server volumes.

- *Note:* Features of AFP that are not available through operating system calls can be accessed by making AFP calls directly through ASP. These AFP calls use the Session Reference # returned by the FILogin call, or the Session Reference # and Volume ID returned by the FIListSessions call. When files are opened through ASP directly, all Reads, Writes, and other calls that involve the open fork must also be done through ASP. In such cases, all buffer management is up to the caller.

The ProDOS Filing Interface allows server volumes to be pseudo-mounted in the empty slots of the workstation. In this way, server volumes are considered along with local volumes, each being accessed based on volume name. In the case of duplicate names, the server list is searched first. When a ProDOS 8 ON_LINE call is executed, the names of both the local volume and file server volume are returned with the appropriate slot and drive numbers.

Volumes from more than one server may be mounted at one time. Volumes may be pseudo mounted in slots that are actually occupied by other cards that are not block devices (such as printer cards). In such cases, the devices actually mounted in those slots are still available for use.

Table 3-17 lists the calls to the ProDOS Filing Interface (PFI) calls. The sections that follow describe each call, the parameter listing, and the result codes.

■ **Table 3-17** PFI calls

Command number	Name	Description
$2A	FIUserPrefix	Returns prefix to user directory
$2B	FILogin	Log in to server
$2C	FILoginCont	Log in continue
$2D	FILogOut	Log off from server
$2E	FIMountVol	Mount a server volume
$2F	FIListSessions	List server sessions and volumes
$30	FITimeZone	Set workstation time zone
$31	FIGetSrcPath	Get system program source path
$32	FIAccess	Set/get directory access
$33	FINaming	Set/get naming conventions
$34	ConvertTime	Converts time to/from ProDOS/AFP formats
$36	FISetBuffer	Provides a temporary storage space
$37	FIHooks *	Set/get notification vectors
$38	FILogin2 *	Enhanced Server Log In
$39	FIListSession2 *	Enhanced listing of server sessions and volumes
$3A	FIGetSVersion *	Determine version of AFP log being used

**Note:* These calls are not available on an Apple IIe workstation or an Apple IIGS running pre-5.0 system software.

FIUserPrefix ($2A)

The FIUserPrefix returns the entire prefix of the user directory.

◆ *Note:* The Apple IIGS does not retain this name following a poweroff.

Under GS/OS, use the File System Translator call FST_Specific ($33), with command set to $0008 (GetUserPath).

The parameter structure for the FIUserPrefix call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$2A
$02	Result Code	Word	<---
$04	Reserved	Byte	$00
$05	User Name Pointer	Long	--->

The user prefix is returned in the buffer pointed to by the User Name Pointer. This call always moves 64 bytes to the user's buffer, regardless of the data being moved or the format. Therefore the largest string it can return is 63 bytes (plus 1 for the length byte). The value in the Reserved field must be 0.

The result codes returned for the FIUserPrefix call are the same as those common to all general system calls.

FILogin ($2B)

The FILogin call is used to log in to a server. PFI calls and operating system MLI calls executed through the PFI can be made only to servers mounted through this call. The parameter structure for the FILogin call is listed here. Only one session per server is allowed through the FILogin call. For information on multiple sessions, refer to information on the AppleTalk Session Protocol in *Inside AppleTalk*.

- *Note:* If a log-in is executed directly through ASP, none of the PFI calls or operating system calls will work with that session.

- *Note:* Future version of the Apple IIGS may not support the FILogin call. Apple recommends that Apple IIGS applications use the FILogin2 call. If FILogin is used in conjunction with FIListSessions2, the Server Name and Zone Name will not be returned.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$2B
$02	Result Code	Word	<---
$04	SLS Network Number	<Word>	--->
$06	SLS Node Number	Byte	--->
$07	SLS Socket Number	Byte	--->
$08	Command Buffer Length	Word	--->
$0A	Command Buffer Pointer	Long	--->
$0E	Reply Buffer Length	Word	--->
$10	Reply Buffer Pointer	Long	--->
$14	Session Reference #	Byte	<---
$15	Attn Routine	Long	--->

The Command Buffer must be in the AFP format for the FPLogin call, with the first 2 bytes reserved for the AFP Command Number. When the call completes, the Reply Buffer contains the reply, if any, in AFP format. The Session Reference # field will return the ASP Session Reference Number. If the call completes with the Login Continue Error, the caller must complete the log-in process with the server by using the FILoginCont call. As far as PFI is concerned, the session has been established, unless the call completes with an error other than Login Continue.

The FILogin call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0A01	Too many sessions
$0A02	Unable to open session
$0A03	No response from server
$0A04	Login continue
$0A13	Already logged in to server
$0A15	User not authorized
$0A16	Parameter error
$0A17	Server going down
$0A18	Bad UAM
$0A19	Bad version number

FILoginCont ($2C)

The FILoginCont call is used for those user authentication methods that require it, such as the National Bureau of Standards Data Encryption Standard (NBS-DES) algorithm. The parameter structure for the FILoginCont call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$2C
$02	Result Code	Word	<---
$04	Session Reference #	Byte	--->
$05	Command Buffer Length	Word	--->
$07	Command Buffer Pointer	Long	--->
$0B	Reply Buffer Length	Word	--->
$0D	Reply Buffer Pointer	Long	--->

The Session Reference # must be the same as that returned by the FILogin call. The Command Buffer must be in the required AFP format, with the first 2 bytes being reserved for the AFP command number. The reply, if any, is returned in the Reply Buffer in AFP format. If this call fails, the session will be canceled. If the call completes with the Login Continue error, the caller must complete the log-in process with the server.

The FILoginCont call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0A03	No response from server
$0A04	Login continue
$0A06	Invalid session reference number or unknown volume
$0A15	User not authorized

FILogout ($2D)

The FILogout call is used to log off a server. This call may be used to cancel a session created by an incomplete Login. The ASP session will be canceled even if the FPLogout call (which this call executes) fails. The parameter structure for the FILogout call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$2D
$02	Result Code	Word	<---
$04	Session Reference #	Byte	--->

The Session Reference # field designates which session is to be terminated. The FILogout call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0A06	Invalid session reference number or unknown volume

FIMountVol ($2E)

Under GS/OS, use the device control call Eject to unmount volumes. (i.e. issue an eject call to the driver for the volume you wish to unmount).

The FIMountVol call is used to pseudo-mount (and unmount) server volumes on a workstation. The parameter structure for the FIMountVol call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$2E
$02	Result Code	Word	<---
$04	Session Reference #	Byte	--->
$05	Mount Flag	Byte	--->
$06	Volume Name Pointer	Long	--->
$0A	Volume ID	<Word>	<---
$0C	Slot/Drive	Byte	<---
$0D	Password Pointer	Long	--->

The Mount Flag field specifies whether the volume is to be mounted or unmounted, as shown in the *Table 3-18.*

■ **Table 3-18** Bit settings for the Mount Flag field

Bit Number	Setting	Description
7	Set	The requested volume is pseudo-mounted in a slot/drive location chosen by the firmware, provided that there is a free slot/drive location.
7	Clear	The volume specified will be unmounted.
6	Set	The password to which the Password Pointer points is placed in the packet.
0	Set	This signifies that it is a User's Volume (that is returned in FIListSessions)

Server volumes will not be mounted into slot/drive locations already occupied by locally mounted block devices. The Session Reference # field is used by the system to identify which volume is to be used. The Volume ID returned by AFP is placed in the Volume ID field, and the slot/drive (ProDOS format) into which the volume was pseudo-mounted is returned.

The FIMountVol call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0A05	Invalid name
$0A06	Invalid session reference number or unknown volume
$0A07	Access denied
$0A08	Too many volumes mounted
$0A09	Volume not mounted
$0A11	Volume already mounted

FIListSessions ($2F)

Under GS/OS, use the Volume call to determine the file system for each volume.

The FIListSessions call is used to retrieve a list of current sessions being maintained through PFI and any volumes mounted for those sessions. The parameter structure for the FIListSessions call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$2F
$02	Result Code	Word	<---
$04	Buffer Length	Word	--->
$06	Buffer Pointer	Long	--->
$0A	Entries Returned	Byte	<---

The list is placed into the specified buffer. If the buffer is not large enough, the buffer will retain the maximum possible number of current sessions and then return an error. The format of the buffer is as follows:

Position	Name	Size	Value
$00	Session Reference #	Byte	<---
$01	Slot/Drive	Byte	<---
$02	Volume Name	28 Bytes	<---
$1E	Volume ID	<Word>	<---

This list is repeated for every volume mounted for each session. For example, if there are two volumes mounted for session number 1, then session number 1 is listed two times. The Slot/Drive field contains the slot and drive numbers (in the standard ProDOS 8 format). Bit 0 of the Slot/Drive field tells if the volume is a User's Volume. If you mount more than two servers and both have user volumes, then the user volume found first in the list (scanned top to bottom) returned by FIListSessions specifies the user volume for use by an application.

The first byte of the Volume Name field contains the length of the name in bytes. If there are no volumes mounted for a session, the value of the Length Byte field is zero and the rest of the field is undefined. The Volume ID field returns the AFP volume ID for the listed volume.

◆ *Note:* The FIListSessions2 call also returns the server and zone name for each volume (if FILogin2 was used).

The FIListSessions call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0A0B	Buffer too small

FITimeZone ($30)

The FITimeZone call is used by each workstation to set its own time zone, relative to the time zone set for the server. The parameter structure for the FITimeZone call is listed here.

◆ *Note:* This call has no effect under GS/OS

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$30
$02	Result Code	Word	<---
$04	Time Flag	Byte	--->

Bit 7 on the Time Flag indicates whether the time should be added to or subtracted from the time zone selected; if Bit 7 is set high, it indicates that the hours should be subtracted from the time zone on the server (that is, as if you are going west). Bits 6 through 0 of the Time Flag indicate the actual number of hours away from the time zone selected; Bit 0 is the same time zone as the server. The FITimeZone call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0A0C	Time flag error

FIGetSrcPath ($31)

Do not use from GS/OS.

The FIGetSrcPath call returns the pathname of the last file that was opened, whether locally or over the network. This call allows system programs to determine the directory they were loaded from.

◆ *Note:* Under ProDOS 8, the pathname of an application is put at $280 when it is loaded and run. Refer to *ProDOS 8 Technical Reference Manual (1987)*, section 5.1.5 for more information. For GS/OS, use Prefix 8 and the Get_Name call (see the *GS/OS Reference Manual*).

The parameter structure for the FIGetSrcPath call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$31
$02	Result Code	Word	<---
$04	Buffer Pointer	Long	--->

The Buffer Pointer field points to the buffer where the pathname is to be placed. The buffer must be at least 129 bytes in length. The first byte of the buffer is the length of the pathname that immediately follows (up to 128 bytes).

The result codes returned for the FIGetSrcPath call are the same as for the general system calls.

FIAccess ($32)

From GS/OS, use the FST specific calls GetPrivileges and SetPrivileges.

The FIAccess call gets and sets directory access on an AFP server. The Access Rights are in AFP format. The parameter structure for the FIAccess call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$32
$02	Result Code	Word	<---
$04	Directional Flag	Byte	--->
$05	Access Rights	4 Bytes	<--->
$09	Pathname Pointer	Long	--->
$0D	Creator Name Pointer	Long	--->
$11	Group Name Pointer	Long	--->

If bit 7 of the Directional Flag is set, the access is being set. When set, bit 6 of the Directional Flag means that the Creator's Name will be dealt with, whereas bit 5 of the Directional Flag means that the Group Name is being found or set. Values may get returned into buffers pointed to by Creator and Group Name Pointers.

If the Creator's name or GroupName are being returned (bit 7 of the directional flag is clear), the buffers pointed to by creator Name Pointer and Group Name Pointer should be at least 32 bytes.

The FIAccess call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0A05	Invalid name
$0A09	Volume not mounted
$0A0A	Unable to set creator
$0A0D	Unable to set group
$0A0E	Directory not found
$0A0F	Access denied
$0A10	Miscellaneous error
$0A12	Unable to get creator and/or group

FINaming ($33)

The FINaming call sets or finds the naming convention. PFI uses the Long Name of AFP for ProDOS. Because ProDOS names are more restrictive than the Long Name of AFP, there may be files and directories that cannot be accessed by ProDOS without switching naming conventions.

Do not make this call from GS/OS. GS/OS always uses the complete AFP syntax for pathnames.

The parameter structure for the FINaming call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$33
$02	Result Code	Word	<---
$04	Directional Flag	Byte	---> Enable flag changes
$05	Naming Convention Flag	Byte	<-->

The Directional Flag enables flag changes, and the Naming Convention Flag indicates the naming convention to be used. The default naming convention is ProDOS naming, with the device table enabled. *Table 3-19* indicates how bits are set for these flags.

■ **Table 3-19** Bit settings for the FINaming call

Flag	Bit Number	Setting	Description
Directional	6	Set (1)	The device table enable or disable is changed as set by bit 6 of the Naming Convention Flag field.
		Clear (0)	Return current device table mode
	7	Set (1)	The naming mode is changed as set by bit 7 of the Naming Convention Flag field.
		Clear (0)	Return current naming mode
Naming Convention	6	Set (1)	Device Table disabled; the ProDOS device table is not updated as network volumes are mounted and unmounted.
		Clear (0)	Device Table enabled
	7	Set (1)	Use AFP Long Name naming convention
		Clear (0)	Use ProDOS naming convention

The following sample program demonstrates this new function.

```
                longa   off
                longi   off
                absaddr off
                65C02   on
DevTable        start
ATCall equ      $42
mli             equ     $bf00
                stz     FINaming+4      ;get the current FINaming setting
                jsr     mli
                dc      i1'ATCall'
                dc      a'FINaming'
                lda     FINaming+5      ;get the returned result
                sta     NameMode        ;store it until later.
DisableTable anop
                lda     #$40            ;set bit 6 to change device table update
                sta     FINaming+4
                sta     FINaming+5      ;set bit 6 to disable device table update
                jsr     mli
                dc      i1'ATCall'
                dc      a'FINaming'
RestoreTable    anop
                lda     #$40            ;set bit 6 to change device table update
                sta     FINaming+4
                lda     NameMode
                sta     FINaming+5      ;restore to the original mode
                jsr     mli
                dc      i1'ATCall'
                dc      a'FINaming'
                rts
FINaming        anop
                dc      h'00'           ;byte - sync mode
                dc      h'33'           ;byte - FINaming command
                dc      i'0'            ;word - Result
                dc      h'00'           ;byte - Direction flag
                                        ;b7 - Change naming mode
                                        ;b6 - Change Device Table update
                                        ;if zero, mode is returned
                dc      h'00'           ;byte - Mode Flag
                                        ;b7 - Enable AFP naming mode
                                        ;b6 - Disable Device Table update
NameMode        ds      1
                end
```

The result codes returned for the FINaming call are the same as are common for all general system calls.

ConvertTime ($34)

The ConvertTime call converts the time to either AFP format or ProDOS 8 format. The parameter structure for the ConvertTime call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	00 (Synchronous only)
$01	Command	Byte	$34
$02	Result Code	Word	<---
$04	Format Flag	Byte	--->
$05	From DATE/Time	Long	--->
$09	To DATE/Time	Long	<---

Both the From DATE/Time field and the To DATE/Time field contain values (data), and not pointers. If the Format Flag is 0, then From DATE/Time is in AFP format. If the Format Flag is 1, then From DATE/Time is in ProDOS format.

The ConvertTime call returns these result codes, as well as the result codes for all system calls.

Result Code	Description
$0A14	Time error

FISetBuffer ($36)

The FISetBuffer call provides temporary storage space of 512 bytes and is provided for miscellaneous use. An application must not make this call. The parameter structure for the FISetBuffer call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$36
$02	Result Code	Word	<---
$04	Direction Flag	Byte	--->
$05	Buffer Length	Word	--->
$07	Buffer Pointer	Long	--->

If bit 7 of the Direction Flag is set,data will be copied from the user's buffer to the temporary buffer, if bit 7 is clear, data will be copied from the temporary buffer to the user's buffer. Buffer length is the amount of data in bytes, to move and must not be more than 512. Buffer Pointer points to the user's buffers

The FISetBuffer call returns these result codes.

Result Code	Description
$0A1A	Buffer Too Long

FIHooks ($37)

The FIHooks call is used for changing the default event notification routine. If the login program passes the default attention routine (null) to PFI, the default hooks will be called. These default hooks can be either set or returned through this call. The parameter structure for the FIHooks call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$37
$02	Result Code	Word	<---
$04	Flag Byte	Byte	--->
$05	MountVector	Long	<-->
$09	UnmountVector	Long	<-->
$0D	AttentionVector	Long	<-->

The Flag Byte field specifies the OS type and whether the hooks are to be set or returned, as shown in Table 1.

■ **Table 3-20** Bit settings for the Hook Flag field

Bit Number	Setting	Description
7	Set (1)	ProDOS 8 active. Clear (0) GS/OS active.
6	Set (1)	The hooks will be set. Clear (0) The hooks will be returned.
5 - 0	Clear (0)	Must be zero.

Note: If bit 6 is clear, hooks to be returned, then bit 7 is ignored and the OS type will not be changed.

The MountVector field is a pointer to the routine that will be called whenever PFI adds a new volume to its internal tables.

The UnmountVector field is a pointer to the routine that will be called whenever PFI removes a volume from its internal tables. The MountVector and UnmountVector will be called in the following environment:

• = Undefined

ENTRY: Called via 'JSL' (Call cannot be made on the Apple IIe)

A Reg	=	Undefined								
X Reg	=	Low word of parameter block pointer								
Y Reg	=	High word of parameter block pointer								
D Reg	=	PFI direct page								
B Reg	=	PFI data bank								
P Reg	=	N	V	M	X	D	I	Z	C	E
		•	•	0	0	0	•	•	•	0

The parameter block contains the following data:

Byte	Session reference number	
	Byte	P8 Unit #
	PString[28]	Volume name
	Word	Volume ID
	PString[32]	Server name
	PString[33]	Zone name

EXIT: Return via 'RTL' (Call cannot be made on the Apple IIe)

A Reg	=	Undefined								
X Reg	=	Undefined								
Y Reg	=	Undefined								
D Reg	=	PFI direct page								
B Reg	=	PFI data bank								
P Reg	=	N	V	M	X	D	I	Z	C	E
		•	•	0	0	0	•	•	0	0

The AttentionVector field is a pointer to the routine that will be called whenever PFI receives a standard attention event for one of the mounted volumes.The AttentionVector will be called in the same environment as the mount and unmount vectors with the following parameter block:

Byte	Session reference number	
	Byte	Type of attention
	Word	Attention data
	PString[32]	Server name
	PString[33]	Zone name

The result codes returned for the FIHooks call are the same as those common to all general system calls.

FILogin2 ($38)

The FILogin2 call is used to log in to a server. This call work primarily like the FILogin call. The exception is that there are three additional parameters at the end of the FILogin call structure. The parameter structure for the FILogin2 call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$38
$02	Result Code	Word	<---
$04	SLS Network Number	<Word>	--->
$06	SLS Node Number	Byte	--->
$07	SLS Socket Number	Byte	--->
$08	Command Buffer Length	Word	--->
$0A	Command Buffer Pointer	Long	--->
$0E	Reply Buffer Length	Word	--->
$10	Reply Buffer Pointer	Long	--->
$14	Session Reference #	Byte	<---
$15	Attn Routine	Long	--->
$19	Server Name Pointer	Long	--->
$1D	Zone Name Pointer	Long	--->
$21	AFP Version Number	Word	--->

The Command Buffer must be in AFP format for the FILogin2 call, with the first 2 bytes reserved for the AFP Command Number. When the call completes, the Reply Buffer contains the reply, if any, in AFP format. The Session Reference # field will return the ASP Session Reference Number. If the call completes with the Login Continue Error, the caller must complete the log-in process with the server by using the FILoginCont call. As far as PFI is concerned, the session has been established, unless the call completes with an error other than Login Continue.

The Server Name Pointer and Zone Name Pointer must point to a valid Pascal String (length byte followed by name). The AFP Version word must be in the following format:

"AFPVersion 1.1" = 0101 (hexadecimal)

"AFPVersion 2.0" = 0200 (hexadecimal)

The high byte is the major version number and the low byte is the minor version number.

The Server Name, Zone Name, and AFP Version fields are NOT used by PFI to login to the server. These fields are required for the ListSessions2 and FIGetSVersion calls. It is up to the programmer making the Login2 call to verify that these parameters are correct.

The FILogin2 call returns these result codes, as well as the result codes for all system calls.

Result	Description
$0A01	Too many sessions
$0A02	Unable to open session
$0A03	No response from server
$0A04	Login continue
$0A13	Already logged in to server
$0A15	User not authorized
$0A16	Parameter error
$0A17	Server going down
$0A18	Bad UAM
$0A19	Bad version number

FIListSessions2 ($39)

The FIListSessions2 call is used to retrieve a list of current sessions being maintained through PFI and any volumes mounted for those sessions. This call work primarily like the FIListSessions call. The exception is that there are two additional parameters returned for every session. The parameter structure for the FIListSessions2 call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$39
$02	Result Code	Word	<---
$04	Buffer Length	Word	--->
$06	Buffer Pointer	Long	--->
$0A	Entries Returned	Byte	<---

The list is placed into the specified buffer. If the buffer is not large enough, the buffer will retain the maximum possible number of current sessions and then return as error. The format of the buffer is as follows:

Position	Name	Size	Value
$00	Session Reference #	Byte	<---
$01	Slot/Drive	Byte	<---
$02	Volume Name	28 Bytes	<---
$1E	Volume ID	<Word>	<---
$20	Server Name	32 Bytes	<---
$40	Zone Name	33 Bytes	<---

The FIListSessions2 call returns these result codes, as well as the result codes for all system calls.

Result	Description
$0A0B	Buffer too small

FIGetSVersion ($3A)

The FIGetSVersion call is used to determine what version of AFP was used to login to a particular server. The parameter structure for the FIGetSVersion call is listed here.

Position	Name	Size	Value
$00	Async Flag	Byte	$00 (Synchronous only)
$01	Command	Byte	$3A
$02	Result Code	Word	<---
$04	Session Number	Byte	--->
$05	AFP Version Number	Word	<---

The AFP Version word will be in the following format:

0101 (hexadecimal) = "AFPVersion 1.1"

0200 (hexadecimal) = "AFPVersion 2.0"

The high byte is the major version number and the low byte is the minor version number.

The FIGetSVersion call returns these result codes, as well as the result codes for all system calls.

Result	Description
$0A06	Invalid session reference number

Chapter 4 The AppleShare File System Translator (FST)

THIS CHAPTER describes the implementation of the AppleShare File System Translator for GS/OS. It assumes a familiarity with GS/OS and AppleTalk. Please note that AppleShare for GS/OS encompasses not only the AppleShare FST, but also the AppleTalk protocol stack, drivers, network booting (if desired), switching between the GS/OS FST and the ProDOS 8 PFI (since AppleShare will be accessible from both ProDOS 8 and GS/OS), and enhancements to the Finder to make it network aware. ■

The AppleShare FST is the implementation of AppleShare for GS/OS. It is meant to supersede AppleShare IIGS, the implementation of AppleShare for ProDOS 16. Since ProDOS 16 makes calls to ProDOS 8 to get its work done, it patches the ProDOS 8 MLI to intercept calls bound for the network. In this way, both ProDOS 8 and ProDOS 16 can use network volumes. GS/OS is completely separate from ProDOS 8. The ProDOS 8 MLI will still be patched to intercept network calls while ProDOS 8 is running. When GS/OS is running, GS/OS will make calls directly to the AppleTalk routines via the AppleShare FST, instead of calling ProDOS 8 to make the AppleTalk calls. This will increase the speed of GS/OS programs using files on the network (compared to ProDOS 16).

The AppleShare FST will only work with file servers supporting AFP version 2.0 or greater.

Compatibility

An important consideration for the AppleShare File System Translator is backwards compatibility with GS/OS, ProDOS 16 and ProDOS 8 implementations of AppleShare. All documented calls that were added to the ProDOS 8 MLI to support AppleShare will still be usable from ProDOS 8. The RamDispatch vector at $E11014 will continue to support full native mode calls from either ProDOS 8 or GS/OS.

The class 0 Open call works as the Open call for ProDOS 16. The class 1 Open call is more restrictive in its setting of deny modes which is safer for opening files. Please use class 1 Open whenever possible, and try to use the requested access parameter when possible (eg.: only ask for read if that is all you will do with the file).

File not found and path not found errors will be reported correctly (when a file is not found, the FST will check for the existence of the parent and issue a path not found if the parent does not exist). This differs from ProDOS 16 which reported path not found for both path not found and file not found error conditions.

Pathname syntax

There are two kinds of syntactic restrictions on pathname syntax: those imposed by GS/OS, and those imposed by the FST (because of naming restrictions in AFP).

GS/OS may impose a maximum length on pathnames. The AppleShare FST does not.

Because standard files will neither use nor return pathnames greater than 508 bytes, this is a reasonably practical limit for pathnames entered by a user.

The span of a pathname is the maximum number of characters in a filename (i.e. between pathname separators, including volume names). GS/OS imposes no restriction on maximum span. The AppleShare FST restricts the maximum span to be less than 32 characters. While AFP volume names are less than 28 characters, this part of the syntax is not checked. Volume names with a length of 28-31 will return a volume not found error.

GS/OS allows "/" or ":" to be a separator. The first "/" or ":" in the pathname is taken to be the separator. A ":" can never be used in a filename. A "/" cannot be used in a filename if the separator is "/". The AppleShare FST disallows a null byte in a pathname. All other characters are permitted. Note that the high bit of a character is significant. Characters with values greater than or equal to 128 are considered extended ASCII and typically display as special symbols on Macintosh and IBM systems.

Numbers as the first filename in a partial pathname are assumed by GS/OS to be prefix designators. Since numbers are valid filenames in AFP, a prefix designator should always be used explicitly with partial pathnames beginning with a number. For example, "0:555:Hello" refers to a file "Hello" in a folder "555" relative to prefix 0; "555:Hello" will give an invalid path syntax error since GS/OS assumes that "555:" is a prefix designator for prefix 555, which is invalid.

Equivalence of Macintosh and GS/OS file types

AppleShare file servers supporting AFP version 2.0 or greater maintain both Macintosh filetype and creator as well as GS/OS filetype and auxtype. Since the filetype information for the two operating systems are distinct, a workstation can set one kind of filetype for Macintosh and another type for GS/OS.

The AppleShare FST will use the Apple II filetype and auxtype fields; it depends on the server to derive appropriate type information for Macintosh files. The AppleShare File Server version 2.0 uses a convention also used by Apple File Exchange and the MAX cross-development tools.

Apple II files are distinguished by a Macintosh creator of "pdos". The Apple II filetype SYS (=$FF) has a Macintosh filetype of "PSYS". The Apple II filetype S16 (=$B3) has a Macintosh filetype of "PS16". The Apple II unknown filetype (=$00) has Macintosh filetype "BINA". Apple II text files (TXT = $04) with auxtype of $0000 (i.e. normal ASCII text, no records) has Macintosh filetype "TEXT". These special cases allow Macintosh to display unique icons for these filetypes.

Macintosh files with creator "pdos" and a filetype of the form "XY " (two hex digits followed by two spaces) will get Apple II filetype $XY and auxtype $0000. Macintosh files with creator "pdos" and a filetype of the form $70uvwxyz ($70 is a lower-case "p") have ProDOS filetype $uv and auxtype $wxyz (note the order of the bytes: on the Macintosh they are stored high-low instead of low-high).

APW source files (ProDOS filetype $B0) are given Macintosh filetype "TEXT" so that they can be edited more easily.

The conversion rules are summarized in the following tables. If more than one rule applies, the one closest to the top of the table will be used.

ProDOS -> Macintosh conversion

ProDOS		Macintosh	
Filetype	Auxtype	Creator	Filetype
$00	$0000	'pdos'	'BINA'
$B0 (SRC)	(any)	'pdos'	'TEXT'
$04 (TXT)	$0000	'pdos'	'TEXT'
$FF (SYS)	(any)	'pdos'	'PSYS'
$B3 (S16)	(any)	'pdos'	'PS16'
$uv	$wxyz	'pdos'	'p' $uv $wx $yz

Macintosh -> ProDOS conversion

Macintosh		ProDOS	
Creator	Filetype	Filetype	Auxtype
(any)	'BINA'	$00	$0000
(any)	'TEXT'	$04 (TXT)	$0000
'pdos'	'PSYS'	$FF (SYS)	$0000
'pdos'	'PS16'	$B3 (S16)	$0000
'pdos'	'XYΔΔ' †	$XY	$0000
'pdos'	'p' $uv $wx $yz	$uv	$wxyz
(any)	(any)	$00	$0000

† Where X,Y are hex digits (i.e. '0'-'9' or 'A'-'F'), and Δ is a space

System calls

This section describes differences of parameters between the AppleShare FST and the ProDOS FST. Please see the *GS/OS Reference Manual, Volume 1* for more detailed information about these calls. Any calls not documented here behave as specified in the *GS/OS Reference Manual*,

CREATE ($01)

The ProDOS filetype and auxtype will be set to the values given in the call; by default, the Macintosh creator will be set to 'pdos' and the Macintosh filetype will be derived according to the rules above. All files will be created as extended files (i.e. have both a data and a resource fork) since there is no way to distinguish between a fork of length 0 and a fork that does not exist.

In a class 1 call, the EOF and resource_EOF fields are ignored. This is because the definition of the call states that the forks' EOFs will be set to 0, and it is impossible with AFP to allocate space in a fork past its EOF.

Only the low byte of the filetype and low word of the auxtype will be used. If the high byte of the filetype or high word of the auxtype is non-zero, an invalid parameter error will be returned.

SET_FILE_INFO ($05)

The ProDOS filetype and auxtype will be set to the values given in the call; by default, the Macintosh creator will be set to 'pdos' and the Macintosh filetype will be derived according to the rules above. The option_list data is the same as for the GET_FILE_INFO call, except that only the Finder Info is used (the other fields cannot be set); any data past the Finder Info field is ignored.

If the file_sys_id field is not the same as AppleShare's file system ID ($0D), then the option_list is ignored. All FSTs will return their file system ID in the first word of the option_list and will ignore setting of the option_list info if the file_sys_id does not match theirs. This allows applications to always get and set the option_list as part of the copying process even when copying from one file system to another.

GET_FILE_INFO ($06)

Folders with no see files and no see folders access will have the read bit in their access word cleared; files, and folders with see files or see folders, have their read bit set. If the file's resource fork is not empty, the storage_type will be returned as $05 (extended), otherwise it will be returned as $01/$02/$03 (seedling, sapling, or tree) depending on the data fork's length. The option_list's data is structured as follows:

word	File_Sys_ID ($0D for AppleShare)
32 bytes	Finder Info
long	Parent Directory ID
4 bytes	Access rights (same format as Get-/SetPrivileges)

See Figure 4-14 for a diagram of the option_list structure.

See *Inside Macintosh IV*, and Macintosh Technical Notes for a description of the Finder Info

The access rights field for directories is in the same format as used in the GetPrivileges and SetPrivileges calls. For files, the field is set to all zeros. Note: this field was included to allow applications like the Finder to determine what access a user has to a folder without having to do a separate GetPrivileges call.

OPEN ($10)

The access, filetype, auxtype, and option_list parameters are as described in the SET_FILE_INFO call. If request_access is $0000 (as permitted), an attempt will be made to open the file as read/write, deny read/write. If this fails, an attempt will be made to open the file as read-only, deny write. If this fails, an attempt will be made to open the file as write-only, deny read/write. If this also fails, an access denied error ($4E) will be returned.

If the class is 0, an attempt will be made to open the file as read/write deny write. If this fails an attempt will be made to open the file as read-only deny nothing. If this fails, an attempt will be made to open the file as write-only deny write. If this also fails, an access denied error ($4E) will be returned. This behavior is the same as for GS/OS and was done for compatibility with GS/OS

Note that using class 0 Open allows files to be opened by multiple users and does not fully prevent one user from changing data that another user is reading, but it does allow multiple users to read a file without changing existing code. Class 1 Open prevents one user from writing data that another user is reading, but does not allow multiple users to read a file without explicitly asking for read-only access. Putting a file in a folder with no make changes access will cause both class 0 Open and class 1 Open with request_access = 0 to open the file for read-only and will allow multiple users to read the file (and not allow the file to be written to).

If request_access is $0001 (read-only), the file will be opened as read-only, deny write. If it is $0003 (read/write), the file will be opened as read/write, deny read/write. If it is $0002 (write-only), the file will be opened as write-only, deny read/write. If the file cannot be opened with the requested mode, an access denied error will be returned.

If you want to open a file with permissions different than above, you should use the FST specific command "Special Open Fork". That call is essentially the same as the open command, but it lets you control all of the permission bits yourself.

◆ *Note:* The System Loader loads files by opening them Read–only, Deny Write (request_access as $0001).

By default, buffering will be turned on for files or directories opened with this call. The buffer will not be filled until the first Read or Get_Dir_Entry call is made (so that buffering may be turned off after the open but before the first read). The size of the buffer for files is 512 bytes; for directories it is 2048 bytes.

Folders with neither see files nor see folders access rights cannot be opened (since the only valid operation on an open folder is GET_DIR_ENTRY). The returned error code is $4E (access denied).

With a class 1 call, all of the parameters after the resource number are file information. Think of this as a combined GET_FILE_INFO and OPEN call (and in fact, that is how it behaves). In particular, the access word returned is not an indication of the access rights you have when the file is opened; it is really a "best case" access to the file. The actual access you get when opening the file is controlled by several things: the access word, access privileges to ancestor and parent folders, and access restrictions ("deny modes") imposed by other users who have the file open.

Note that using a class 1 open with request_access = 0, is usually not a good idea since you don't know what access you really got to the file (until you try) because the FST will try several combinations as described above. If your application can deal with several different kinds of access to the file, it is best to try those different access modes individually until you get one you can handle. For example, if you can handle either read-write or read-only access but prefer read-write, try opening the file with request_access = 3 (read-write). If this fails, try opening with request_access = 1 (read-only). This way you will know exactly what access you have to the file. Remember, too that if you use class 1 open for read-write, nobody else will be able to open the file and multiple users won't be able to run your application at the same time.

If you use a class 1 call with PCount > 4 (i.e. you are asking for file info to be returned), and you don't have privileges to see the object you are opening (if the object is in a drop folder, for example), the call will return with an error $4E (access denied), since you don't have access to get the file info you requested.

READ ($12)

The READ call will not be supported for directories. GS/OS directories will not be synthesized. One should use the Get_Dir_Entry call to enumerate directories. A read on a directory will return error $4E (access denied).

If part of the range to be read is locked by another workstation, a $4E error will be returned and the transfer count will be set to indicate the number of bytes transferred before the locked range was encountered.

◆ *Note:* There may be bytes that were not part of the locked range, but were not transferred.

Regardless of the value in the cache priority field, data will not be put in the system cache. By default the FST maintains a block buffer containing the 512 bytes of the block containing the current mark. This block buffer can be controlled on a per-file basis by the FST specific call "Buffer Control".

If buffering is disabled and newline mode has been enabled with more than one newline character, the read will be completed one byte at a time. This is done because the server's newline mechanism provides for only one newline character. Beware that this mode of reading a file imposes tremendous amounts of overhead and should be avoided if at all possible.

WRITE ($13)

Regardless of the value in the cache priority field, data will not be put in the system cache. By default the FST maintains a block buffer containing the 512 bytes of the block containing the current mark. This block buffer can be controlled on a per-file basis by the FST specific call "Buffer Control".

Writes to directories are not allowed. They will return error $4E (access denied).

CLOSE ($14)

The file will always be closed, even if there is an error. This is because any error an application gets may not be correctable by the application or the user (eg. the data to be flushed before the close is locked by another workstation, or a connection has been lost with the server).

SET_EOF ($18)

If a fork is extended (made longer), the additional bytes will be allocated but might not all be zero.

In a class 1 call, if the base indicates that the EOF should be set to EOF - displacement, the server's current EOF will be determined and the EOF will be set relative to that; this could be different than the workstations assumption of the EOF if another workstation has modified the fork's EOF. This could also delete data that another workstation has written between the times when the current EOF was determined and the new EOF set.

This call will force any buffered data to be written to the server. The EOF will be set after this data is written.

GET_EOF ($19)

The fork's EOF will be determined from the server; this may not match the workstation's assumption of the EOF if another workstation has modified the fork's EOF. Note that another workstation could change the EOF after completion of this call, making the results inaccurate.

This call will force any buffered data to be written to the server. The EOF will be determined after this data is written.

GET_DIR_ENTRY ($1C)

Get_Dir_Entry is not supported for files. It will return the error $4E (access denied).

Folders enumerated by GET_DIR_ENTRY that have neither see files nor see folders will have the read bit in their access word cleared. Folders with see files or see folders will have the read bit set.

The access, filetype, auxtype, and option_list parameters are as the GET_FILE_INFO call. The FST will internally maintain the directory entry number (entry_num) to allow forward and backward scanning of the directory. By default, several entries will be buffered for better performance (this can be disabled by using the FST Specific call "Buffer Control"). An end of directory error ($61) will be returned when an entry is requested that does not exist in the buffer (or buffering is disabled for the directory), and that entry cannot be read from the server.

Since AppleShare is a shared file system, entry_num may change for a file, even while the directory is being scanned because other users could add or delete files in the directory. Also, if the base and displacement fields are both zero, the total number of entries will be returned. Note that more or fewer entries may actually be returned if the directory is enumerated since other machines can create and delete files while you are enumerating the directory.

The best way to enumerate a directory is to simply open the directory and make successive Get_Dir_Entry calls with base and displacement both set to $0001. When you get an error $61 (end of directory), you are finished enumerating. You should remove duplicate entries from your list.

READ_BLOCK ($22)

This call will return an error $88 (network error) for AppleShare devices, in order to be compatible with System Disk 3.2. Remember, the preferred method for identifying a network volume is by doing a Volume call and seeing that the file_sys_id = $0D.

WRITE_BLOCK ($23)

This is an invalid operation for an AppleShare device. This call always return an error. The current error code is $4E (access denied). This is different from the $88 returned under 3.2, and may change in the future.

FORMAT ($24)

This is an invalid operation for an AppleShare device. This call always return an error. The current error code is $2B (write protected). This is different from the $88 returned under 3.2, and may change in the future.

ERASE_DISK ($25)

This is an invalid operation for an AppleShare device. This call always return an error. The current error code is $2B (write protected). This is different from the $88 returned under 3.2, and may change in the future.

GET_BOOT_VOL ($28)

If GS/OS is booted over AppleTalk, this command will return the name of the user volume on the server the user logged in to during booting. All system files should be present on this volume just like any other boot volume.

GET_FST_INFO ($2B)

The file_sys_id will be returned as $0D (AppleShare). The attribute parameter will be returned as $0000 (System Call Manager should not uppercase pathnames, do not clear high bits of pathname, this is a block FST, formatting not supported). The block_size parameter will be returned as 512; this value is only useful in determining the number of bytes used, free, and total on a volume (since these values are given in blocks).

FST_SPECIFIC ($33)

Used to make control calls to the FST. The FST specific calls are described in the section titled "FST Specific Calls" following.

FST_SPECIFIC calls

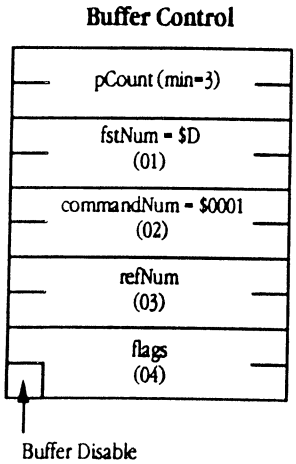
The FST_SPECIFIC call is used to make special calls to the AppleShare FST. The FST number must be $0D (AppleShare). A command of $0000 is invalid. Commands $000E through $FFFF are reserved.

If the command number is out of range, error $53 (invalid parameter) will be returned. Error $52 (unknown volume type) will be returned if a refnum for a file opened by another FST is used. Error $52 will also be returned if a pathname uses a device name for a device other than an AFP (AppleShare) driver. Error $45 (volume not found) will be returned if a pathname specifies a volume name that does not match any mounted AppleShare volume (even if a volume by that name exists for a different file system).

Buffer Control ($0001)

word	PCount (minimum = 3)
word	FST# = $D
word	Command = 1
word	Reference #
word	Buffer Disable flags (default = $0000)
	Bit 15 set = Disable buffering (every read/write goes to server,every GetDirEntry translated into a single FPEnumerate).
	Bits 0-14 Reserved

■ **Figure 4-1** Buffer Control



Command $0001 is the Buffer Control command. It is followed by a word specifying the reference number of a file/directory whose buffering is to be enabled/disabled. The next word is optional. It specifies the buffer disable flags; if the high bit is set, then buffering is disabled for that file/directory. The default value of the buffer disable parameter is $0000 (turn on buffering). A file reference number of 0 is invalid.

For folders, the buffer size is 2048 bytes. When buffering is off, each Get_Dir_Entry will immediately cause an enumerate of one entry from the server. When a Get_Dir_Entry call is made with buffering on, the requested entry will be returned from the buffer if possible. Otherwise, the buffer will be filled with as many entries from the server as possible, including the requested entry; then the requested entry will be returned. The buffer is not pre-filled when the folder is opened. The number of entries kept in the buffer is variable and depends on the size of the long and short names of the files/folders.

For files, the buffer size is 512 bytes (the same as the block size reported by the FST). When buffering is off, every Read and Write call transfers data from/to the user's data buffer directly to/from the server. When buffering is on, and a Read or Write of 512 bytes or more is made, any unwritten data in the buffer is written and the Read/Write is made from/to the user's data buffer directly to/from the server.

When buffering is on and a Read or Write of less than 512 bytes is made, the block (512 bytes, with a starting offset that is a multiple of 512 bytes) containing the first byte to be read/written is read into the buffer; if the block was already in the buffer, no read is done; if a different block is in the buffer, any unwritten data is written and the new block is read into the buffer. The read/write then proceeds to the end of the buffer. If the read/write extends past the end of the buffer, any unwritten data is written and the next block is read into the buffer. The read/write then completes by reading/writing from/to the buffer.

Unbuffered reads with 0 or 1 newline characters are handled directly by the server (i.e. the read to the server requests the same number of bytes as the user requested). Unbuffered reads with 2 or more newline characters turn into reads of one character at a time from the server (until a newline is encountered or all bytes have been read or end of file reached); please note that this takes a LONG time, and you are probably better off not using 2 or more newline characters with buffering off.

Buffered reads with 1 or more newline characters become reads of 512 bytes at a time, on 512 byte boundaries (as if it were a read of less than 512 bytes). Each block is read into the buffer and then the bytes are copied to the user's data buffer one at a time (while being compared against all the newline characters). Buffered reads with no newline characters are as described above.

Byte Range Lock ($0002)

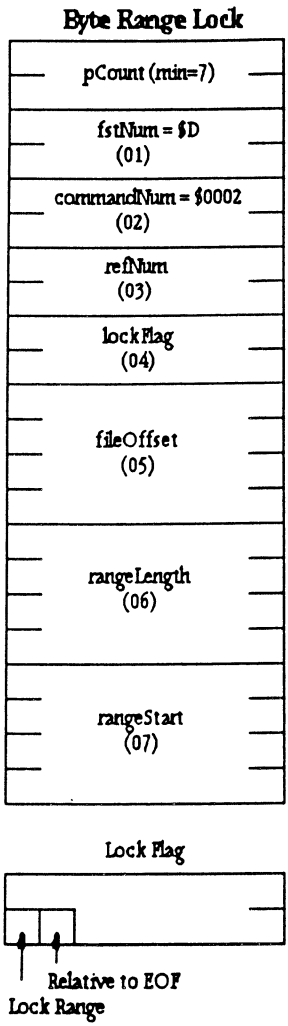
word	PCount = 7		
word	FST# = $D		
word	Command = 2		
word	Reference #		
word	Lock Flag		
	Bit 15	set	Lock range
		clear	Unlock range
	Bit 14	set	Offset relative to EOF
		clear	Offset relative to start of file
long	Offset in File		
long	Length of Range		
long	Start of Range (returned)		

For the Lock Flag, the following constants can be combined:

Lock_Range = $8000

Relative_to_EOF = $4000

■ **Figure 4-2** Byte Range Lock



Command $0002 is the Byte Range Lock command. It is followed by five required parameters (so the PCount field should be 7, 2 for FST # and Command, 5 for the parameters of Byte Range Lock). The first parameter is a word containing the reference number of the file to lock. The second parameter is the Lock Flag. If bit 15 is set, the range will be locked; if clear, it will be unlocked. If bit 14 is set, the offset is relative to the end of the file; if clear, the offset is relative to the start of the file. All other bits are reserved and should be set to 0. The next parameter is a long word containing the offset into the file (may be negative if relative to the end of the file). The next parameter is the length of the range to be locked. The last parameter is the actual start of the locked range (relative to the beginning of the file) as returned by the server.

Possible errors are: $4D (position out of range -- user already has some or all of range already locked, or unlocking a range not locked by that user), $4E (access denied -- some or all of range is locked by another user), $43 (invalid reference number), $53 (invalid parameter).

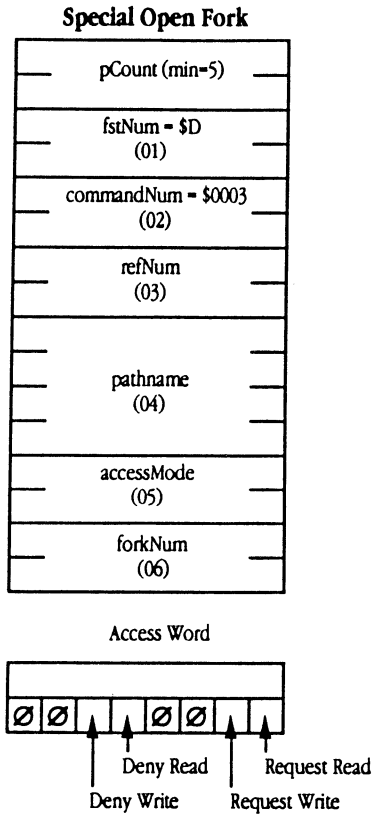
Special Open Fork ($0003)

word	PCount (minimum = 5)
word	FST# = $D
word	Command = 3
word	Reference # (returned)
long	Pointer to class 1 pathname
word	Access mode

Bit 0	Request Read Access
Bit 1	Request Write Access
Bits 2,3	Reserved
Bit 4	Deny Read to others
Bit 5	Deny Write to others
Bits 6..15	Reserved

word Resource number (default =$0000)

■ **Figure 4-3** Special Open Fork



Command $0003 is the Special Open Fork command. It is followed by three required parameters and one optional parameter (so the PCount field should be 5 or 6: 2 for the FST# and Command, and 3 or 4 for the parameters of Special Open Fork). The first parameter is the reference number (ref_num) returned by GS/OS to the access path. Use this ref_num the same as you would a ref_num returned by an OPEN call. The second parameter is a pointer to a class 1 string representing the pathname of the file to be opened. The third parameter is the access mode giving the read/write permissions desired and to be denied to others as described below. The forth, and optional, parameter is the resource number: a value of $0000 will cause the data fork to be opened, a value of $0001 will cause the resource fork to be opened; a value of $0000 is assumed if the parameter is not given.

The access word is arranged as follows (if the bit is set, the condition is asserted):

Bit 0	Request Read Access
Bit 1	Request Write Access
Bit 2,3	Reserved
Bit 4	Deny Read to others
Bit 5	Deny Write to others
Bits 6..15	Reserved

By default; files opened with SpecialOpenFork will have buffering turned off (to prevent "state" data when other users are writing to the file).j This can be changed with the BufferControl call.

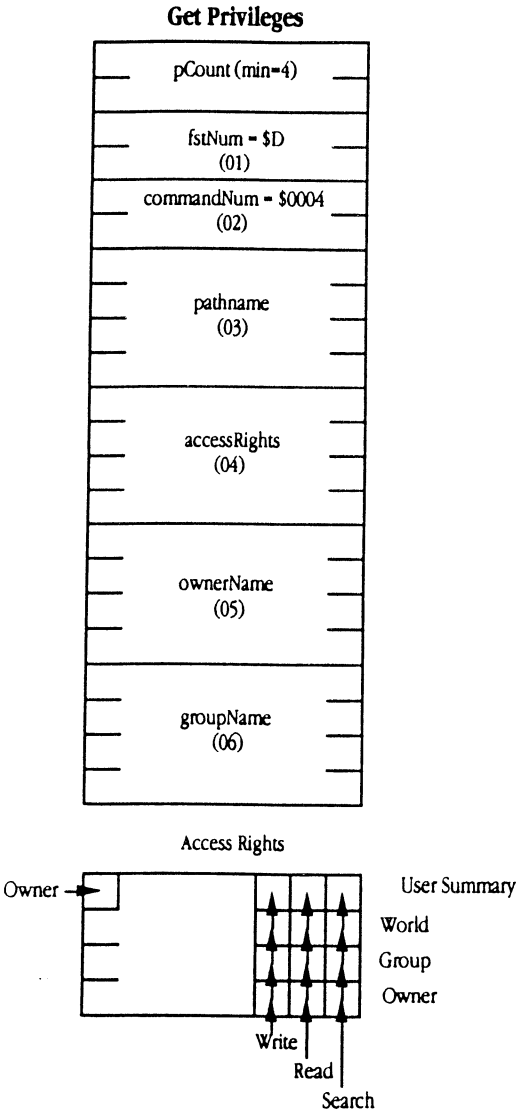
◆ *Note:* This parameter has the same meaning as in the ProDOS 8 Special Open Fork command.

Possible errors: same as for OPEN command. A deny mode conflict will result in an access denied error.

GetPrivileges ($0004)

word	PCount (min = 4)		
word	FST# = $D		
word	Command = 4		
long	Pointer to class 1 pathname		
long	Access Rights (returned)		
	byte	User Summary	
		Bit 0	See Folders allowed
		Bit 1	See Files allowed
		Bit 2	Make Changes allowed
		Bits 3..6	Reserved
		Bit 7	Owner (set if you are folder owner)
	byte	World	
		Bit 0	See Folders
		Bit 1	See Files
		Bit 2	Make Changes
		Bits 3..7	Reserved
	byte	Group	
		Bit 0	See Folders
		Bit 1	See Files
		Bit 2	Make Changes
		Bits 3..7	Reserved
	byte	Owner	
		Bit 0	See Folders
		Bit 1	See Files
		Bit 2	Make Changes
		Bits 3..7	Reserved
long	Pointer to GS/OS output buffer for Owner Name		
long	Pointer to GS/OS output buffer for Group Name		

■ **Figure 4-4** Get Privileges



Command $0004 is the GetPrivileges command. It is followed by four parameters, the first two of which are required (so the minimum PCount is 4 and the maximum is 6). The first parameter is a pointer to a class 1 pathname of a directory whose access privileges are to be set or retrieved. The second parameter is a long where access rights for the directory will be returned. The third parameter is a pointer to a GS/OS output buffer where the owner's name will be stored. The fourth parameter is a pointer to a GS/OS output buffer where the group name will be stored.

The access rights field consists of four bytes: one each for user summary, world access, group access, and owner access. For each of these bytes, bit 0 is search access (see folders), bit 1 is read access (see files), and bit 2 is write access (make changes). The user summary byte reflects the access that the current user has for that directory; if bit 7 is set, the current user is the owner of the directory.

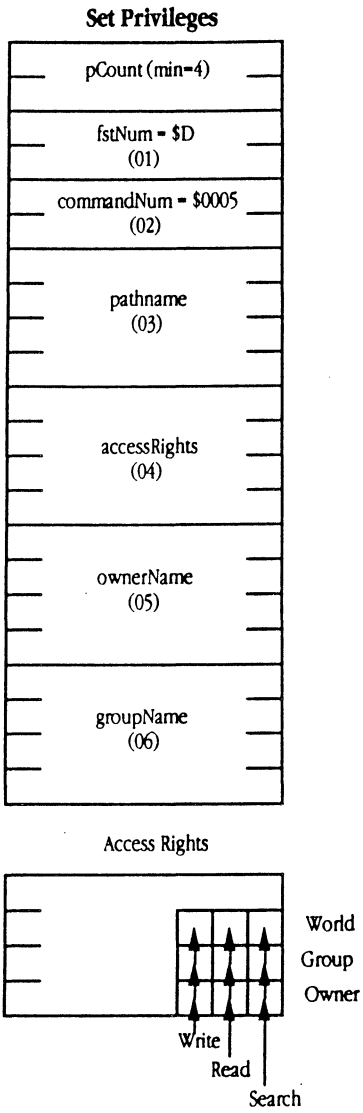
If the folder is owned by the guest user (usually displayed as "<Any User>"), the owner name will be returned as a null string. If the folder has no group associated with it, the group name will be returned as a null string.

Possible errors include: $4B (bad storage type) if the pathname specifies a file instead of a folder.

SetPrivileges ($0005)

word	PCount (min = 4)		
word	FST# = $D		
word	Command = 5		
long	Pointer to class 1 pathname		
long	Access Rights		
	byte Reserved		
	byte World		
		Bit 0	See Folders
		Bit 1	See Files
		Bit 2	Make Changes
		Bits 3..7	Reserved
	byte Group		
		Bit 0	See Folders
		Bit 1	See Files
		Bit 2	Make Changes
		Bits 3..7	Reserved
	byte Owner		
		Bit 0	See Folders
		Bit 1	See Files
		Bit 2	Make Changes
		Bits 3..7	Reserved
long	Pointer to buffer where Owner Name is stored (same format as a GS/OS output buffer, but the buffer length word is ignored).		
long	Pointer to buffer where Group Name is stored (same format as a GS/OS output buffer, but the buffer length word is ignored).		

■ **Figure 4-5** Set Privileges



Command $0005 is the SetPrivileges command. Its parameter list is the same as for the GetPrivileges command except that the access rights, owner name, and group name fields are input instead of output (since the values are being set, not retrieved). The owner name and group name point to structures similar to a GS/OS output buffer where the first word (normally a total buffer length) is ignored, the next word is the string length, and the rest of the buffer is the string itself. This structure definition allows you to do a GetPrivileges call, modify the data, and do a SetPrivileges call using the same owner name and group name pointers (the same way you can share the option_list parameter for Get_File_Info and Set_File_Info).

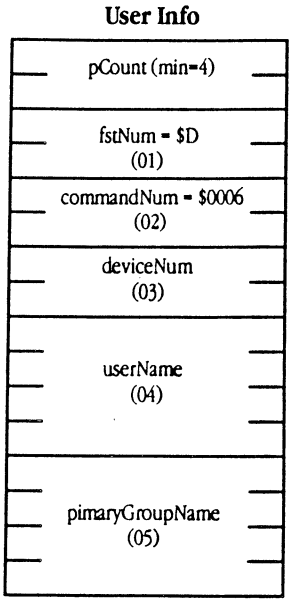
Setting the owner name to the null string assigns the folder to the guest user (usually known as "<Any User>"). The string "<Any User>" is not a valid user name (unless you have a registered user by that name). Setting the group name to the null string causes no group to be associated with the folder (and therefore the group's access rights are ignored).

Possible errors include: $4B (bad storage type) if the pathname specifies a file instead of a folder, $4E (access denied) if the user is not the current owner of the folder, $7E (unknown user) if the user name given is not the name of a registered user, and $7F (unknown group) if the group name given is not the name of a group.

User Info ($0006)

word	PCount (min = 4)
word	FST# = $D
word	Command = 6
word	Device number (of any volume on the desired server)
long	Pointer to GS/OS output buffer for User Name
long	Pointer to GS/OS output buffer for Primary Group Name

■ **Figure 4-6** User Info



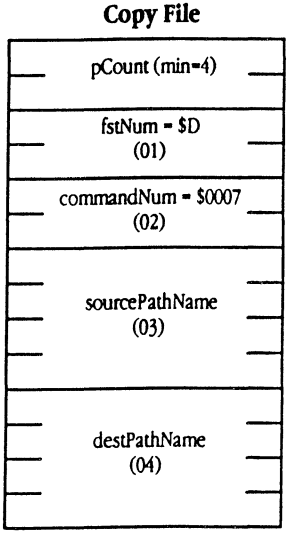
Command $0006 is the User Info command. This command will return the user name and primary group name of a user. It has two required parameters and one optional parameter. The first parameter is the device number of a volume on the server whose user info is to be returned. The second parameter is a pointer to a GS/OS output buffer where the user name is returned. The third parameter (optional) is a pointer to a GS/OS output buffer where the user's primary group name is returned.

If the user is logged on as a guest, the user name will be returned as a null string. If the user has no primary group, then it will be returned as a null string.

Copy File ($0007)

word	PCount (min = 4)
word	FST# = $D
word	Command = 7
long	Pointer to class 1 string of source pathname
long	Pointer to class 1 string of destination pathname

■ **Figure 4-7** Copy File



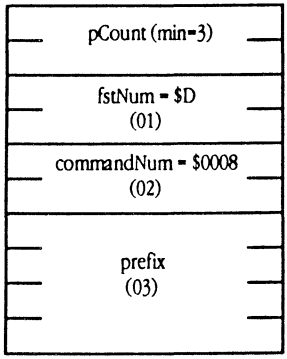
Command $0007 is the Copy File command. This command will cause a file on a server to be copied by the server. The copy may be between different volumes as long as both volumes are on the same server. This call has two required parameters. The first is a pointer to a class 1 string containing the source file's name. The second is a pointer to a class 1 string containing the destination file's name.

Possible errors include: $53 (invalid parameter) if either volume is not a server volume or if the volumes are not on the same server, $4A (version error) if the server does not support this call.

GetUserPath ($0008)

word	PCount (min = 3)
word	FST# = $D
word	Command = 8
long	Pointer to class 1 string containing prefix (returned)

■ **Figure 4-8** GetUserPath

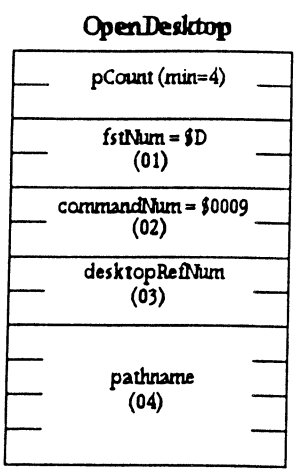


Command $0008 is the GetUserPath command. It returns a pointer to a class 1 string containing the pathname of the user's folder on the user volume, using colons as separators and without a trailing colon. The prefix string is not written into a class 1 output buffer, so you should copy the string into local buffer of sufficient size. If there is no user volume mounted, or the user name could not be determined for some reason, a data unavailable error is returned ($60). This path is constructed on each call (unlike the FIUserPrefix call). The string's contents will not change until the next call to GetUserPath. DO NOT modify the string. The string is suitable for use as a parameter to a SetPrefix call.

OpenDesktop ($0009)

word	PCount (min = 4)
word	FST# = $D
word	Command = 9
word	Desktop refnum (returned)
long	Pointer to class 1 string of path/volume name

■ **Figure 4-9** OpenDesktop

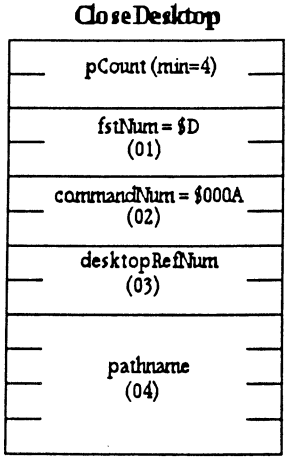


Command $0009 is the OpenDesktop command. It takes a volume/path name and returns a desktop refnum (DTRefnum). A desktop refnum must be supplied for all other desktop database calls (currently, only for getting/setting file comments).

CloseDesktop ($000A)

word	PCount (min = 4)
word	FST# = $D
word	Command = $A
word	Desktop refnum
long	Pointer to class 1 string of path/volume name

■ **Figure 4-10** CloseDesktop

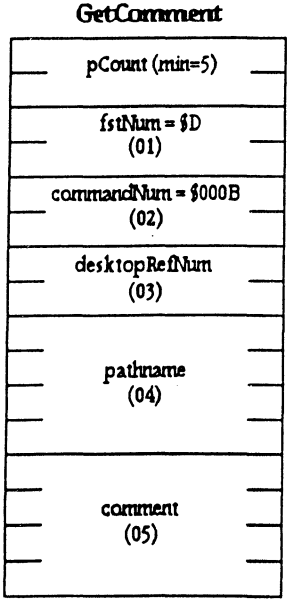


Command $000A is the CloseDesktop command. It takes a desktop refnum and volume/path name and frees all resources allocated when that refnum was opened.

GetComment ($000B)

word	PCount (min = 5)
word	FST# = $D
word	Command = $B
word	Desktop refnum
long	Pointer to class 1 string of pathname
long	Pointer to class 1 output buffer for comment

■ **Figure 4-11** GetComment

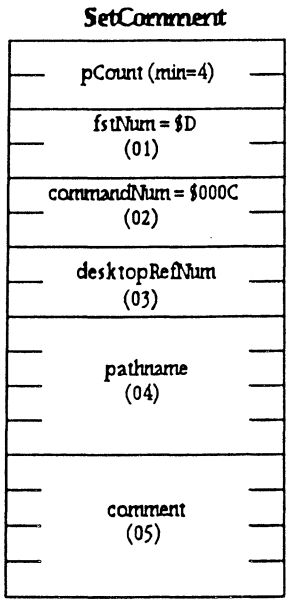


Command $000B is the GetComment command. It takes a DTRefnum and a pathname and returns a string (the comment associated with that file/folder). If no comment has been stored for that file/folder, then a null string will be returned for the comment.

SetComment ($000C)

word	PCount (min = 4)
word	FST# = $D
word	Command = $C
word	Desktop refnum
long	Pointer to class 1 string of pathname
long	Pointer to class 1 string of comment (default = null string)

■ **Figure 4-12** SetComment

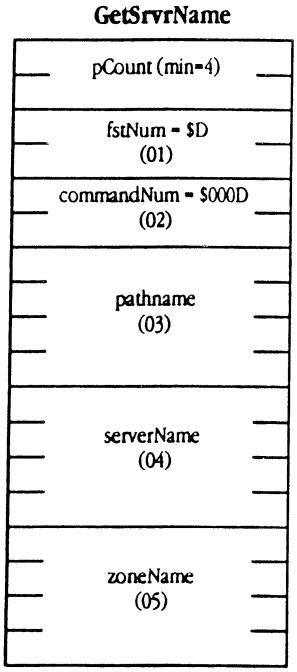


Command $000C is the SetComment command. It takes a DTRefnum, a pathname, and a string. If the string is non-null, then the comment for that pathname will be set to the given string. If the string is null, then the comment for that pathname will be removed. Note: if the comment string is longer than 199 characters, it will be truncated to 199 characters without an error.

GetSrvrName ($000D)

word	PCount (min = 4)
word	FST# = $D
word	Command = $D
long	Pointer to class 1 pathname
long	Pointer to class 1 output buffer for server name
long	Pointer to class 1 output buffer for zone name

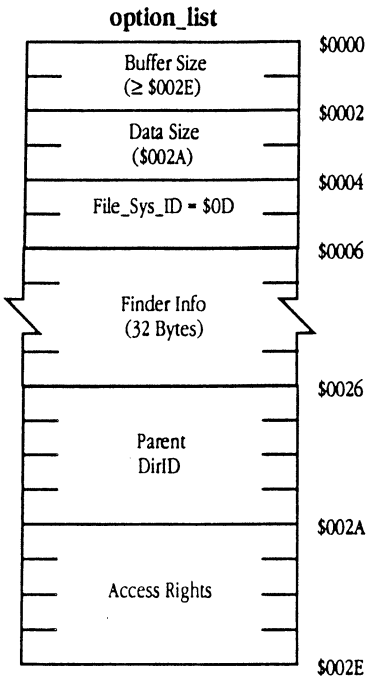
■ **Figure 4-13** GetSrvrName



Command $000D is the GetSrvrName command. It takes a pathname and returns the server name and zone name for that volume. If either of the server name or zone name buffer pointers are null ($0000 0000), that string will not be returned. If the server name or zone name are unknown, they will be returned as null strings.

Option List

■ **Figure 4-14** Option List



General implementation

When handling file system calls, the FST will itself create and send AFP packets to the server as opposed to trying to make the calls through PFI.

The only syntax checking that will be performed on pathnames is that the span (maximum length of a filename component) is less than or equal to 32 characters (the max for AFP); GS/OS itself will enforce the restriction against colons and nulls in a pathname component.

Normally, pathnames sent to the server will be relative to the root of the volume (i.e. the ancestor ID will be 2 = the volume directory). When a pathname is too long to fit in a packet, the FST will break it up into packet-size chunks by taking as many components from the start of the path as possible, finding its DirID, and repeat as needed until a DirID and partial path is obtained for accessing the file. This will cause more network traffic, but allow for long pathnames.

The session/volume level information will be obtained from AppleShare device drivers. The .AFPn drivers maintain the relationship between an AppleShare volume and a Device Information Block (DIB). The FST maintains the Volume Control Record (VCR) for AppleShare volumes that are mounted.

If interrupts are disabled when the FST has to make an AppleTalk call (i.e. an SPCommand or SPWrite), an I/O Error (error code $27) will be returned instead of making the call. In most cases, this error will be propagated back to the user. Note that some calls may not require an AppleTalk call to be made (such as GetMark) and will complete correctly with interrupts disabled; some calls (such as Read and Write with small request counts, or GetDirEntry) may or may not complete with interrupts disabled (depending on the current mark, any data that is buffered, etc.). It is strongly encouraged that file system calls should not be made with interrupts disabled!

Appendix A **Result Codes**

THIS APPENDIX summarizes the result codes for calls to the Apple IIGS workstation. Table A-1 lists each code by number, with a brief description. ■

■ **Table A-1** Description of result codes

Code	Description
Result Codes common to all system calls	
$0000	No error
$0101	Invalid command
$0102	Heap/memory management error
$0103	No timer installed
$0104	Sync/Async call error
$0105	Too many times
$0106	Timer Cancelled
Result Codes for LAP Calls ($02xx)	
$0201	No packet in buffer
$0202	End of buffer
$0203	LAP data too large
$0204	Retry count exhausted
$0205	Illegal LAP type
$0206	Duplicate LAP type
$0207	Too many protocols
$0208	Type not found
$0209	Data lost in purge
Result Codes for DDP Calls ($03xx)	
$0301	Too many sockets open
$0302	Socket not open
$0303	Socket already open
$0304	Invalid socket type
$0305	DDP data too large
$0306	No bridge available
Result Codes for NBP Calls ($04xx)	
$0401	Too many names
$0402	Name already exists
$0403	Name not found
$0404	User's buffer full
$0405	Wildcard not allowed
$0406	Invalid name format
$0407	Incorrect address
$0408	Too many NBP processes

■ **Table A-1** Description of result codes (continued)

Code	Description
$0409	NBP aborted
$040A	NBP Param Block not Found
Result Codes for ATP Calls ($05xx)	
$0501	ATP data too large
$0502	Invalid ATP socket
$0503	ATP control block not found
$0504	Too many active ATP calls
$0505	No release received
$0506	No response active
$0507	ATP send request aborted
$0508	ATP send request failed, retry exceeded
$0509	Async call aborted, socket was closed
$050A	Too many ATP sockets
$050B	Too many responses expected
$050C	Unable to open DDP socket
$050D	ATP Send Response was released
Result Codes for ZIP Calls ($06xx)	
$0601	Network error
$0602	ZIP overflow
$0603	ZIP not found
Result Codes for ASP Calls ($07xx)	
$0701	Network error
$0702	Too many ASP calls
$0703	Invalid reference number
$0704	Size error
$0705	Buffer error
$0706	No response from server
$0707	Bad version number
$0708	Too many sessions
$0709	Server busy
$070A	Session closed
Result Codes for PAP Calls ($08xx)	
$0801	Too many sessions

■ **Table A-1** Description of result codes (continued)

Code	Description
$0802	Invalid reference number
$0803	Quantum error
$0804	Too many commands
$0805	Name not found
$0806	Session closed
$0807	Network error
$0808	Server not responding
$080A	Buffer size error
Result Codes for RPM Calls ($09xx)	
$080B	PAP in use
$0901	Invalid flag byte
$0902	Invalid time values
Result Codes for PFI Calls ($0Axx)	
$0A01	Too many sessions
$0A02	Unable to open session
$0A03	No response from server
$0A04	Login continue
$0A05	Invalid name
$0A06	Invalid session reference number or unknown volume
$0A07	Unable to open volume
$0A08	Too many volumes mounted
$0A09	Volume not mounted
$0A0A	Unable to set creator
$0A0B	Buffer too small
$0A0C	Time flag error
$0A0D	Unable to set group
$0A0E	Directory not found
$0A0F	Access denied
$0A10	Miscellaneous error
$0A11	Volume already mounted
$0A12	Unable to get creator and/or group
$0A13	Already logged in to server
$0A14	Time error
$0A15	User not authorized
$0A16	Parameter error
$0A17	Server going down
$0A18	Bad UAM
$0A19	Bad version number

Appendix B **Be AppleShare Aware**

AN "APPLESHARE AWARE" program is a program that can be successfully run from an AppleShare file server. Such a program should be able to load and save files on a file server, and be fully functional. It should be able to handle error conditions in a reasonable manner (such as putting up a dialog box instead of crashing the machine), and the user should be able to quit from the program and return to a calling program (instead of having to reboot or power off the machine).

This document describes some steps you can take as a developer to help make your programs AppleShare aware. It also describes some things you can do to make your programs even more usable in an AppleShare environment (such as being multi-launch), and how to take advantage of some AppleShare-specific features. ■

Multi-launch applications

A multi-launch application is one that can be launched (executed) by more than one computer at a time. Multi-launch applications are particularly important for the Apple II family since most schools use Apple II's and it is common for an entire class to use the same application at the same time. Teachers are much more likely to use a multi-launch application on a file server than to distribute individual disks for each student.

The Apple II operating system has traditionally been a single user, single computer operating system and file system. With the addition of AppleShare support to the Apple II, many computers (and many types of computers) can share the same files (on the file server) at the same time. It is not hard to make a program multi-launch; it just takes some thinking and care about how you use files.

The first thing to remember about multi-launch applications is that one copy of the application will be shared by several computers. The system loader will take care of opening and loading the file in a safe manner such that several computers can load the application at the same time. As the programmer, you must remember that you should not write to the application files (to save configuration information, for example) just like you shouldn't write in books borrowed from a library -- other people have to use it, too.

System Software 5.0 has a new feature called the "@" prefix. It is a system prefix defined when your application is launched. If the application was launched from an AppleShare volume, it will be set to the name of the user's folder on the file server. If the application was launched from a non-AppleShare volume, it will be set to the name of the folder containing the application. If you use the "@" prefix as part of the pathname for saving configuration information, it will automatically go in a safe place, separate for each user. For example, if your program was called "Fred", you might use the pathname "@:Fred.Config" for storing preferences and configuration data.

Sharing open files

The class 1 version of the Open call lets you supply a parameter indicating the access you require to the open file. You can specify read, write, read and write, or "as permitted". If you request read permission (request_access=1), it will also deny others the ability to write to the file (so they can't change the data you are reading). If you request write or read and write (request_access =2 or 3, respectively), it will deny others the ability to open the file at all (so they cannot read data as you are changing it, and so they cannot overwrite your changes to the data). Realize that "as permitted" (request_access=0) will first try to open the file for read and write (meaning no other computer can open it); if that fails, it will try read-only; if that fails, it will try write-only. Note that there is no way of knowing what access you have to the file, and you may not have read/write access. If your program opens files, think about how it uses the contents of the files, and open them in an appropriate manner.

For example, an adventure game might want to load a map of rooms in a dungeon. In this example, the program really only needs to read the contents of the file, and not modify the file. Since all you need to do is read the file, you should open the file read-only (request_access=1). If you do this, and several computers run the program at the same time, they will all be able to open the dungeon file successfully (since a read-only open allows others to open the file read-only). If you use request_access=0, or don't even supply the field (it is optional), only the first computer will be able to open the file; the rest will get an error when trying to open the file (access denied, $4E).

As a second example, consider a word processing program. It would want to read from the file so that it can be displayed or printed. It would also want to write to the file so that it can be edited and save the changes. In this case, the program would open the file with request_access=3 (read and write). Don't assume that request_access=0 will give you read and write access; other users who have opened the file, or access privilege settings may restrict your access. Also, the file should be kept open the entire time the file is being edited. If you don't, another computer could open the file for editing after you have closed it. Then, the edited version that is written last will stay, and all other versions will be overwritten.

As a third example, consider a file copying program (like the Finder). It would open the source file read-only (so that other computers can copy it or use it). It would open the destination file write-only (request_access=2) since it only needs to write to the file, and no other computer should be allowed to read or write to the copy while it is being written. Note that opening the destination for read and write could cause the open to fail if access privileges to the file prevent read access (such as if the file is in a "drop folder").

The class 0 version of the Open call is compatible with the ProDOS 16 Open call. Since it did not provide a mechanism to tell the operating system what access was needed to the file, it allows files to be opened in a manner that is not completely safe in order that several computers could open the same file at the same time (the first computer to open the file could potentially change it as other computers are trying to read from it). The class 1 Open call is safe, and allows you to specify the access that you require to the file.

All authors are strongly encouraged to use the class 1 version of the Open call and to use a non-zero value for the request_access field. This way, files can be shared if possible, and if the open succeeds, you will know that you have the access to the file that you need.

Interrupts

AppleTalk needs to have interrupts enabled to function correctly. When interrupts are off, packets cannot be received from or sent to other computers. This will cause network services to stop functioning. One particularly visible aspect of this problem is losing a connection with a file server. It only takes four consecutive missed packets for the workstation to assume the server has shut down or has become unreachable.

Do not leave interrupts disabled any longer than absolutely necessary. Beware that if interrupts are disabled inside a loop, that the effect is multiplied by the number of iterations. Leaving interrupts disabled for just a few microseconds could cause a packet to be missed. Obviously, there are some times when interrupts must be disabled, such as in a critical timing loop for a disk driver.

Interrupts must be on for an incoming packet to be received. Therefore, repeatedly turning interrupts on and off can be just as bad as leaving them off the entire time. For example, if a section of code has interrupts disabled 80% of the time and enabled 20% of the time, you will miss approximately 80% of all incoming packets.

Remember, interrupt handlers (like heartbeat tasks) execute with interrupts off. Keep their run time as short as possible (such as setting a flag for a foreground task to check).

Do not make operating system calls with interrupts disabled. These calls could potentially take long periods of time to complete (for example, a large file read). AppleShare calls will not be able to complete with interrupts disabled.

Multi-user applications

A multi-user application is an application that lets several users access and possibly change some common data at the same time. A multi-user application is usually multi-launch. A typical example is a database program that lets several users view and edit records at the same time. In this case, the read/write protections are applied to individual records instead of the entire file. Doing this requires using some commands specific to AppleShare.

First, you would use the FST_Specific call SpecialOpenFork to open the file (fork). With this call you not only provide the access you want to the file, but the access you will allow others to the file. For example, a database file might be opened for read/write, deny nothing. This way, all users can open the file and read and write to it at the same time. (buffering off).

To prevent one workstation from writing to the file and corrupting information being read or written by another workstation, you use the FST_Specific call ByteRangeLock. It takes an open file refnum, some flags, an offset into the file, and a length. The (length) number of bytes starting at the given offset can be locked or unlocked. When a range of bytes is locked, no other workstation can read or write those bytes; in fact, the same workstation using a different refnum cannot access those bytes. Note that you can lock a range past the EOF of the file, which is necessary when extending the size of the file.

For example, you might want to add a new record to a database. First, you would lock the header of the file and read it in to determine where to place the new record. Then you would lock the range where the new record will be located. Next, update the header to indicate the new record has been allocated, write out the header, and unlock it. Now, write the new record to the range you have locked, and unlock the range.

Remember that you should have locked any range of bytes that you are reading or writing, and that you should re-read a range of bytes if you have unlocked and locked it again. Note that buffering is disabled by default for the SpecialOpenFork call to prevent inconsistencies between the buffer's and the file's contents (with the normal Open call, this is not a problem since no other workstation is allowed access that could cause such an inconsistency).

Appendix C **Apple II AppleShare Compatibility Test Script**

THIS APPENDIX is a test script to be followed for all applications tested for AppleShare compatibility. This is a general feature test script covering only those features common to most applications. The test script tests compatibility only and not whether programs are AppleShare aware.

Note: The AppleShare Compatibility Test Script is under development and is subject to revision. Please submit any suggestions or revisions for it to: AppleShare Compatibility Test Script, M/S 75-3T, Apple Computer, Inc., 20525 Mariani Ave., Cupertino, CA 95014. ■

Introduction

There are ten phases of testing: Installation, Launching, General Operations, File Checkout, Server Alerts, Printing, Macintosh/Apple II interactions, Concurrent Operations, Boundary Conditions, and Playtime. Make sure that you know the application thoroughly before you begin the test phases.

Complete the check list by placing a check in the appropriate section, or a N/A if the test is not applicable to the application you are working on. A "NO" response indicates a bug or script error. If your response indicates a bug, note the bug number(s). If the script is in error or needs an addition or modification, write in "Script Error" on the BUG# line. Add a reference number to your notation and make a corresponding note at the end of the script noting the error, your addition, or modification. The test script will be revised from your note, so please be specific.

Repeat this test script for each configuration described in the AppleShare Compatibility Spread sheet. All operations are to be done with an Apple IIe or IIGS unless otherwise noted.

Please use this script as a jumping off point. At the end of the script is an area to note your own tests as well as suggestions for additional tests. It is important to test beyond the script to cover areas that may have been glanced over or tested only from one angle. When you do expand beyond this document, however, please note all tests and results and note whether you think that you tests should be made a regular part of this script.

Preparation

Before you can begin testing the application(s), you must follow these steps:

1. Install the server CPU, one Macintosh workstation, two Apple II workstations, preferably one Apple IIe workstation and one Apple IIGS workstation, Peek station on network, one LaserWriter and ImageWriter.
2. Set up server for Apple II users following instructions in the Admin guide.
3. Register at least two users in addition to the administrator. User 1 must have the primary group of Student. User 2 must have the primary group of Teacher.
4. Set up [code name] as foreground application on server and capture a LaserWriter and an ImageWriter (II and LQ).
5. Set up user 1 so that default printer is [code name] captured printer.
6. Set up the other user 2 so that default printer is a network printer.
7. Install the Aristotle Menu Management and Menu display programs onto the server following the directions in the Aristotle manual.
8. Install Apple II System Utilities, BASIC System, the operating system, and the Finder (if Apple IIGS workstation is being used in test) onto server.
9. Log onto server from workstation.

Test Script

DATE	
TESTER	
APPLICATION	
DEVELOPER	
VERSION	

SERVER CPU	
SOFTWARE VERSION	
SYSTEM DISK	
HOPS FROM WORKSTATION	
ZONES FROM WORKSTATION	

☐ Check if test run booting off server

☐ Check if test run booting off ws disk

COMMENTS

WORKSTATION 1 CPU	
SOFTWARE VERSION	
SYSTEM DISK	
ROM VERSION	
MEMORY	
SOFTWARE VERSION	
SYSTEM DISK	
ROM VERSION	
MEMORY	
[CODE NAME] VERSION	
S&P VERSION	
SUPER SERIAL CARD SLOT	

1. Installation

1.a. If the application has an install routine, attempt to install application on a server volume.

Was attempt successful?

If attempt was not successful, note error message or type of failure.

YES	NO
BUG#	

1.b. If install was not successful or if application does not have an install routine, copy application to server volume.

Note utility used to copy.

Was attempt successful? (Were all files installed that were supposed to be?)

If attempt was not successful, note error message or type of failure.

YES	NO
BUG#	

If you cannot place the application on a server volume at all, follow the script placing only the document on the server. Set up privileges to the application so that both User 1 and User 2 have access to it.

2. Launching

Section 2.a through 2.c are to be tested either by booting off a workstation disk and launching the program from which the application is to be launched, or by setting the program from which the application is to be launched as the startup application and booting off the server. Note which procedure is to be used in the following test on page one of this script. For information on how to set an application as a startup application, refer to the [Code Name] Admin Manual.

Note: except where noted, it does not matter which user you log on as.

2.a Launching from ProDOS 8

Enter application's prefix (path up to application).

Enter RETURN.

enter application's pathname (path including application).

Enter RETURN

Is launch successful?

Note: if you have trouble launching, check amount of RAM, or set server volume to hardcoded pathname.

YES	NO
BUG#	

2.b Launching from BASIC

Type "PR#3"

Enter RETURN

Type "prefix /" and enter application's prefix (path up to application)

Enter RETURN

Type "-" and enter name of application.

Enter RETURN

Is launch successful?

Note: if you have trouble launching, check amount of RAM, or set server volume to hardcoded pathname.

YES	NO
BUG#	

2.c Launching from IIGS Finder

Run this test only if testing IIGS workstation. Otherwise skip to 2.d.

Double click on application.

Is launch successful?

Note: if you have trouble launching, check amount of RAM, or set server volume to hardcoded pathname.

YES	NO
BUG#	

2.d. Launching from Aristotle Menu Display.

From server Admin program, set user's startup application to Menu Display program.

Log on to server as User 2 and launch Management Program. Follow instructions in Aristotle manual to create a class providing User 1 with access to the application.

Boot off server as User 1.

Select Menu Display

Is launch successful?

Note: if you have trouble launching, check amount of RAM.

YES	NO
BUG#	

3. General Operations

The following tests are geared toward the "typical" application that produces data files and includes basic editing features. If the application you are testing does not include such features, mark N?A in YES box and write in more applicable tests. Applicable tests would include only those that involve network activity.

Before beginning the following tests, launch application in any one of the preceding ways. In subsequent tests of this product, launch using another method.

Note the method you are using to launch application.

3.a Open application and create or open a large document

Are operations successful?

YES	NO
BUG#	

3.b Cut and paste within document or between document in different directories and volumes (both on the server and local)

local).*Are operations successful?*

YES	NO
BUG#	

3.c Save document to server.

Are operations successful?

YES	NO
BUG#	

3.d Saving under different name

3.d.1 Save as different name in same directory.

Is attempt successful?

YES	NO
BUG#	

3.d.2 Save to a server directory to which have complete access.

Is save successful?

YES	NO
BUG#	

3.d.3 Save to a server directory to which you have no see files privilege.

Is save successful and/or message appropriate?

YES	NO
BUG#	

3.d.4 Attempt to save to a server directory to which you have no make changes privilege.

Is save prohibited and message appropriate?

YES	NO
BUG#	

3.e Save while quitting application

Is attempt successful?

YES	NO
BUG#	

3.f Place application in directory to which you have no write access.

Attempt to launch application.

Is launch successful?

YES	NO
BUG#	

3.g Place document in a server directory to which you have no write access.

3.g.1 Attempt to open document

Is save prohibited and message appropriate?

YES	NO
BUG#	

3.g.2 Attempt to save changes to document

Is save prohibited and message appropriate

YES	NO
BUG#	

3.g.3 Close document.

Is operation successful?

YES	NO
BUG#	

3.h Excessive or unnecessary server activity renders an application incompatible.

Is the application compatible in this regard?

Is the application running much slower than locally?

YES	NO
BUG#	

3.i Enter any additional general operations tested and their results:

Operation:	Result:	
	OK	BUG
	BUG#	
	OK	BUG
	BUG#	
	OK	BUG
	BUG#	
	OK	BUG
	BUG#	
	OK	BUG
	BUG#	
	OK	BUG
	BUG#	
	OK	BUG
	BUG#	

4. File Checkout

An application is said to have a file-checkout problem if it allows two or more users to open the same document and save changes to it concurrently, such that they overwrite each other's changes. Acceptable solutions to his problem include not allowing subsequent users to open the document, making subsequent users change the name of their version of the document, or allowing subsequent users to pen the document for read access only.

4.a Place document in a folder to which two users have full access. From each work station launch a local copy of the application.

4.a.1 From work station one, open the document. From work station two, try to open the same document.

If the document cannot be opened, is the message informing you so clear?

YES	NO
BUG#	

If the document cannot be opened from work station two, skip to section 5.

4.a.2 From work station one, make changes to the document and then save. From work station two, make changes to the document and attempt to save.

Are you prevented from saving and instructed to save the file the file under another name?

YES	NO
BUG#	

4.a.3 From both work stations, make changes to the document. Save changes from work station one. Quit form work station two.

Are you prevented from saving and instructed to save the file under another name?

YES	NO
BUG#	

If work station two is told to save the document under a different name, try the following:

4.a.4 From either work station, make some changes, save, make more changes, and save once more.

Are you allowed to save the document without receiving a message that the file has been edited by another user?

YES	NO
BUG#	

4.b Some applications that do not exhibit a file-checkout problem will allow subsequent users to open a document if the first user has only read access to the document. In such a case, subsequent users have only read access to the document as well.

Place the document in a directory to which work station one has only read access and to which work station two has full access. Open the document from work station one.

Attempt to open the document form work station two.

If the attempt is successful, is there a clear message informing the user that they cannot make changes to the document?

YES	NO
BUG#	

5. Sever Alerts

To test how application handle server alerts, initiate server shutdown, then cancel several times during the following operations:

5.a	Idle	OK?	BUG?	#
5.b	Loading application	OK?	BUG?	#
5.c	Reading from server	OK?	BUG?	#
5.d	Writing to server	OK?	BUG?	#
5.e	Cutting and pasting	OK?	BUG?	#
5.f	Printing to a network printer	OK?	BUG?	#
5.g	Printing to [Code Name]	OK?	BUG?	#
5.h	Converting document	OK?	BUG?	#
5.i	Sending data (communication programs)	OK?	BUG?	#
5.j	Receiving data (communication programs)	OK?	BUG?	#
5.k	Closing document	OK?	BUG?	#
5.l	Quitting application	OK?	BUG?	#

6. Printing

6.a Print to a network ImageWriter (II & LQ)

Is attempt successful?

YES	NO
BUG#	

6.b Print to a network LaserWriter (II & NT)

Is attempt successful?

YES	NO
BUG#	

6.c Log on as the same user from two work stations. Begin printing from work station one and change printer selected in Chooser or Admin form the second work station.

Does print job complete successfully?

YES	NO
BUG#	

6.e Complete [Code Name] worksheet and attach to script.

7. Macintosh/Apple II Interactions

7.a From a Macintosh workstation,

7.a.1 Read data last written to server by II	OK?	BUG?	#
7.a.2 Update data last written to server by II	OK?	BUG?	#
7.a.3 Delete data last written to server by II	OK?	BUG?	#

7.b From an Apple II workstation,

7.b.1 Read data last written to server by the Macintosh	OK?	BUG?	#
7.b.2 Update data last written to server by Macintosh	OK?	BUG?	#
7.b.3 Delete last data written to server by Macintosh	OK?	BUG?	#

7.c Conduct various operations with Macintosh/Apple II related translators (such as those provided with Apple File Exchange)

Note operations and results

Operation:	Result:	
	OK	BUG
	BUG#	
	OK	BUG
	BUG#	
	OK	BUG
	BUG#	
	OK	BUG
	BUG#	
	OK	BUG
	BUG#	
	OK	BUG
	BUG#	
	OK	BUG
	BUG#	

8. Concurrent Operations

This phase of testing involves two different workstations trying to work with the same document or application at the same time. Note that concurrent operations differ from simultaneous operations. Some of the following operations are not likely to occur concurrently. Use you judgment and first test those operations that are most likely to occur. Test the remainder only if you have time. Mark the space with OK if you encounter no problem with that step of the script. Otherwise, fill the space with the appropriate bug number(s).

The vertical axis represents work station one, while the horizontal axis represents work station two.

8.a Test the following with an Apple II and work station one and an Apple II as work station two.

Work station one (WS 1) always works with the document. Work station two (WS 2) always works from ProDOS, System Utilities, or BASIC.

WS 1	WS2 – manipulating document					
	COPY	RENAME	DELETE	DENY ACCESS	MOVE	
OPEN						
SAVE						
CUT&PASTE						
CONVERT						
PRINT						
CLOSE						
QUIT						

8.b Test the following with two Apple IIs

Work station one (WS 1) always works with document. Work station two (WS 2) always works form ProDOS, System Utilities, or BASIC.

WS 1	WS2 – manipulating document					
	COPY	RENAME	DELETE	DENY ACCESS	MOVE	
OPEN						
SAVE						
CUT&PASTE						
CONVERT						

PRINT						
CLOSE						
QUIT						

8.c Test the following with a Macintosh as work station one (WS 1) and an Apple II as work station two (WS 2).

WS 1 always works with the document, while WS 2 always works from ProDOS, System Utilities, or BASIC.

WS 1	WS2 – manipulating document					
	COPY	RENAME	DELETE	DENY ACCESS	MOVE	
OPEN						
SAVE						
CUT&PASTE						
CONVERT						
PRINT						
CLOSE						
QUIT						

8.d Test the following with an Apple II as work station one (WS 1) and a Macintosh as work station two (WS 2).

WS 1 always works with the document, while WS 2 always works from ProDOS, System Utilities, or BASIC.

WS 1	WS2 – manipulating document					
	COPY	RENAME	DELETE	DENY ACCESS	MOVE	
OPEN						
SAVE						
CUT&PASTE						
CONVERT						
PRINT						
CLOSE						
QUIT						

9. Boundary Conditions

9.a Perform the following operations running with as little memory as possible:

9.a.1	Load application	OK?	BUG?	#
9.a.2	Read from disk	OK?	BUG?	#
9.a.3	Write to disk	OK?	BUG?	#
9.a.4	Cut and Paste	OK?	BUG?	#
9.a.5	Print to network printer	OK?	BUG?	#
9.a.6	Print to [Code Name]	OK?	BUG?	#
9.a.7	Convert document	OK?	BUG?	#
9.a.8	Close document	OK?	BUG?	#
9.a.9	Quit application	OK?	BUG?	#

What is the least memory with which the application can run? [ ] k

9.b. Perform the following operations with a full server volume

9.b.1	Load application	OK?	BUG?	#
9.b.2	Read from disk	OK?	BUG?	#
9.b.3	Write to disk	OK?	BUG?	#
9.b.4	Cut and Paste	OK?	BUG?	#
9.b.5	Print to network printer	OK?	BUG?	#
9.b.6	Print to [Code Name]	OK?	BUG?	#
9.b.7	Convert document	OK?	BUG?	#
9.b.8	Close document	OK?	BUG?	#
9.b.9	Quit application	OK?	BUG?	#

10. Playtime

Some applications may not conform well to the operations in this test script. This is your chance to exercise those aspects of the application that you feel were not tested enough.

As time allows, play with the application, conducting operations not included in this test script. Some ideas to consider are: More exotic operations, Application-specific operations, Concurrent operations involving more than two work stations, simultaneous (as opposed to concurrent) operations

Operation:	Result:	
	OK	BUG
	BUG#	
	OK	BUG
	BUG#	
	OK	BUG
	BUG#	
	OK	BUG
	BUG#	
	OK	BUG
	BUG#	
	OK	BUG
	BUG#	
	OK	BUG
	BUG#	

- Test should be made whether program has, as it should not do, written to either of its forks (code, data). This may be done by duplicating the program file, funning the tests, then running a file compare program as an additional test. Before the file compare is run, an additional reconfiguration of the program from its reconfiguration menu(s) should be one of the test stops. Program reconfiguration is an operation particularly likely to involve the program writing to itself.
- After installation application files, files installed should be noted. At the end of the test, the directory should be compared to identify any program generated temporary file names. Any fixed temporary file names may indicate an incompatibility.
- Test should include an evaluation of extent of the extent of program segmentation, if any.
- Try to start up over the network from two Apple IIs at the same time. Launch the same application form both work stations at the same time. Do this when logged on as different users, and the same user.
- Concurrency and boundary tests should be done when starting up over the network, if possible.
- Can a file be deleted while you have it open (or being edited)? If so, you have a file checkout problem.
- Does the application reasonably handle the server shutting down (see operations in Server Alerts section)?