




***Advanced
Programmer's
Guide II:
File System***

*Revision 23.0
DOC10056-3LA*




Advanced Programmer's Guide II: File System



.....

Third Edition

William T. Carbonneau



This manual documents the software operation of the PRIMOS operating system on 50 Series computers and their supporting systems and utilities as implemented at Master Disk Revision Level 23.0 (Rev. 23.0).



Prime Computer, Inc., Prime Park, Natick, Massachusetts 01760

The information in this document is subject to change without notice and should not be construed as a commitment by Prime Computer, Inc. Prime Computer, Inc., assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Copyright © 1990 by Prime Computer, Inc. All rights reserved.

PRIME, PR1ME, PRIMOS, and the Prime logo are registered trademarks of Prime Computer, Inc. 50 Series, 400, 750, 850, 2250, 2350, 2450, 2455, 2550, 2655, 2755, 2850, 2950, 4050, 4150, 4450, 6150, 6350, 6450, 6550, 6650, 9650, 9655, 9750, 9755, 9950, 9955, 9955II, Prime INFORMATION CONNECTION, DISCOVER, INFO/BASIC, MIDAS, MIDASPLUS, PERFORM, PERFORMER, PRIFORMA, Prime INFORMATION, PRIME/SNA, INFORM, PRISAM, PRIMAN, PRIMELINK, PRIMIX, PRIMEWORD, PRIMENET, PRIMEWAY, PRODUCER, PRIME TIMER, RINGNET, SIMPLE, Prime INFORMATION/pc, PT25, PT45, PT65, PT200, PT250, and PST 100 are trademarks of Prime Computer, Inc.

Printing History

Preliminary Edition (DOC9229-1LA) January 1985 for Revision 19.4
First Edition (DOC10056-1LA) September 1985 for Revision 19.4.2
Second Edition (DOC10056-2LA) July 1987 for Revision 21.0
Third Edition (DOC10056-3LA) June 1990 for Revision 23.0

Credits

Editorial: Thelma Henner, Mary Skousgaard

Project Development: Glenn Morrow

Technical Support: Julie Cyphers, Sonya Zegarra

Illustration: Mary Easter, Carol Smith, Myron Stein

Production: Judy Gordon



How to Order Technical Documents

To order copies of documents, or to obtain a catalog and price list:

United States Customers

Call Prime Telemarketing,
toll free, at 1-800-343-2533,
Monday through Thursday,
8:30 a.m. to 8:00 p.m. and
Friday, 8:30 a.m. to 6:00 p.m. (EST).

International

Contact your local Prime
subsidiary or distributor.



PRIME SERVICESM

Prime provides the following toll-free number for customers in the United States needing service:


1-800-800-PRIME

For other locations, contact your Prime representative.



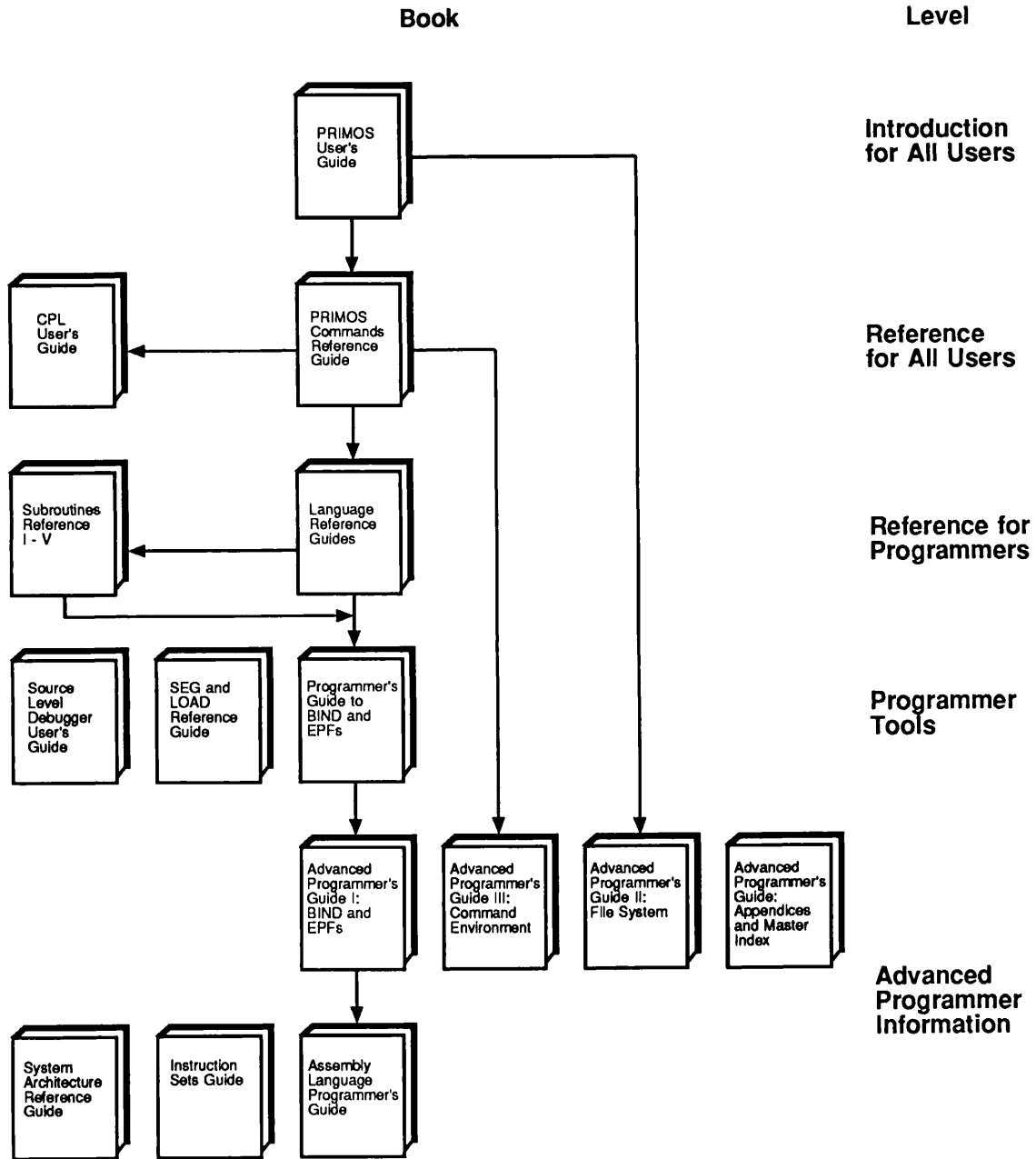
Surveys and Correspondence

Please comment on this manual using the Reader Response Form provided in the back of this book. Address any additional comments on this or other Prime documents to:



Technical Publications Department
Prime Computer, Inc.
500 Old Connecticut Path
Framingham, MA 01701

Reading Path for PRIMOS Documentation



Qpath.D10056.3LA

Contents

■ ■ ■ ■ ■ ■ ■

About This Book

1 What Is a File System?

- Data . . . 1-1
- Storage . . . 1-2
- Objects . . . 1-2
- Procedures . . . 1-3
- File Systems: Summary . . . 1-3

2 The PRIMOS File System

- What Is the PRIMOS File System? . . . 2-1
- The File System Before Rev. 23.0 . . . 2-2
 - Disk Trees on More Than One System . . . 2-3
 - Limitations of Multi-rooted Name Space . . . 2-4
- The Rev. 23.0 File System . . . 2-4
 - The Root Directory . . . 2-5
 - The Singly-rooted File System Directory Structure . . . 2-5
 - The Common File System Name Space . . . 2-7
 - The Name Server . . . 2-7
 - The Global Mount Table . . . 2-7
 - Logical Mounts . . . 2-8
 - Portals . . . 2-8
- PRIMOS File System Objects . . . 2-9
 - Naming and Accessing Objects . . . 2-10
 - Root Directories . . . 2-10
 - Physical Disks . . . 2-13
 - Disk Partitions . . . 2-14
 - Directories . . . 2-15
 - Segment Directories . . . 2-15
 - Access Categories . . . 2-16
 - Files . . . 2-16

3 Accessing the PRIMOS File System

- Object-naming Conventions . . . 3-1
 - Objectnames . . . 3-2
 - Pathnames . . . 3-2
 - How and When Objects Are Named . . . 3-5
 - Access Methods . . . 3-5
- Access Control . . . 3-6
 - Attaching to a File Directory . . . 3-6
 - Access Control Lists . . . 3-7
 - Password Directory . . . 3-8
- How and When Access Is Calculated . . . 3-9
 - Access Calculation Concepts . . . 3-9
 - Access Calculation When Opening Files . . . 3-11
 - Access Calculation When Attaching to Directories . . . 3-11
 - Access Calculation for Other Operations . . . 3-12
- File Units . . . 3-13
 - Information Associated With a File Unit . . . 3-13
- Opening a File . . . 3-16
 - File Unit Number Allocation . . . 3-17
 - File Unit Numbers . . . 3-18
 - File Pointers . . . 3-18
 - Positioning Files . . . 3-19
 - Truncating Files . . . 3-19
- Closing Files . . . 3-19
 - Closing on Normal Program Termination . . . 3-19
 - Closing on Abnormal Program Termination . . . 3-20
- File Attributes . . . 3-20
 - The Date and Time Last Accessed (DTA) Attribute . . . 3-21
 - The Date and Time Created (DTC) Attribute . . . 3-22
 - The Date and Time Last Modified (DTM) Attribute . . . 3-23
 - The Date and Time Last Backed Up (DTB) Attribute . . . 3-24
 - The Read/Write Lock Attribute . . . 3-24
 - The File Type Attribute . . . 3-25
 - The Dumped/Not-dumped Attribute . . . 3-26
 - The Special/Not-special Attribute . . . 3-27
- Quotas . . . 3-27

4 Programmer Interfaces to the File System

- Communicating With the File System . . . 4-1
 - Commands . . . 4-1
 - Command Functions . . . 4-2
 - Subroutine Calls . . . 4-2
 - System Primitives . . . 4-2
 - Arguments and Options . . . 4-3

Attach Points and Access Rights . . .	4-4
Objectnames . . .	4-6
File Units and Attributes . . .	4-7
PRIMOS Responses (Return Codes) . . .	4-8
File System Operations: An Overview . . .	4-9
General Requirements . . .	4-9
Creating Objects . . .	4-10
Opening Objects . . .	4-10
Reading Objects . . .	4-10
Writing Objects . . .	4-11
Deleting Objects . . .	4-11
Access Control to File System Objects . . .	4-12
Attach/ACL Requirements . . .	4-12
Attaching . . .	4-12
Access Control List (ACL) Functions . . .	4-15
Creating File System Objects . . .	4-22
Creating Portals . . .	4-22
Creating File Directories . . .	4-24
Creating Files . . .	4-26
OPENING FILE SYSTEM OBJECTS . . .	4-27
Opening File Directories . . .	4-27
Opening Files . . .	4-29
Reading File System Objects . . .	4-30
Writing File System Objects . . .	4-35
Closing File System Objects . . .	4-37
Deleting File System Objects . . .	4-38

5 Search Rules

Search Rules and Search Lists . . .	5-1
Default Search Lists . . .	5-2
Advantages of Search Rules . . .	5-2
Search Rule Types . . .	5-3
Administrator and System Search Rules . . .	5-3
User-specified Rules . . .	5-4
Search List Types . . .	5-4
User-defined Lists . . .	5-4
ATTACH\$. . .	5-5
COMMAND\$. . .	5-7
INCLUDE\$. . .	5-8
BINARY\$. . .	5-8
ENTRY\$. . .	5-9
Creating and Setting Search Rules . . .	5-9
Creating a Search Rules File . . .	5-9
Setting Search Lists . . .	5-10

Search Rule Keywords . . .	5-12
The <code>-insert</code> Keyword . . .	5-12
The <code>-system</code> Keyword . . .	5-13
The <code>-optional</code> Keyword . . .	5-15
The <code>-added_disks</code> Keyword . . .	5-15
The <code>-public</code> Keyword . . .	5-17
The <code>-static_mode_libraries</code> Keyword . . .	5-17
The <code>-primos_direct_entries</code> Keyword . . .	5-17
The <code>[origin_dir]</code> Keyword . . .	5-18
The <code>[home_dir]</code> Keyword . . .	5-18
The <code>[referencing_dir]</code> Keyword . . .	5-19
Accessing Search Lists . . .	5-19
PRIMOS Command Environment . . .	5-19
CPL Programs . . .	5-20
Program Subroutines . . .	5-20
ATTACH\$ Invoked by Other Search Lists . . .	5-21

6 Attach Points

The Initial Attach Point . . .	6-1
The Home Attach Point . . .	6-3
The Current Attach Point . . .	6-4
Operations That Reset the Current Attach Point . . .	6-5
Functions Used To Manipulate Attach Points . . .	6-7
The AT\$ Subroutine . . .	6-7
The AT\$ABS Subroutine . . .	6-10
The AT\$ANY Subroutine . . .	6-13
The AT\$REL Subroutine . . .	6-16
The AT\$ROOT Subroutine . . .	6-19
The GPATH\$ Subroutine . . .	6-20
The SRCH\$\$ Subroutine . . .	6-23
Questions and Answers About Attach Points . . .	6-25

7 Text Storage and Retrieval

Subroutines for Accessing Files . . .	7-1
Difference Between Variable-length and Fixed-length Record Files . . .	7-2
Variable-length Records . . .	7-3
Fixed-length Records . . .	7-3
Hybrid Approaches . . .	7-4
Maximum Length of a File . . .	7-5
How to Open, Extend, Truncate, and Close Text Files . . .	7-5
Opening a File . . .	7-6
Positioning a File to End-of-file . . .	7-13
Truncating a File . . .	7-16
Closing a File . . .	7-19

- How to Read and Write Variable-length Text Files . . . 7-23
 - The RDLIN\$ and WTLIN\$ Interfaces . . . 7-23
 - Sample Programs Using RDLIN\$ and WTLIN\$. . . 7-27
- How To Read, Write, and Position Fixed-length Files . . . 7-30
 - The PRWF\$\$ Interface . . . 7-30
 - Sample Uses of PRWF\$\$. . . 7-38
- Format of a Variable-length Record File . . . 7-41
- Format of a Fixed-length Record File . . . 7-42
 - Determining the Blocking Factor . . . 7-43
 - Calculating Record Position During Random-access Operations . . . 7-44
- Questions and Answers About Text Files . . . 7-45

8 Data Storage and Retrieval

- File Organization . . . 8-1
- Segment Directories . . . 8-2
 - Subroutines Used to Access Segment Directories . . . 8-2
 - How to Open a Segment Directory . . . 8-3
 - How to Position a Segment Directory . . . 8-9
 - How to Extend a Segment Directory . . . 8-13
 - How to Open a Member File Within a Segment Directory . . . 8-16
 - How to Delete a Member File Within a Segment Directory . . . 8-21
 - Scanning a Segment Directory . . . 8-23
- File Directories . . . 8-28
 - Creating a File Directory . . . 8-28
 - Opening a File Directory . . . 8-32
 - How to Scan a File Directory . . . 8-37
- Reading and Writing Data Files . . . 8-40
- Questions and Answers About Data Files . . . 8-41

9 Access Control Lists (ACLs)

- Subroutines That Manipulate ACLs . . . 9-1
 - Setting Access on Files and Directories . . . 9-1
 - Creating Access Categories . . . 9-2
 - Changing Access to a File System Object . . . 9-6
 - Setting the Access for an Object to That of Another Object . . . 9-6
 - Reading the Access for an Object . . . 9-8
- How Programs Should Parse an ACL . . . 9-10
- Questions and Answers About ACLs . . . 9-10

10 File Attributes

- How to Read the File Attributes of an Object . . . 10-1
 - Example . . . 10-5
- Setting File Attributes . . . 10-7

11 *Disk Quotas*

- Retrieving Information on Disk Space in Use . . . 11-1
 - Retrieving Quota Information for a Directory . . . 11-1
 - Retrieving Quota Information for the MFD . . . 11-3
- Improving Quota System Performance . . . 11-4

12 *Interprocess Communication via the File System*

- General Concepts . . . 12-1
 - File and System Read/Write Locks . . . 12-1
 - Caveats on Using the File System for Interprocess Communication . . . 12-3
- Sample Models of Communication via File System . . . 12-5
 - Multiple Processes Creating File-based Transactions . . . 12-5
 - Multiple Competing Servers Accessing File-based Transactions . . . 12-6
 - Two-process Transaction Management . . . 12-9
 - Multiple Processes Accessing a Database . . . 12-9

Appendix

A *File System Glossary . . . A-1*

Index

About This Book



The *Advanced Programmer's Guide* is a four-volume series that provides technically sophisticated information for systems-level programmers. This series supplements basic reference information found in other PRIMOS[®] manuals.

The books in this series are intended for programmers who are experienced with the PRIMOS operating system and 50 Series™ systems. In addition, you should be experienced in at least one high-level programming language supplied by Prime (preferably PL/I, C, or FORTRAN-77).

The *Advanced Programmer's Guide* series consists of four volumes:

- *Advanced Programmer's Guide I: BIND and EPFs* (DOC10055-2LA)
- *Advanced Programmer's Guide II: File System* (DOC10056-3LA)
- *Advanced Programmer's Guide III: Command Environment* (DOC10057-2LA)
- *Advanced Programmer's Guide: Appendices and Master Index* (DOC10066-4LA)

The four volumes of the *Advanced Programmer's Guide* can be ordered as a set using DCP10171.

Specifics of This Volume

This volume contains detailed technical information about the PRIMOS file system. It describes the systems-level programmer interfaces to the file system, including those used to attach to file system objects, to set access rights on file system objects, and to manipulate text and data files. In addition, it describes disk quotas, inter-process communications, and programmer interfaces to the singly-rooted file system. This volume provides information about Prime subroutines used solely to interact with the file system.



Specifics of the Series

The *Advanced Programmer's Guide* series divides information among the volumes of the set as follows:

- Volume I: BIND and EPFs describes Executable Program Formats (EPFs), including registered EPFs, and describes the EDIT_BINARY binary file editor.
- Volume II: File System (this volume) describes the PRIMOS File System. It provides detailed information about the File Server, access rights, search rules, and data and text manipulation in file system objects.
- Volume III: Command Environment describes how to use EPF initialization routines and how to invoke a user program as a command, subroutine, or function from a user program or from PRIMOS command level.
- Appendices and Master Index provides appendix material applicable to all of the volumes in this document set. It lists the standard error codes used by PRIMOS, along with their messages and meanings. It describes the new features of recent PRIMOS revisions that may be of interest to advanced programmers. Finally, it provides a Master Index to all four volumes of the *Advanced Programmer's Guide* series.

This series describes the lowest-level interfaces supported by PRIMOS and its utilities. It is designed for systems-level programmers who are designing new products, such as language compilers, data management software, electronic mail subsystems, utility packages, and so on. Such products are themselves higher-level interfaces, typically used by other products rather than by end users, and therefore must use some or all of the low-level interfaces described in this series for best results. Most of the information in this series deals with interfaces to PRIMOS that are typically used only in small portions of a product and with overall product design issues that should be considered before coding begins.

Higher-level interfaces not described in this guide include:

- Language-directed I/O
- The applications library (APPLIB)
- The sort packages (VSRTL and MSORTS)
- Data management packages (such as MPLUSLB and PRISAMLIB)
- Other subroutine packages

All the above interfaces are described in other manuals, such as language reference manuals and the *Subroutines Reference* series.

References

Users of this series should be familiar with the *PRIMOS User's Guide* (DOC4130-5LA), which contains information on system use, directory structure, the condition mechanism, CPL files, ACLs, global variables, and how to load and execute files with external subroutines. New information for Rev. 23.0 can be found in the *PRIMOS User's Release Document* (DOC10316-1PA). You should also have an understanding of the architecture of Prime systems, as described in the *50 Series Technical Summary* (DOC6904-2LA).

You should use the *Advanced Programmer's Guide* along with the standard PRIMOS references: the *PRIMOS Commands Reference Guide* (DOC3108-7LA updated by RLN3108-71A) and the five-volume *Subroutines Reference* series:

- *Subroutines Reference I: Using Subroutines* (DOC10080-2LA updated by UPD10080-21A)
- *Subroutines Reference II: File System* (DOC10081-2LA)
- *Subroutines Reference III: Operating System* (DOC10082-2LA)
- *Subroutines Reference IV: Libraries and I/O* (DOC10083-2LA)
- *Subroutines Reference V: Event Synchronization* (DOC10213-1LA updated by UPD10213-11A)

For a complete list of available Prime documentation, consult the *Guide to Prime User Documents*.

Prime Documentation Conventions

The following conventions are used throughout this document. The examples in the table illustrate the uses of these conventions.

<i>Convention</i>	<i>Explanation</i>	<i>Example</i>
Uppercase	In command formats, words in uppercase bold indicate the names of commands, options, statements, and keywords. Enter them in either uppercase or lowercase.	SLIST
Italic	Variables in command formats, text, or messages are indicated by lowercase italic.	LOGIN <i>user-id</i>
Abbreviations	If a command or option has an abbreviation, the abbreviation is placed immediately below the full form.	SET_QUOTA SQ
Brackets	Brackets enclose a list of one or more optional items. Choose none, one, or several of these items.	LD [-BRIEF -SIZE]
Braces	Braces enclose a list of items. Choose one and only one of these items.	CLOSE { <i>filename</i> -ALL }
Braces within brackets	Braces within brackets enclose a list of items. Choose either none or only one of these items; do not choose more than one.	BIND [{ <i>pathname</i> <i>options</i> }]
Monospace	Identifies system output, prompts, messages, and examples.	address connected
Underscore	In examples, user input is underscored but system prompts and output are not.	OK, <u>RESUME MY PROG</u>
Hyphen	Whenever a hyphen appears as the first character of an option, it is a required part of that option.	SPOOL -LIST
Ellipsis	An ellipsis indicates that you have the option of entering several items of the same kind on the command line.	<i>pdev-1</i> [... <i>pdev-n</i>]
Parentheses	In command or statement formats, you must enter parentheses exactly as shown.	DIM <i>array</i> (<i>row</i> , <i>col</i>)

What Is a File System?

1



It is hard to imagine a large corporation, a small business, or even an individual being able to do any business at all without some form of data. Something as simple as an address book is one kind of data that an individual might use. A checkbook is another. Businesses use data in the form of mailing lists, accounts receivable, accounts payable, cash on hand, and many other collections of words and numbers in their daily transactions. In order to use these words and numbers in any efficient and meaningful way, they must be organized in some fashion, and there must be tools by which their owners can manipulate them. The function of a file system is to provide the organization and the tools to store and use information by means of a computer.

Data

The first characteristic of a file system, then, is that it is a collection of **data** — information in the form of letters, digits, and symbols arranged into useful groups of words and numbers. If the groups are put into some fixed sequence, such as a last name, a first name, a middle initial, and a telephone number, each group can be called a **field**. A field is usually designated as either alphanumeric (consisting of a mixture of letters, digits, and symbols) or numeric (consisting mostly of digits, but possibly including a plus or a minus sign, a decimal point, one or more commas, and perhaps a currency symbol). Other kinds of fields, such as pure alphabetic or binary, are recognized by some programming languages.

A **record** is the basic unit upon which most file systems operate. A number of fields can be combined into a structured element known as a **data record**. There are also unstructured records, which consist of strings of alphanumeric information of varying lengths; these are, strictly speaking, also data records, but to distinguish between structured and unstructured records, the unstructured records can be called **text records**. As a programmer, you will be using both kinds of records: you will write programs in the form of text records; your programs will most likely deal with data records.

Storage

The second characteristic of a file system is that its data has been placed in some kind of **storage** from which it can be retrieved when needed. Many forms of storage exist: punched cards, paper tape, magnetic tape, and various forms of magnetic disks. In these chapters we deal only with storage on disks.

Objects

Having a collection of data arranged into fields and records and stored on a disk is a big step toward organizing the data. It is really all that you absolutely need to store and retrieve data. Given a set of commands that the computer understands, you could at this point successively retrieve records until the desired one is found, and then do some kind of operation on it.

But this is a tedious task, and there might be more than one class of records upon which you want to perform different kinds of operations. For example, the telephone number records would serve a purpose different from that of, say, accounting records, and for reasons of efficiency or privacy, it would be useful to keep these two classes of records separate.

A useful file system should be able to segregate different classes of data into different groups, or **objects**, the most basic of which is the **file**. The previous paragraph hinted at the existence of two files, one a list of names and telephone numbers, and the other a list of names and accounting information. A company employee whose job is to do telephone surveys of customers could retrieve their telephone numbers from the first file without having to read and skip, or even being able to see, any of the information about their accounts in the second file.

You can also imagine a second level of segregation, in which files, as well as records, might be grouped together to serve some particular purpose. A company with a nation-wide customer base, for example, would maintain account files of all of its customers, but might want to operate on them on a state-by-state or regional basis. One approach to this task would be to cluster the files for each state or region into another kind of object: a catalog containing the names of the files in the cluster. These objects serve as **directories** to the objects contained in them, and indeed, some file systems, including the PRIMOS file system, call them just that. Directories, along with a suitable language, enable identical actions to be performed on several files by simply addressing the directory that contains them.


File systems provide other kinds of objects, whose purposes are to ease the burden of dealing with large collections of data, controlling access to them, and increasing the efficiency of operating on them. What PRIMOS provides is described in Chapter 4, Programmer Interfaces to the File System. How you as a programmer use them is explained in the remainder of this volume.




Procedures

No matter how sophisticated it may be, data organization is only an idea, useless without some way to implement it, and then to act on the organized data. For these purposes, a set of tools, or **procedures**, is needed. Procedures, written into programs, enable you to create file system objects, write data into them, read data from them, control access to them, and perform other related functions on both the objects themselves and the information contained in them.

File Systems: Summary



No matter how elementary or sophisticated your work is, you need a file system to perform that work. File systems come in many forms, with a variety of capabilities ranging from simple file creation, reading, and writing to the construction of highly complex database with hierarchical structures and intricate access control mechanisms. But the ultimate goals of any file system are simple: to organize data, to enable and simplify access to it, and to exercise control over who can do what to it.



The next three chapters explain the elements of the PRIMOS file system and how they work together to achieve these goals.

The PRIMOS File System

2



This chapter describes the structure and components of the PRIMOS file system. The topics covered include

- What is the PRIMOS file system?
- The file system before Rev. 23.0 (multi-rooted hierarchy)
- The Rev. 23.0 file system
- The singly-rooted hierarchy
- The common file system name space
- PRIMOS file system objects

What Is the PRIMOS File System?

The PRIMOS file system is Prime's implementation of a collection of objects and procedures that let you create a file storage structure. You manipulate this file storage structure in order to fulfill your data storage, access, and security needs.

Each Prime machine has one or more physical disks that store data. Each physical disk is logically divided into one or more sections called disk partitions. For example, one partition would hold the tools for administering the system, another would hold user directories, another would hold data management databases, and so forth. Each disk partition, in turn, is made up of directories. A directory is the logical "file drawer" that holds the files. The files themselves hold the data.

The PRIMOS file system theory and structure is described in more detail in the sections following. The first section following presents a useful review of the file system structure before Rev. 23.0. The remainder of the chapter deals with the PRIMOS file system at Rev. 23.0.

The File System Before Rev. 23.0

Before Rev. 23.0, PRIMOS organized a directory structure like an inverted tree. The pre-Rev. 23.0 directory tree consists of a root (the disk partition), branches (the directories), and leaves (your files — the objects of most of your work with the file system). Each disk partition is the source of a tree whose components have distinct names. See Figure 2-1 for an illustration of a machine with the pre-Rev. 23.0 directory tree structure.

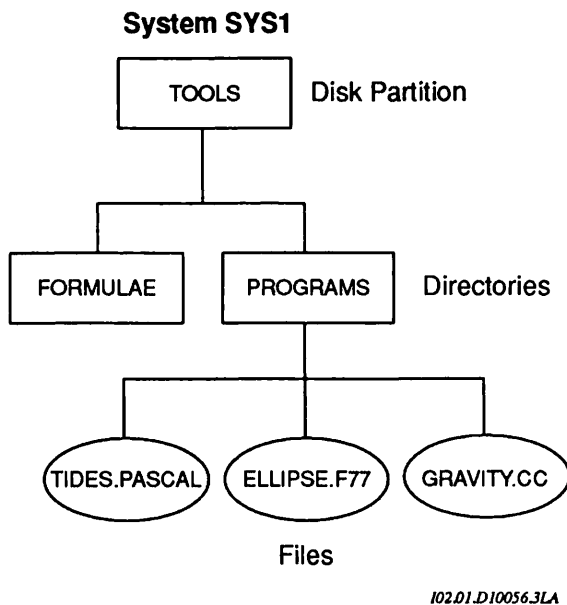


Figure 2-1. Pre-Rev. 23.0 Directory Tree Structure

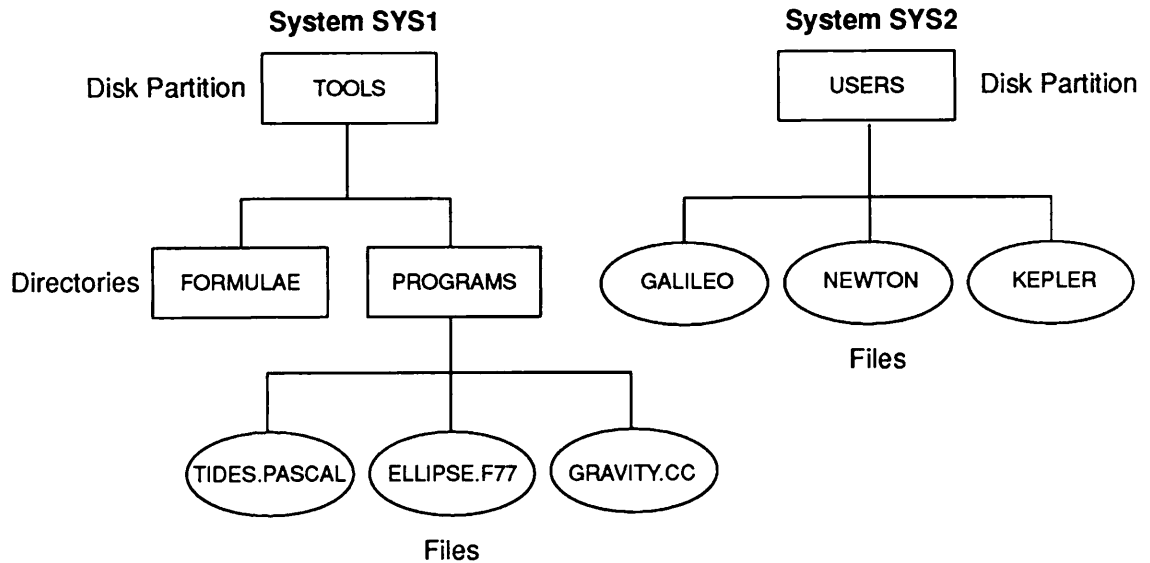
In the above example, the tree hierarchy of one of the disk partitions on System SYS1 is shown. This partition is called <TOOLS> and contains two directories, FORMULAE and PROGRAMS; PROGRAMS contains files. The directories and files that reside under <TOOLS> use its name as a starting point for their own names. This, in turn, determines where these objects are located:

- <TOOLS>FORMULAE
- <TOOLS>PROGRAMS
- <TOOLS>PROGRAMS>TIDES.PASCAL
- <TOOLS>PROGRAMS>ELLIPSE.F77
- <TOOLS>PROGRAMS>GRAVITY.CC

Typically, a system contains more than one partition, each one having a separate starting point with directories and files under it. The name of each object under that disk partition is ultimately qualified by the disk. That is, the pathname of each begins with the root name, and is said to be a **fully-qualified pathname**.

Disk Trees on More Than One System

Many sites have more than one system, or machine, with users having to utilize data on more than one machine. These users must use more than one tree to accomplish their tasks. Consider two different disk tree hierarchies on two different systems. See Figure 2-2 for an illustration of two machines with the pre-Rev. 23.0 directory tree structures.



102.02.D10056.3LA

Figure 2-2. Pre-Rev. 23.0 Directory Tree on Two Machines

The illustration shows a disk tree on two different systems: <USERS> on System SYS2 and <TOOLS> on System SYS1. Each disk is a separate and distinct root (starting point) for the names (and thus the locations) of their subordinate file system objects.

The collection of names of all the file system objects is known as the **file system name space**. The name of every object in the pre-Rev. 23.0 tree hierarchy is defined by its disk partition. Therefore, the type of configuration illustrated in Figure 2-2 is called a **multi-rooted name space**.

Limitations of Multi-rooted Name Space

In a multi-rooted name space, the manner in which you named and referenced every file system object was limited by that object's physical location. Consider another example based on Figure 2-2 in the previous section. Suppose user KEPLER, on System SYS2, has been told to copy a file called <TOOLS>PROGRAMS>ELLIPSE.F77, which resides on System SYS1. However, KEPLER does not know that, even though he can access System SYS1, the remote disk <TOOLS> was never added to his system's local disk table. If he attempts to copy ELLIPSE.F77, the COPY command returns an error:

```
OK, COPY <TOOLS>PROGRAMS>ELLIPSE.F77 *>==  
Not found. Unable to attach ``<TOOLS>PROGRAMS`` (copy)  
ER!
```

In Figure 2-2, System SYS1 and System SYS2 have been networked because users on both machines have common tasks to perform. However, user KEPLER encounters problems performing his tasks because there is no transparent access to file system objects between these two machines. Disks on a pre-Rev. 23.0 system must be administered manually, and added manually. KEPLER must now wait for the remote disk to be added to his system's disk table, or obtain a remote user ID, before he can proceed.

This is symptomatic of the limitations of the pre-Rev. 23.0 file system in a multi-machine environment. Even though your machine is part of a network, tasks involving other machines can be cumbersome because your view of the file system is local, encompassing only your machine.

Also, there is the danger of inadvertent diskname duplication; PRIMOS does not inform you if there are two identical disknames in the same network environment. This means that a fully-qualified pathname may not be unique.

The Rev. 23.0 File System

A number of changes have been made to the PRIMOS file system. These changes allow a collection of 50 Series machines to share what is known as a **common file system name space**. Among the advantages of configuring a common file system name space are

- Administration of disks is made easier, since remote ADDISK commands are no longer needed.
- Distributed applications are easier to build, since pathnames uniquely reference file system objects regardless of which machine the reference came from.

- The limit on the number of disk partitions which can be referenced from a single machine is increased from 238 to 1280.

Your System Administrator might decide not to configure a common file system name space, but in order for PRIMOS to support the new scheme, there are some file system changes that affect everyone at Rev. 23.0. The most notable of these changes is the singly-rooted file system hierarchy. This hierarchy is implemented by means of a root directory and changed pathname syntax and semantics of pathnames.

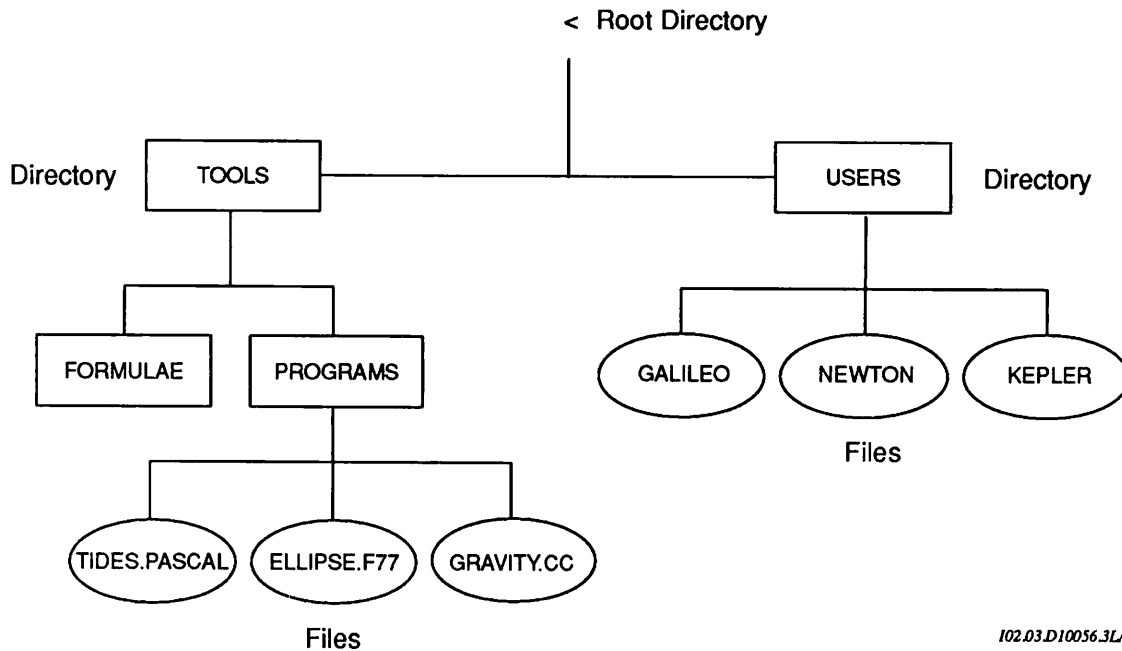
The Root Directory

At Rev. 23.0, the PRIMOS file system has been modified to have a single root directory. This root directory, designated as "<", represents a level higher than the MFD in the file system hierarchy. It is the starting place for interpreting pathnames. The root directory contains only directories, which correspond to the MFDs of local and remote disk partitions. Therefore attaching down from the root directory places a user in the MFD of a specific disk partition.

The root directory has many of the characteristics of other directories, yet is special in a few ways. Like other directories, you can attach to the root and list its contents. However, it cannot be deleted or modified. The only way to add or delete entries in the root is to use the ADDISK and SHUTDN commands, discussed in the *Operator's Guide to System Commands*.

The Singly-Rooted File System Directory Structure

Like the multi-rooted directory structure, the singly-rooted file system directory structure is also organized like a tree: it has a root (the root), branches (the directories), and leaves (the files). See Figure 2-3 for an illustration of this structure.



102.03.D10056.3LA

Figure 2-3. Singly-rooted Directory Tree Structure

The MFD of each of the disk partitions appears as a directory under the root directory. Partitions on one machine or more than one machine are under the root. Examine the contents of the root by issuing the ATTACH and LD commands, as follows:

```
OK, ATTACH <
OK, LD
```

```
< (LU access)
```

```
321 Directories.
```

```

AAAAAA      AAAAAB      AAAAAC      AAAAAD
AAAAAE      AAAAAF      AAAAAG      AAAAAH
AAAAAI      AAAAAJ      AAAAAK      AAAAAL
AAAAAM      AAAAAN      AAAAAO      AAAAAP
AAAAAQ      AAAAAR      AAAAAS      AAAAAT
AAAAAU      AAAAAV      AAAAAW      AAAAAX
AAAAAY      AAAAAZ      BAAAAC      BAAAAD
BAAAAE      BAAAAF      BAAAAG      BAAA AH
BAAA AI     BAAA AJ     BAAA AK     BAAA AL
--More--Q
  
```

```
OK,
```


The Common File System Name Space

At Rev. 23.0, System Administrators can create a common file system name space for a collection of machines. Having a common file system name space means that all disk partitions on a specified collection of machines are visible to every machine in that collection. You can reference file system objects within that collection (provided that you have the proper PRIMENET RFA access and the proper ACL rights). DSM defines the collection of machines which share a common file system name space.

All machines which share the file system name space see a common and complete view of the file system hierarchy, because PRIMOS replicates the root directory on each machine. Since the starting point for interpreting pathnames at Rev. 23.0 is the root directory, having identical root directories on a collection of machines means that fully-qualified pathnames always mean the same thing.

The Name Server

The root directory is replicated by starting up the process server called Name Server, new at Rev. 23.0. The Name Server replicates the root directory among a DSM-defined collection of machines. Your System Administrator creates a common file system name space on a pre-determined collection of machines by using DSM to define which machines are in the same name space, then starting the Name Server on each of those machines.

The foundation for the common file system name space boundaries is the DSM config group. Configuring DSM over the network means dividing it into groups of machines, none of which overlap. Each group has a consistent picture of which machines are in the group. This view of the network is exactly what is needed by the Name Server. Thus, when each Name Server is started, it must consult DSM to determine which other machines have Name Servers it must consult with.

The Global Mount Table

The Rev. 23.0 file system interprets pathnames based on the root directory, not the disk table. Underlying the root directory is a new PRIMOS database which contains the total list of disk partitions and portals which a given machine can reference. (Portals are discussed in the next section.) This database is known as the Global Mount Table (GMT).

For every disk partition or portal in the local file system name space, the GMT lists

- on which machine each disk or portal is located
- where in the common file system name space the disk or portal is “grafted” (its pathname).

To obtain this information, use the LIST_MOUNTS command. This command has a number of options which allow users to relate pathnames to specific disks and systems, machine names to disks, and disk names to system names. For more information about LIST_MOUNTS, see the *PRIMOS User's Release Document* and the *Operator's Guide to System Commands*.

Logical Mounts

At Rev. 23.0, disk partitions have expanded capabilities, thus adding to their flexibility in the common file system name space. It is possible to

- add a disk partition to the root with a name that is up to 32 characters long, for example, <A_REALLY_LONG_DIRECTORY_NAME. This is in keeping with the fact that partitions are treated as directories in the common file system name space.
- graft a disk partition over any existing directory in the tree hierarchy except the MFD. The directory over which the partition is mounted is called the **mount-point directory**, and its contents are thereafter inaccessible until the grafted partition is shut down. One advantage of adding partitions that are subordinate to other partitions is that it is easier to extend the storage capacity of systems whose applications are written to use fully-qualified pathnames. Another advantage is that a directory tree can expand as much as is required.

In both of the above cases, the partition is called a **logical mount**. The System Administrator adds a partition as a logical mount by using a new option, -MOUNT_PATH, of the ADDISK command. The -MOUNT_PATH option is described in the *Operator's Guide to System Commands*, and logical mounts are discussed in more detail in the *System Administrator's Guide, Volume I: System Configuration*.

Portals

At Rev. 23.0, a Network Administrator can partition a collection of Prime machines into one or more common file system name spaces by using the Name Server. Remember that a common file system name space consists of all the disk partitions on all of the machines that are members of the same DSM config group, and that disk partitions are shared by all machines in the common file system name space. To reference a disk partition on a remote machine, however, it is necessary to use a **portal**. A portal acts as a gateway between name spaces, allowing you to transparently perform operations upon file system objects in other name spaces (provided that the ACLs are set correctly and that the target partition is not labeled as private).

A portal is a directory which has been transformed so that references to it are redirected to a directory on a remote machine. There are two types of portals, and each type affects where the target of the portal is:

- A **root-directed portal**, the more powerful of the two types, redirects references to the root directory of another machine. The remote machine can be any machine outside your file system name space as long as it resides on the network.
- A **disk-directed portal** redirects references to the MFD of a specified disk. This type of portal is defined primarily for compatibility with earlier revisions of PRIMOS which do not have a root directory.

Portals can only be created by using the `ADD_PORTAL` command on the supervisor terminal. The command must specify a fully-qualified pathname of an existing directory which is to be transformed into a portal, and also must specify the node name of a machine which is the target of the portal. Remove a portal from a directory by using the `REMOVE_PORTAL` command at the supervisor terminal. Programmers with privileged access can add or remove portals using the `NAM$AD_PORTAL` and `NAM$RM_PORTAL` subroutines, respectively. For more information about these subroutines, see Chapter 4, Programmer Interfaces to the File System. For more information about `ADD_PORTAL` and `REMOVE_PORTAL`, see the *Operator's Guide to System Commands*.

PRIMOS File System Objects

The remaining portion of this chapter describes in detail the objects that make up the PRIMOS file system. The next sections describe

- Naming and accessing objects
- Root directories
- Physical disks
- Disk partitions
- Directories
- Segment directories
- Access categories
- Files

Naming and Accessing Objects

Each object must have a name so that it can be uniquely identified. The person who creates an object assigns it a name. Disk partitions and directories are usually assigned names by a System Administrator. You, the programmer, assign your own names to objects that belong to you: user directories, segment directories, access categories, and files.

Once you start using the tree structure, you will want to store some data in it, and reuse the data that you have stored. The PRIMOS file system supports three **access methods**, or ways of reading and writing data: the Sequential Access Method (SAM), the Direct Access Method (DAM), and the Contiguous Access Method (CAM)

You will want to have some control not only over who has access to your files, but also over what kinds of things those who do have access can do to your files. Other users who share your system will want to exercise the same control over theirs. Typically, you might want all members of your department to be able to read your files, a select few to be able to change them or add to them, and you alone to be able to create and delete them. The PRIMOS file system gives you a variety of **access control** tools to establish whatever degree of control you wish over any, some, or all of the objects that belong to you. These tools involve **user identifications** and a set of **permissions**, or **access rights**, which together make up **Access Control Lists (ACLs)**. An older form of access control, the directory password, is still supported, but its use is declining in favor of the access control list.

Note The tree structure is made up of **file system objects**. An object is a collection of data that has its own name, the name by which you can refer to the object when you want to do something with it. The paragraphs that follow describe each of these objects in detail.

Root Directories

At the top of the file system tree structure is the **root directory**. This is the starting point for referencing all file system objects within that tree. The root is represented by the **root character**, <, which is the lesser-than mathematical symbol, also called the right-angle symbol. A **fully-qualified pathname** is an absolutely unique name across the common file system name space for those systems on which the Name Server is running.

You can reference the root by itself using the ATTACH command or the AT\$ subroutine.

OK, ATTACH <
OK, LAC

ACL protecting "<Current directory>":
\$REST: LU

OK, LD

< (LU access)

321 Directories.

ADDNTS	ADMFNL	AQQLID	ARDFNU
ARDHIV	AUX1	BAVFL	BOMDOD
BTD1	DADDB2	DKLIST	DMQFNW
DOMAND	DOMMSW	DOMQRJ	DORF
DQGR11	DQGR12	DQGR13	DQGR14
DQGR15	DQGR16	DQGR17	DQGR18
DQGR19	DQGR20	DQGR21	DQUDOD
DQUGR1	DQUGR4	DQUGR5	DQUGR6
DQUGR7	DROM1	DSF&G	DSWINT
DSWQRJ	DSWTST	DATA1	DATA2
DATA3	DEGRQ	DEGRQ2	DEGRQ5
DBMSRD	DDMDFV	DIDT2X	DISDVR
DMDQRF	DMGSRD	DMGTST	DMROAM
DMYGRQ	DMYSRD	DMYTST	DRAF/D
DRAF/I	DRAFTW	DSAG	DUMQM
FDMS36	FMS1	FNGDB	FNGDB1
FNGDB2	FNGDB3	FNGDB4	FNU1
FNU2	FNU3	FNUDA2	FNUDB
FNVIR1	FNVIR2	FNVIR3	FNVIR4

--More--q

OK,

A leading < character in a pathname signifies a fully-qualified pathname. The root symbol is acceptable by itself as a valid pathname, but it is not a valid filename. You can generally use the root directory as any other directory. Note that only the ADDISK and SHUTDN commands can change the contents of the root.

Root Syntax Change: The new root syntax satisfies the pre-Rev. 23.0 syntax rules as a fully-qualified pathname. However, the pathname <TOOLS>PROGRAMS>ELLIPSE.RUN is interpreted differently at Rev. 23.0:

Pre-Rev. 23.0 The pathname is interpreted as the file ELLIPSE.RUN which resides in the top-level directory PROGRAMS, which in turn resides in the disk partition with the name TOOLS.

Rev. 23.0 The pathname is interpreted as the file ELLIPSE.RUN which resides in the directory PROGRAMS, which in turn resides in the directory TOOLS, which resides in the root directory (<).

Root Characteristics: The root directory has special characteristics because it is both a replicated directory and it is also the starting point of pathname interpretation in the common file system name space. These characteristics are summarized as follows:

ACL protection The ACLs on the root directory itself are \$REST:LU; no other objects may be created in the root since it is the starting point of the common file system name space. The only way to modify the contents of the root is with either the ADDISK or SHUTDN commands.

Open/close operations You cannot directly open the root; a SRCH\$\$ < operation fails since < is not a valid filename. The root can be indirectly opened by calling SRSFX\$ < or by attaching to the root using AT\$ROOT and using SRSFX\$ with a zero-length name. The root can be closed by calling SRSFX\$, CLO\$FU, or CLO\$FN.

Entries in the Root Directory: All root-directory entries are directories which were created when the disk partition was added with the ADDISK command. These root-directory entries generally take and maintain the attributes of the MFD of that disk partition when the ADDISK operation was done. The exceptions to this rule are the following:

- The DTM attribute is set to the time that the ADDISK command was issued, regardless of the DTM setting on the MFD.
- The DTA is not set.
- The ACL which protects the MFD and the access which is computed from it are determined dynamically. Therefore, the ACL and computed access associated with the root-directory entry are always the same as the ACL and computed access for the associated MFD.
- If the root-directory entry is created as a result of a remote ADDISK command (which is only if the Name Server is not started), the attributes contain either default or non-applicable values.

Physical Disks

Disk partitions are configured on physical disks. PRIMOS supports three kinds of physical disks: Cartridge Module Devices (CMDs), Fixed Media Devices (FMDs), and Storage Module Devices (SMDs). Each of these is available in several storage capacities; the total range of *usable* storage space provided by the three types is from approximately 30 usable megabytes for the smallest CMD to approximately 759 usable megabytes for the largest FMD.

Storage space is divided into **surfaces**, **tracks** (or **cylinders**), and **sectors**, the numbers and capacities of which are physical properties of the devices, and vary from one type of device to another. All of the devices and their capacities and physical characteristics are described in detail in the *Operator's Guide to File System Maintenance*.

Each physical disk, when it is first introduced to the PRIMOS operating system, is initialized, or **formatted**, by a System Administrator or System Operator, using the MAKE command (described in the *Operator's Guide to File System Maintenance*). One function of formatting is to create, on the physical disk, one or more logical disks, or partitions, by defining the starting surface number and the number of surfaces that make up the partition. (A partition may not be smaller than one surface.)

Some physical disks can contain a single partition, while others either are required to be or operate more efficiently when configured into two or more partitions. The actual number of partitions that a physical disk ultimately contains depends both on its physical characteristics and on the uses to which it is put.

The *System Administrator's Guide, Volume I: System Configuration*, discusses the considerations involved in the planning and execution of disk partitioning.

Another function of formatting is to create a file known as the Disk Record Availability Table (DSKRAT), which enables the file system to keep track of which physical records contain data and which physical records are available to have data stored in them. Each physical record on the disk is represented in this file by one bit, whose value is 0 if the record is in use, and 1 if the record is available. The DSKRAT file typically occupies several contiguous physical records, starting at track 0, sector 2, on the first surface on the disk. The DSKRAT file has the same name as the disk partition.

Note A physical record is not the same as the data or text records mentioned earlier; these are called **logical records**. Unless otherwise noted, the term **record** in this book refers to logical records.

Another function of formatting is to provide the disk with a **bootstrap file** (named BOOT). This file contains machine-executable instructions that initiate the loading of the PRIMOS operating system, enabling PRIMOS to be loaded and started from any disk connected to the computer system. The bootstrap file

consists of a single physical record, located at track 0, sector 0 on the first surface of the disk.

During formatting, the MAKE program may detect a "bad" sector, that is, a sector having a flaw that makes it impossible to record data into that sector reliably. When this happens, MAKE creates a file called the **badspot file** (named BADSPT) in which are recorded the locations of any such sectors that it encounters. The file system refers to this file in order to avoid attempts to write data to unreliable sectors.

The DSKRAT, BADSPT, and bootstrap files are largely invisible and of little direct interest to you as a programmer. The file system uses DSKRAT and BADSPT automatically, and the bootstrap record is normally invoked only by the System Operator.

The final object that formatting creates is the Master File Directory (MFD), beginning at track 0, sector 1, on the first surface of the partition.

Disk Partitions

A **disk partition**, which may also be referred to as a logical disk or a volume, is a logical section of the physical disk which is demarcated for a specific use. It normally appears directly below the root in the root hierarchy and in this event is called a **root entry**; however, logical partitions may also be grafted onto the tree hierarchy at a point in the tree that is lower than the root.

The logical disk partition is treated as a directory by the Rev. 23.0 file system. You can get a list of disk partitions by using the LIST_MOUNTS command, or using the NAM\$L_GMT subroutine, described in the next section.

You can reference a disk partition (root entry) using the ATTACH command or the AT\$ or AT\$ANY subroutines:

```
OK, ATTACH <DIRONE>  
OK, LD
```

```
<DIRONE>MFD (LUR access)
```

```
3 Files.
```

```
BADSPT          BOOT          DIRONE
```

```
2 Directories.
```

```
MFD             RPTS
```

```
OK,
```


Notice that the pathname <DIRONE returned the pathname <DIRONE>MFD;
The two pathnames are synonymous.

Directories

A **directory** is a collection of file system objects assembled for a common purpose.

System or Project Administrators often assign directories to individual users as **origin directories**, although lower-level directories may be assigned just as well for this purpose. (An origin directory is the starting point for a user to access the file system.) The objects that can be immediately subordinate to a user directory are lower-level directories, segment directories, access categories, or files. In addition to pointing to the objects it contains, a directory also includes access control and quota information for them.

Not all directories are assigned as origin directories. On the command partition, for example, a number of directories immediately under the root entries may contain objects such as command files, records of system usage, and other kinds of data that are related to system operation.

Any directory that is one or more levels below a root-directory entry is a **lower-level directory**, or simply a directory. Directories can point to the same kinds of objects that root entries can point to, including more lower-level directories. Directories can be nested to many levels.

While the nesting level limit depends on such factors as the physical capacity of the disk on which the directories reside and on quotas that may have been established on their superior directories, the real determining factor may be the length of the fully-qualified pathname, which is limited to 256 characters. User access to and interaction with a lower-level directory whose pathname contains more than 256 characters is uncertain, because the pathname is truncated. (Pathnames are explained in the section entitled Object-naming Conventions in Chapter 3.)

Segment Directories

The directories described so far all fall into a class known as **file directories**. There is another class known as a **segment directory**, used primarily to contain program segments created by the PRIMOS SEG command, and multiple-index files such as those created by the MIDASPLUS or Prime INFORMATION subsystems.

Segment directories can be contained in file directories just as any other file system object can. But they can point only to *numbered* data files and segment directories, and cannot contain the *names* of lower-level directories or other objects such as data files or access categories. Their main function is to increase the efficiency of certain utility and application programs through the use of

numbered, rather than named, objects. Once the identifying number of an object is made known to PRIMOS, it is more efficient to locate and operate on than is an object identified by a pathname or a filename.

You can create a segment directory explicitly from a terminal or use PRIMOS subroutines, expressly designed for this purpose, in any program that is intended to manipulate segmented files. You can see the evidence of a segment directory's creation by inspecting the contents of the file directory that contains it, but its actual creation is transparent to you as you sit at your terminal.

Refer to Chapter 8, Data Storage and Retrieval, for further information on segment directories and to *Subroutines Reference II: File System*, and to the *SEG and LOAD Reference Guide* for information on segmented programs.

Access Categories

An **access category** is a directory entry that contains an access control list. When you specify that a certain set of users have specific rights to operate on one of your file system objects, that list of users and rights (the ACL) takes up space in the directory that contains the object. If a number of objects require the same list, creating that list for each individual object becomes wasteful, and it is useful to be able to specify this common list by defining it once and having it reside in only one place. The function of the access category is to contain the list; the access to each object can then be set by referring to the name of the access category.

The subject of access control and ACLs is explained in more detail in the section entitled Access Control in Chapter 3 of this manual.

Files

A **file** is an object that contains a collection of user data. In this broad sense, any file system object can be thought of as a file, since all objects presumably contain information useful to their users. A root entry, for instance, has information that its user, the file system, uses in its search for file system objects. But a file, from the point of view of the user, contains no pointers to further subordinate objects; it is a leaf at the end of a branch in the tree structure.

There are **system files** and **user files**. System files are created by PRIMOS or its administrators and operators for use by the operating system. Some of them can be read by users for purposes such as listing users on the system and getting status information of various kinds. But because of the access controls usually applied to them and their directories, few system files, if any, can be changed or deleted by the ordinary user.

User files, on the other hand, are created by you and other users to fulfill the needs of your application programs. You normally create structured data files by running your application programs, or text files by using a text editor or word

processing application. You can control access to your files as tightly or as loosely as you wish to satisfy your security needs and those of any group(s) to which you may belong.

Accessing the PRIMOS File System

3

.....

This chapter describes the concepts you the programmer need to know in order to access the PRIMOS file system. Topics covered include

- Object–naming conventions
- Pathnames
- Access control

Object–naming Conventions

Every directory, access category, and file must have an identification that is unique within the entire collection of objects known to the file system. This requirement appears, at first glance, to place a heavy burden on you — that of knowing about the name of every existing object any time you want to assign a name to a new object. But PRIMOS eases this burden in much the same way as a mailing address enables the Postal Service to locate a particular John Smith: by using a hierarchical access path to John Smith through a state, city, street name, and house number. It is this access path that is unique, even though some of the individual components may not be.

The mailing address is interpreted by reading geographical elements in a specific order, from the most inclusive to the least inclusive. The file system's access path is formed and interpreted in precisely the same way, by combining the names of file system objects in order, from the most to the least inclusive. The resulting string of names (plus some separators to show where one name ends and another begins) is called a **pathname**, and, for the file system, it is only this pathname that must be unique.

Thus, the only uniqueness requirement you must satisfy is that, within a given directory, each object must have a unique name. This is the same as saying that in a given city there can be only one Washington Street (but there could be a Washington Street in every city in the United States).

Objectnames

An **objectname** is a string of up to 32 characters selected from the following set:

letters (A through Z)

digits (0 through 9)

special characters `_ # $ - . * & /`

An **objectname** cannot begin with a digit or contain any spaces. Also, you should avoid names beginning with `_`, `-`, `&`, and `$`, because they can cause confusion in certain commands and syntaxes. You can use the underscore (`_`) to represent a space if you want your **objectname** to consist of two or more words. Use the period (`.`) for separating **objectname** components.

An **objectname** can consist of one or more **components**. When there are two or more components in an **objectname**, each is separated from the next by a period. Components can be used for whatever purpose you wish, such as to identify several objects as being related to each other in some way. As a programmer, you use components as **suffixes** to source-text filenames to identify the language used in writing your programs (for example, `.FTN` for FORTRAN programs, or `.CBL` for COBOL programs). PRIMOS provides subroutines whose functions are to manipulate suffixes. Refer to the *PRIMOS User's Guide* for an explanation of components and for a list of suffixes that Prime software recognizes.

Although the file system allows up to 16 components in an **objectname**, two or three are usually sufficient for most practical applications. In any case, remember that an **objectname**, including all components and their periods, cannot be more than 32 characters long.

Pathnames

A **pathname** is a string of **objectnames** representing the access path that the file system follows to get to a specific object. There are several kinds of **pathnames**, detailed in the following paragraphs.

Fully-qualified Pathname: From the file system's point of view, an object's **pathname** contains the name of each directory level that must be crossed to get to the desired object. Such a **pathname** is called a **fully-qualified pathname**.

The fully-qualified **pathname** begins with the root directory, represented by the lesser-than mathematical symbol (`<`). Following the root is the disk partition, also called the root-entry directory, since the Rev. 23.0 file system treats disk partitions directly below the root as *directories*. After the disk partition, you

typically find the names of directories progressively narrower in scope until you reach the one containing the desired object. The absolute pathname ends with the name of an object. A pathname cannot be more than 256 characters long.

The name of the root is the < symbol, and each subsequent objectname is separated from the next by a > symbol. Following is a typical absolute pathname:

<DIRECTORY>DIRECTORY> . . . >OBJECTNAME

Relative Pathname: As a terminal operator or a programmer, you often need to supply only part of an absolute pathname: the part that follows the name of the directory you are currently working in. This kind of pathname is called a **relative pathname**; it is relative to the directory you are in. It can be used because the file system keeps track of the elements of the absolute pathname that precede and include the name of this directory. Most commands that you invoke from your terminal, as well as many of the file system subroutines you write into your programs, allow you to use relative pathnames.

You use a relative pathname whenever you want to work on an object that is subordinate to the directory you are currently in, but not immediately subordinate to it; that is, when one or more directory levels exist between the one you are in and the object you want to address. The form of a relative pathname is

*>LOWER-LEVEL_DIR> . . . >OBJECTNAME

Here the asterisk (*) represents the part of the pathname that the file system “remembers,” and when it is combined with the elements that you supply explicitly, the result is an absolute pathname that leads from the root to the object. The part of the pathname represented by the asterisk is called the **home**, or **working**, directory pathname; the directory itself is the home, or working, directory. If your home directory is BRANCH1, the home directory pathname represented by the asterisk is <FOREST>BEECH>BRANCH1. The part of the pathname that you supply after the asterisk to get to the file LEAF6 in directory BRANCH8 is >BRANCH8>LEAF6, giving the following relative pathname:

*>BRANCH8>LEAF6

This, in turn, is interpreted by the file system as the absolute pathname:

<FOREST>BEECH>BRANCH1>BRANCH8>LEAF6

Simple Pathname: When the object you want to address is immediately contained in your home directory, you can use an even simpler form of pathname, known as a **simple pathname**. A simple pathname consists of only the name of the object you want to work with; it does not contain any > symbols. Objectname, entryname, and simple filename are terms used synonymously with

simple pathname. If, as in the last example, your home directory is BRANCH1, and you want to do some operation on the directory BRANCH8, you can use the simple pathname BRANCH8.

Note There is an exception to the interpretation of a simple name when you use the ATTACH command. If, using the example above, you attempt to attach to BRANCH8 by issuing the command

ATTACH BRANCH8

the ATTACH command interprets BRANCH8 as an unqualified, or full, pathname (described below) rather than the simple pathname of the subordinate directory BRANCH8. The result is that PRIMOS searches for a top-level directory of that name, and in all likelihood will fail to find it. To attach to the lower-level directory BRANCH8, you would use a relative pathname:

ATTACH *>BRANCH8

Unqualified Pathname: An unqualified pathname is one in which the first element that you specify is a top-level directory. For example, an unqualified pathname might be

BEECH>BRANCH1>TWIG4

An unqualified pathname is a partial pathname which is completed when the ATTACH\$ search rule finds the first element of the pathname. The file system searches all of the active disk partitions that are visible to your system to find the first occurrence of that first element. If your system is part of a common file system name space or a part of a network, all visible disk partitions on all of the active systems are searched. Local disks are searched first, in order of logical disk number, and then remote disks are searched in the same manner. This can take some time. (You can limit the scope of such a search, or change the order in which disk partitions are searched, by modifying the ATTACH\$ search list. Search lists are described in Chapter 5, Search Rules.) The search ends when the first top-level directory with the specified name is found. That top-level directory, and any intervening lower-level directories specified in the pathname, are then followed to the desired object.

Note With the advent of the common file system name space, the optimum setting of system search rules to reduce search time becomes even more important: the disks that are used most frequently by users on your system should head the search list. If you feel that your system search rules could be more efficient, contact your System Administrator, or refer to the *System Administrator's Guide, Volume I: System Configuration*.

There are three points you must understand about a file system search using an unqualified pathname:

- Once a top-level directory with the specified name is found on any disk, the search terminates.
- If the desired object and all subsequent directories specified in the pathname exist under that directory, PRIMOS performs whatever operation you requested on the object.
- The implication of this file system search method is that, if you want to use a full pathname and be sure of finding the object you want to operate on, all of the objects named in the pathname must exist within the directory that begins the pathname, and the directory that begins the pathname must be unique in your name space (and on any other systems that may share a network with your system).

How and When Objects Are Named

You assign a name to a file system object when you create it; how you create it depends on the kind of object you are creating.

A text file such as a memo or a source program is normally created by using an editor program, and is named by specifying a filename the first time you ask the editor to store it. Different editors have different ways of doing this, documented in their respective manuals and user's guides; storage commands usually take the form of a FILE, STORE, SAVE, or WRITE.

Application-related data files are usually created by a user program that executes an **open file** subroutine. If it does not find the name of the file it is asked to open, the subroutine creates the file and assigns the given name to it. The subroutine call also contains information as to the type of file to be created, and whether it is to be opened for input, output, or both.

File directories, lower-level directories, and access categories can be created either by executing a subroutine in a user program or by using a PRIMOS command at a terminal.

Segment directories are created by various application programs that manipulate segmented files. User programs can call subroutines to create segment directories.

Access Methods

PRIMOS provides three means of file access: the Sequential Access Method (SAM), the Direct Access Method (DAM), and the Contiguous Access Method (CAM). In these access methods, the file appears as a linear array of words indexed by a current position pointer.

A SAM file enables your program to read or write a number of halfwords beginning at the pointer, which is advanced as the halfwords are read or written. File system subroutines enable you to position the pointer anywhere within an open file, and to read and write data sequentially from that point. File data can be transferred anywhere in the addressing range. When a file is closed and reopened, the pointer is automatically returned to the beginning of the file.

A DAM file also appears to be a linear array of halfwords. This method, however, has faster access times in positioning operations, since PRIMOS keeps an index to allow fast random positioning. Subroutine calls to manipulate SAM and DAM files are identical.

A CAM file contains groups of 2048-byte records that are contiguous; that is, the records in a CAM file are not fragmented across the disk. The groups of contiguous records are called **extents** and are indexed by that file's **extent map** and it occupies the first record in the file. Within each extent, the records are ordered sequentially. CAM files use fewer pointers than do DAM files, and access time is much faster since the records are not fragmented. CAM files do, however, require additional memory since one extent map resides in memory for each open CAM file.

Access Control

Two requirements must be met before your program can operate on a file system object:

- Your program must be attached to the file directory that contains the desired object, and, so that this can happen,
- Your program's user must have at least Use access to that directory

Attaching to a File Directory

You can attach your program to the directory containing the desired object in one of two ways:

- Explicitly, if you invoke the ATTACH command specifying the pathname of the file directory before you invoke your program
- Implicitly, if you do not explicitly attach to the file directory, but supply any form of pathname other than a simple objectname when you invoke your program

When you explicitly attach to a file directory by using the ATTACH command, that directory becomes your home directory. You can then invoke your programs

using simple objectnames as arguments; your programs will locate their target objects provided they are immediately contained in that directory.

For example, if you write a program called COUNT to count the number of lines in a text file, and install it in the directory MYDIR.MEMOS, one way you can invoke it is

**ATTACH MYDIR.MEMOS
RESUME COUNT CHARITY**

Since the COUNT program was invoked with the simple objectname CHARITY as its argument, COUNT looks in the home directory MYDIR.MEMOS, established by the ATTACH command, for the file CHARITY.

You do not always need to attach explicitly to a file directory before invoking your programs; they can still operate on objects outside the home directory if you supply the object's relative, full, or absolute pathname rather than its simple name. Taking the COUNT program again as an example, and still assuming the same home directory, you could invoke it in the following way:

RESUME COUNT INIT_DIR>LOGIN.CPL

For this invocation, COUNT has to go to a directory INIT_DIR, outside the home directory, to locate the file LOGIN.CPL. To do this, it attaches temporarily to the outside directory by means of a **current attach point**; the target directory is called the **current directory**.

If you want to enable your programs to operate in both of the ways shown in these examples, you must use the SRSFX\$ subroutine, which is capable of searching for objects outside as well as within the home directory. The SRCH\$\$ subroutine can search only in the home directory; if you use it in a program, and the target object is outside your home directory, you must attach to the directory containing the object before invoking the program, as in the first example.

The intent of the current attach point is that the attachment is in effect only for the duration of the program's execution. When the program terminates, the attach point should revert to the home directory. This is especially important if the program does not terminate normally; in order to provide a consistent result in cases of abnormal termination, most Prime software resets the current attach point to the home directory whether it terminates normally or abnormally.

Attach points and the subroutines that manipulate them are described in more detail in Chapter 6, Attach Points.

Access Control Lists

As stated earlier, users must have access to all directory levels leading to the objects they are working on, as well as to the objects themselves. The means by

which you as a programmer are given access, and by which you can control access, to the various directories and files involved in your daily work are clearly explained in the *PRIMOS User's Guide*. As a programmer of utilities and applications for other users, however, you need to be aware of the kinds of things your programs can and should do to enable those users to control access to the objects these programs create and use.

PRIMOS provides a set of subroutines that you can write into your programs to enable them to manipulate access control lists (ACLs) in precisely the same way as you can by issuing ACL-related commands at your terminal. For example, if you write a program that constructs a database for a group of users, it is particularly useful for that program to be able to establish a database ACL for that group of users at the time the database is created. Or, consider a utility program that creates new files at various times, all of which should be identically protected. Using subroutine calls, this program can create an access category to which each new file is linked when it is created.

An access category, while it takes more disk space than a single access control list (about as much as two average ACLs), saves disk space when the same ACL is to protect more than two objects; this is because any number of objects can be linked to the access category once it is created. Your program can make these links when it creates its files, after it checks to see whether the access category already exists.

You can use an access category to synchronize the access to multiple objects when rights are added or removed from the access category's ACL: whenever a new right is added or an old right is removed, the change applies to all objects protected by the access category, removing the need to update each object's ACL individually.

The access rights that you can assign to your own file system objects (using PRIMOS commands) and that your programs can assign to their objects (using access control subroutines) are all fully described in the *PRIMOS User's Guide*. You can specify ALL to include OPDALURWX. (If some future rev. of PRIMOS supports new access rights, you will not get them automatically when you read in your file that has been assigned ALL. You will have to reassign ALL or add the new rights individually.)

Access control subroutines can deal with both individual users and groups of users. Your System or Project Administrator can define a user group (whose groupname begins with a period, such as .DBUSER) consisting of the user identifications of all of the users of a particular utility or application program. Your programs can use the access control subroutines to grant or deny access to these groups as well as to individuals.

Password Directory

In an older form of file access control, PRIMOS allows a limited set of access rights to be specified on a per-file basis. A file directory can be given an owner

password and a **non-owner** password and a set of rights for each: R (read), W (write), and D (delete). This form of protection is giving way to the more comprehensive ACL mechanism, and will not be further described in this book. Details can be found in the *PRIMOS User's Guide*, as can procedures by which you can convert the older form to the ACL form.

How and When Access Is Calculated

In most situations, users need not be concerned about when access is actually calculated by PRIMOS. However, there are some subtleties of the ACL mechanism that the advanced user should be aware of. This section discusses

- Access calculation concepts
- Access calculation when opening files
- Access calculation when attaching to directories
- Access calculation for other operations

Access Calculation Concepts

For a given file system operation, there are two times that relate directly to the ACL mechanism:

- When access is read
- When access is used

For the most part, reading and using occur at the same time. A sample case is the deletion of a file. When you delete a file, PRIMOS first reads the access for that file, and then it uses that access to determine whether or not you may delete the file.

When you attach to a directory, however, the access is read once. It is then used immediately to determine whether or not you may attach to the directory. If you are allowed to attach, PRIMOS remembers the access it read for the directory. Subsequent operations within and upon that directory may reuse the access that PRIMOS read when you first attached. Therefore, if you attach to a directory, and then change the access for that directory, you will find that for certain operations the access change has not taken effect. The access information read for a home or current directory is not discarded until you attach away from the directory.

The following example illustrates an effect of this behavior.

```
OK, ATTACH COGENT
OK, CREATE SENOR
OK, ATTACH COGENT>SENOR
OK, LIST_ACCESS
```

```
"<Current directory>" protected by default ACL (from "<X1>COGENT"):
    COGENT:  ALL
    $REST:  LUR
```

```
OK, LIST_ACCESS COGENT>SENOR
```

```
"COGENT>SENOR" protected by default ACL (from "<X1>COGENT"):
    COGENT:  ALL
    $REST:  LUR
```

```
OK, LD
```

```
<X1>COGENT>SENOR (ALL access)
1 record in this directory, 1 total record out of quota of 0.
```

```
No entries selected.
```

```
OK, SET_ACCESS COGENT>SENOR COGENT:U -NO_QUERY
OK, LIST_ACCESS
```

```
ACL protecting "<Current directory>":
    COGENT:  ALL
    $REST:  LUR
```

```
OK, LIST_ACCESS COGENT>SENOR
```

```
ACL protecting "COGENT>SENOR":
    COGENT:  U
    $REST:  NONE
```

```
OK, LD
```

```
<X1>COGENT>SENOR (ALL access)
1 record in this directory, 1 total record out of quota of 0.
```

```
No entries selected.
```

```
OK, ATTACH COGENT>SENOR
OK, LD
```

```
Insufficient access rights. (current_directory) (ld)
ER!
```

In this example, LIST_ACCESS commands are invoked at different times to illustrate the difference between the home directory and the same directory when referenced explicitly by pathname. In the first two invocations, LIST_ACCESS reports the same access when the directory is referenced as the home directory and when it is referenced by pathname.

Then, without changing the home attach point, you set the access to the home directory so that you have only Use access. Among other things, this removes List access from the ACL on the SENOR directory.

At this point, the third LIST_ACCESS command on the home directory shows that you still have ALL access to SENOR. A fourth LIST_ACCESS command on the same directory (using the pathname) reports that you have only Use access. This discrepancy is illustrated further by the fact that you can still type LD and see the directory contents (or lack thereof).

However, when you reestablish SENOR as the home attach point, PRIMOS reads the new ACL for this directory. This results in your having only Use access to the home directory, which prevents you from examining the directory contents using LD. It is when you attach again to the lower-level directory that the new ACL takes effect.

Similarly, the new ACL takes effect for any other users that attach to the directory, but not for users who were already attached to the directory when the ACL on it was reset.

Access Calculation When Opening Files

When opening a file or segment directory, the access is read and used when the open operation first takes place. The access is not used again during read or write operations. The access is used if a change-access operation is performed (by using the SRCH\$\$ subroutine with the K\$CACC key). However, the access is not read again in this case. Therefore, once a file is open on a file unit, changing the access of the file does not affect any operations performed on that file unit up until the time that file unit is closed. (See the File Units section, following, for an explanation of file units.)

Access Calculation When Attaching to Directories

When you attach to a directory, as either a home or a current directory, PRIMOS reads and uses the access on the directory during the attach operation. Subsequent operations on the home or current directory use the access without reading it again, as illustrated earlier. However, subsequent operations on the same directory when the name of the directory is specified causes PRIMOS to read the access for the directory to check the access rights for those operations. Once PRIMOS has read the access for the directory, it does not update any access it has already read for the origin, home, or current directories.

The following example illustrates these points.

```
OK, ATTACH COGENT
OK, CREATE SENOR
OK, ATTACH COGENT>SENOR
OK, ED
  INPUT
  A TEST FILE.
```

```
EDIT
```

```
FILE TEST FILE
OK, LIST ACCESS TEST FILE
```

```
"TEST_FILE" protected by default ACL (from "<X1>COGENT"):
      COGENT:  ALL
      $REST:   LUR
```

```
OK, SET ACCESS COGENT>SENOR COGENT:ALURW
OK, LIST ACCESS TEST FILE
```

```
"TEST_FILE" protected by default ACL (from "<X1>COGENT"):
      COGENT:  ALL
      $REST:   LUR
```

```
OK, LIST ACCESS COGENT>SENOR>TEST_FILE
```

```
"COGENT>SENOR>TEST_FILE" protected by default ACL
  (from "<X1>COGENT>SENOR"):
      COGENT:  ALURW
      $REST:   NONE
```

```
OK, DELETE COGENT>SENOR>TEST_FILE
```

```
Insufficient access rights. Unable to delete "COGENT>SENOR>TEST_FILE"
(delete)
```

```
OK, DELETE TEST_FILE
```

```
OK,
```

Here, an attempt to delete a file by pathname fails because the access on its parent directory denies Delete access to the user. However, because the user was attached to the parent directory before the access was changed to deny Delete access, deleting the file as a member of the home directory succeeds.

Access Calculation for Other Operations

Aside from opening files and attaching to directories, most file system operations cause PRIMOS either to use the access for the current directory or to read and use the access for the appropriate file system object just once. For example, renaming a file causes PRIMOS to use the access for the current directory and make certain that both Delete and Add rights are granted.

File Units

A **file unit** is an open channel to a file, a segment directory, or a file directory. Through this channel, your programs read data from and write data to a file system object. Associated with a file unit is a **file unit number**, that is, a numeric pseudonym for the object's name. This number is assigned either by the program (static allocation) or by PRIMOS (dynamic allocation) when the program opens the file (see File Unit Number Allocation, later in this section). It uniquely identifies the file unit for a particular process (user).

Generally speaking, your program performs the following operations to operate on a file system object:

1. **Opens the file:** establishes an open file unit and assigns a file unit number.
2. **Accesses the file:** the open file unit enables operations on the file.
3. **Closes the file:** revokes access to the file.

Information Associated With a File Unit

As described previously, a file unit identifies an open file system object. Internally, PRIMOS maintains information on each open file unit.

Current Object Position: The **current object position** points to the location in the file system object at which the next data read or write begins. For files, the position points to a particular halfword in the file. For segment and file directories, the position points to a particular entry in the directory.

The current object position is adjusted automatically by PRIMOS as data is read from or written to an object. In addition, your program may change the current object position without reading or writing data by using the PRWF\$\$ subroutine, described in Chapter 7, Text Storage and Retrieval.

For files, the current object position is always between 0 and the end-of-file location of the file, or end of file, inclusive. The end-of-file location is the same value as the number of halfwords in the file; when a file is first created, the end-of-file location is 0.

To append new data to the end of an existing file, first position the file unit to the end-of-file location, which represents the position of the next halfword to be appended to the file. (If you do not know the end-of-file location, simply position to the largest possible halfword number, 2147483647. Although PRIMOS returns an error code of E\$EOF to indicate that the end of file has been reached, PRIMOS sets the current object position to the end-of-file location.)

At the end-of-file position, writing data to the file automatically extends the file as the data is written; an attempt to read data at this point returns the error code E\$EOF (end of file).

Open Mode: The open mode determines what operations are valid for an open file unit. A read operation requires the file unit to be open for reading; a write operation requires the file unit to be open for writing; both operations are valid if the file unit is open for both reading and writing. Your program sets the open mode when it first opens a file. Your program can open a file for reading, for writing, or for both reading and writing. To do this, the user running your program must have the corresponding access to the target object. For files and segment directories, the required access is Read, Write, or both Read and Write, to match the actions for which they are opened; for file directories, which are open only for reading, List access is required.

A special open mode, known as **virtual memory file access read (VMFA-read)**, also exists. The PRIMOS executable program format (EPF) mechanism uses VMFA-read to map an EPF into virtual memory from the disk. A file unit open for VMFA-read cannot be read or written by a program.

When your program tries to open a file unit to an object for a specific action such as writing, another file unit may already be open to that object for the same purpose. In such cases, PRIMOS checks the open mode requested by your program against the read/write lock then in effect for the object. Your program's open request is rejected if the lock specifies that only one user at a time can do what the open request intends to do, and

- Another user is already using the object for that purpose, or
- Your program has already opened the object on another file unit for the same purpose.

See the section entitled **The Read/Write Lock Attribute**, later in this chapter for the meanings of the possible values for the lock.

Your program can change the open mode of a file if the new open mode does not conflict with the access or read/write lock controls described above. The CH\$MOD subroutine, described in *Subroutines Reference II: File System*, performs this function.

Object Type: The type of the object open on a file unit determines what kinds of operations are valid on that file unit. **Object types** include

- SAM, DAM, and CAM files, for which most operations (except directory operations) are valid, such as data read and data write
- SAM and DAM segment directories, for which only segment directory operations are valid, such as position to segment directory member and delete segment directory member
- File directories, for which only file directory operations are valid, such as read next directory entry and read named directory entry

Access categories cannot be opened on a file unit; they are restricted in size, so they are read and written in single operations and do not require an associated file unit.

If your program attempts an operation that conflicts with the object type, PRIMOS returns one of several error codes:

- E\$DIRE (Operation illegal on directory), indicating an attempt to perform an operation valid only for SAM or DAM files on a segment or file directory
- E\$NTSD (Not a segment directory), indicating an attempt to perform an operation valid only for segment directories on a file or file directory
- E\$NTUD (Not a top-level directory), indicating an attempt to perform an operation valid only for file directories on a file or on a segment directory

Because these object types are all opened in the same way, these errors are returned only when your program attempts to perform the invalid operation, typically after opening the object. To enable your program to detect an inappropriate object type earlier, have it check the type value returned by the subroutine it calls to open the object. If the type value is not appropriate to the intended operations, your program should close the file unit and report an error.

Object Modified: An object-modified flag is initially reset when a file unit is first opened (before the object is modified). When the first data write is performed on the object, this flag is set (after the object has been modified).

When the file unit is later closed, PRIMOS uses the object-modified flag to determine whether the date and time last modified (dtm) field for the object should be updated. If the flag is not set, PRIMOS does not update the dtm field. Therefore, simply opening a file for writing and then closing the file does not cause the dtm field to be updated. (The date and time last accessed field *is* set under this and other circumstances described later in this chapter.) A program must actually write data to the file and then close the file to update the dtm field.

Disk Shut Down: A disk-shut-down flag is initially reset (meaning the disk is not shut down) when a file unit is first opened. If a disk partition is shut down by the System Administrator or System Operator (by using the SHUTDN command), the disk-shut-down flags for all file units open to objects on that disk are set (meaning the disk is shut down). After that, any attempt by a program or user to continue performing operations on an affected file unit is rejected with the error code E\$SHDN (disk has been shut down).

Calculated Access to Object: When your program opens an object, PRIMOS calculates the user's access to the file to make sure that the user can operate on the file. PRIMOS records the resulting summary of the user's access to the file in the information for the corresponding file unit. A later attempt by your program to change the open mode of the file is checked against this copy of the user's access, not against the current access on the object itself (which may have changed since the file unit was opened).

Read/Write Lock: PRIMOS records the read/write lock of an open file unit in the information for that file unit so that it can quickly determine whether record-level locking for writes is necessary. If at least two file units are open to the same object for writing, or one is open for reading and another is open for writing, PRIMOS must ensure that simultaneous operations on those file units result in predictable behavior. Because such a situation is permissible only when the read/write lock is set to an appropriate value, PRIMOS checks the read/write lock for the file unit to determine how careful it must be in guarding against simultaneous access during a read or write. The more permissive the read/write lock setting, the more care PRIMOS has to take, and the lower the performance of each read or write operation will be.

Opening a File

Your program may open a file for reading only, for writing only, for both reading and writing, or for VMFA-read (EPFs only). If your program opens a file for reading only, your program can read the file, but cannot change the file. If your program opens a file for writing only, your program can write the file, but cannot read the file.

To open a file, your program calls one of many system subroutines, described in Chapter 7, Text Storage and Retrieval, and Chapter 8, Data Storage and Retrieval. Each subroutine provides different functionality for opening a file, but they all provide the following services.

- Search the specified file directory (if a pathname is specified) or the current directory (if a simple objectname is specified) to see whether the requested filename is there.
- Create the file if the filename is not present and your program is opening the file for writing or for both reading and writing. If the filename is not present, and your program is opening the file for reading only, these subroutines return a “not found” indicator.
- Determine a file unit number. The file unit number is the only identifier PRIMOS uses for transferring data to and from the file.
- Set up tables and initialize buffers in the operating system.

If your program opens a file for writing only, or for reading and writing, your program may change that file. If the system subroutine creates a new file at the time of opening, no information is contained in the file.

Because open-for-write files are subject to alteration (deliberate or accidental), your program should keep files closed except when they are being used. Open files absorb system resources; they may also be unavailable to other users. However, frequent open and close operations also absorb system resources;

therefore, try to balance your program's use of files so that open and close operations are infrequent without resulting in file units being open but inactive for long periods of time.

When the user is communicating with the file structure through one of the standard Prime translator or utility programs, files are referred to by name only. PRIMOS, or your program, handles the details of opening or closing files and assigning file units. For example, the user can enter an external command such as ED FILE1, which loads and starts the text editor and takes care of the details of assigning the file FILE1 to an available unit for reading or writing.

File Unit Number Allocation

PRIMOS allows two ways of allocating file unit numbers:

- Dynamic allocation
- Static allocation

Dynamic allocation allows a program to leave to PRIMOS the task of selecting an available file unit number. When opening a file, a program specifies dynamic file unit allocation, and PRIMOS returns to the program the file unit number it has assigned to the open file. The program then uses this file unit number when reading or writing the file.

Static allocation is performed by a program. When opening a file, a program passes the file unit number to PRIMOS. If the specified file unit is already in use, PRIMOS rejects the attempt to reuse the file unit; otherwise, PRIMOS uses the program-defined file unit number to read or write the file.

Dynamic allocation is the recommended method for most programs. Its advantages are as follows:

- You do not have to worry about different parts of your program having conflicting file unit number requirements.
- Your program can call another program that also uses dynamic unit allocation without causing file unit number conflicts.
- A very large number of file units (32761) are available when using dynamic allocation, whereas static allocation allows a maximum of 126 file units open simultaneously for a given user.
- Your program is guaranteed exclusive use of file units.

Static allocation offers very few advantages; these rarely outweigh any of the advantages of dynamic allocation:

- You can design several programs that are to run together as a package so that they use agreed-upon statically allocated file unit numbers; thus, these programs do not have to pass dynamically allocated file unit numbers back and forth to each other.
- Your program can use a numerical constant as the file unit number, rather than requiring the use of a variable.
- Prime translators do treat certain file unit numbers specially (when enabled using the `-ALLOW_PRECONNECTION` option), so your program may use these file unit numbers if it invokes Prime translators.

File Unit Numbers

File unit numbers 1 through 127 may be specified for static allocation by your program. File units 128 through 32761 are returned by PRIMOS only when your program requests dynamic unit allocation. Your program cannot specify a file unit number between 128 and 32761 (inclusive) when opening a file system object.

Unit -4 is the command output file unit. Your program should not read data from or write data to this file unit. Your program may read the current object position of this file unit, or use `GPATH$` to obtain the full pathname of the command output file.

Unit -1 is the current directory; unit -2 is the home directory; unit -3 is the origin directory. These three units are usually open to the corresponding directories. You may use this knowledge to perform certain operations efficiently. For example, to read the directory entries in the user's origin directory, your program can simply call `DIR$RD` using the `K$INIT` key the first time for file unit -3. It does not have to attach to the origin directory (thus preserving the current attach point) or to open the origin directory for reading (thus saving time and a file unit).

File Pointers

Once your program has opened a file, a file pointer is associated with the file unit. To understand how the file pointer works, imagine that the halfwords in a file are serially numbered beginning at halfword number 0. The file pointer is the number of the next halfword to be processed in a file. It identifies the point at which data are read from and written to the file. As your program reads and writes halfwords, the associated file pointer is incremented once for each halfword read or written. If your program reads a line of text, for example, the file pointer is positioned, after the read, to the beginning of the next line of text in the file.

Positioning Files

Your program can move the file pointer backward and forward within a file without moving any data. This is called **positioning a file**, and is described in more detail in Chapter 7, Text Storage and Retrieval. The value of a file pointer is called the **position of the file**. Positioning a file to its beginning is often called **rewinding a file**.

Truncating Files

Your program can shorten a file by **truncating** it. When your program truncates a file, the part of the file that is located at or beyond the file pointer is eliminated from the file, and an end-of-file mark is placed at the pointer position. If the file pointer is positioned at the beginning of the file, all of the information in the file is removed, but the filename remains in the file directory. If the file pointer is positioned at the end of the file, the truncation has no effect.

PRIMOS handles the returning of disk space occupied by truncated records to the free record pool on the disk.

Many programs truncate a text file just before closing it if they have written new information to the file. Because text files are typically variable-length record files, as described in Chapter 7, Text Storage and Retrieval, they are usually written from beginning to end; even if only one line in a file is changed, the entire file is rewritten in case the new line is longer or shorter than the line it replaces. In the process of rewriting an entire file, a program may write a new version that is shorter than the old version. Truncating the file ensures that old data is not left at the end of the new file.

Closing Files

Your program should always close a file before terminating execution, whether termination is normal or abnormal. Closing files is described in more detail in Chapters 7, Text Storage and Retrieval and 8, Data Storage and Retrieval.

Closing on Normal Program Termination

Your program may close a file unit, also referred to as closing a file, when it finishes its processing of the file. When your program does this, the file unit number and the corresponding table areas in the operating system are “cleaned up” and released for reuse by another program or user.

Closing on Abnormal Program Termination

When control returns to PRIMOS by way of an error condition, files are not normally closed. To provide this functionality in your program, have your program close any file units it opened when it detects a fatal error. (Of course, your program should still report the original error; be careful to separate error code variables used to clean up after an error from error code variables used to detect original errors.)

You may also choose to have your program make an on-unit for many error conditions, as described in the *Subroutines Reference III: Operating System*. If one of these conditions occurs while your program is running, your program can close any file units it has opened and then continue to signal the error condition. Typically, this is done for the QUIT\$ condition, signaled when the user types CONTROL-P or BREAK.

Note, however, that although closing file units upon recognition of the QUIT\$ condition has advantages, a distinct disadvantage is that the user cannot restart your program by issuing the START command. If the user attempts this, the program continues executing where it was stopped until it attempts to use one of the closed file units. At this point, an error indicator is returned to the program.

File Attributes

PRIMOS maintains a set of **file attributes** for every file, segment directory, file directory, and access category on disk. The file attributes of a file system object can be read and written by a user program that has sufficient access to the parent directory of the target object. File system attributes include

- The date and time the object was created
- The date and time the object was last accessed
- The date and time the object was last modified
- The date and time the object was last backed up
- The read/write lock of the object
- The file type (which once established can only be read)
- The dumped/not dumped state of the object
- The special/not special state of the object (which is set at disk initialization and can only be read)

Note The date and time created (dtc) and date and time last accessed (dta) attributes may appear in directory entries beginning at PRIMOS Rev. 20.0. These expanded entries are accessed through the use of a hash table. At Rev. 20.0, MAKE creates all directories as hashed ACL directories unless an option is specified that creates a pre-Rev. 20.0 disk. A Rev. 20.0 system can use pre-Rev. 20.0 disks, as can a pre-Rev. 20.0 system. A system running pre-Rev. 20.0 PRIMOS can not use local Rev. 20.0 disks, but it can use remote Rev. 20.0 disks.

The Date and Time Last Accessed (DTA) Attribute

The date and time last accessed (dta) attribute of a system object or its parent is modified under various circumstances as depicted below.

Table 3-1.

	<i>Object DTA Modified?</i>	<i>Parent DTA Modified?</i>
Close an open entry (from read or write)	Y	N
Segment directory subfile	N	Y
After read from write-protected disk	N	N
Write attribute		
dump	N	N
dtm	N	N
dtb	N	N
dtc*	N	Y
dta*	N	N
other (delete switch, protection, rwlock, logical type, truncated bit)	N	Y
Read any attribute	N	Y
Tape backup (MAGSAV)	Y	N
Tape restore (MAGRST – Set to time of restore)	Y	N
Size	N	Y
Remote size	N	Y
Pre-Rev. 20.0 system operating on Rev. 20.0 hashed directory	Y	N
Remote backup (MAGSAV)	Y	N

Table 3-1. (continued)

	<i>Object DTA Modified?</i>	<i>Parent DTA Modified?</i>
Pre-Rev. 20.0 system operating on Rev. 20.0 hashed directory	N	N
Pre-Rev. 20.0 system operating on Rev. 20.0 hashed directory	N	N

Note * dta and dtc can be set only by members of the user group named .BACKUP\$. Backups performed by members of this group are recorded in the date and time last backed up (dtb) attribute.

Format of the Date and Time Last Accessed Attribute: The format of the dta attribute of a file system object is declared in PL/I as follows:

```
dcl 1 dta,
  2 date,
    3 year bit(7), /* Starting in year 1900. */
    3 month bit(4), /* January is month 1. */
    3 day bit(5), /* The first day of the month is day 1. */
  2 time fixed bin(15); /* (Seconds since midnight)/4. */
```

As shown in this declaration, the dta attribute occupies one fullword, or two halfwords. The first halfword is organized as follows:

YYYYYYMMMMDDDD

Here, YYYYYY is the year minus 1900, MMMM is the month (January is month 1), and DDDDD is the day of the month.

The second halfword is the number of seconds past midnight divided by four. The remainder portion of the result of the division is discarded. Therefore, the granularity of the dta field is four seconds.

The Date and Time Created (DTC) Attribute

The date and time created (dtc) attribute contains the date and time that a file system object was created.

Format of the Date and Time Created Attribute: The format of the dtc attribute of a file system object is the same as that for the date and time last accessed attribute.

The Date and Time Last Modified (DTM) Attribute

Whenever a change occurs in the file system data or structure, the date and time last modified (dtm) attribute of the file system object involved is set to the current date and time. User programs may use the dtm attribute of file system objects to determine when the objects were most recently modified.

User programs may also change the dtm attribute of a file system object to any date and time.

How PRIMOS Sets the Date and Time Last Modified Attribute: The dtm attribute of a file system object is set depending upon the object type, as shown below.

<i>Type</i>	<i>DTM Attribute Set</i>
File	When the file is first created, and whenever the file is closed after data in the file has been modified or after the file has been truncated. (The dtm attribute of a file is not changed when any other attributes of the file are changed.)
Segment directory	When the segment directory is first created, and after the segment directory is closed when any of its members have been created, deleted, modified, truncated, or renamed, or when its size is changed.
File directory	When the directory is first created, when one of its members is created, deleted, or renamed, or when certain attributes of one of its members are changed by a user program. Changes to all attributes except the dumped bit, the date and time last modified, and the date and time last backed up cause the updating of a parent directory's dtm field. The parent directory's dtm field is also updated when the access control for one of its members is changed.
Access category	When the access category is first created, or when its contents are changed. Changing the contents of an access category does not, however, update the date and time last modified field of any objects protected by that access category.

The purpose of the dtm attribute is to record the change of any file system data or structure somewhere in the file system itself. Thus, creating a new file sets the dtm attribute for both the file and its parent directory. Subsequently deleting the file also updates the dtm attribute for its parent directory. Although the net result may be that the contents of the directory are unchanged, the recent dtm attribute of the parent directory is an indicator that activity has taken place within the directory.

Format of the Date and Time Last Modified Attribute: The format of the dtm attribute of a file system object is the same as that for the date and time last accessed attribute.

The Date and Time Last Backed Up (DTB) Attribute

The date and time last backed up (dtb) attribute contains the date and time that a file system object was last backed up by a member of the BACKUP\$ group.

Format of the Date and Time Last Backed Up Attribute: The format of the dtb attribute of a file system object is the same as that for the date and time last accessed attribute.

The Read/Write Lock Attribute

One of the responsibilities of the PRIMOS file system is to ensure against attempts by several user processes to read and write one file simultaneously. For example, if user FRED opens a file for reading and writing, user BARNEY is unable to open the file until user FRED has closed it.

Some applications require this restriction to be lifted. For example, an application might require several users to have a file open for writing at the same time. The PRIMOS file system allows this to be specified via a read/write lock attribute.

The Nature of the Read/Write Lock Attribute: Every segment directory and file has a read/write lock attribute. File directories and access categories do not have them, since PRIMOS is entirely responsible for synchronizing updates to these objects.

A file is protected against concurrent access by its read/write lock. The read/write lock attribute for a file is checked every time a user opens the file for reading, writing, or both reading and writing. In addition, a check is made to see if the file is already open for reading and/or writing. Depending on the results of these two checks, the attempt to open the file may be rejected with the error code E\$FIUS (File in use).

Even if only one user is accessing a file, that user may receive a file-in-use error if he or she attempts to open the file twice. PRIMOS does not distinguish between two different processes attempting to open a file and one process attempting to open a file on different file units. For example, if a user attempts to open one file for writing on two different file units, the second attempt to open the file may fail.

Segment Directories and the Read/Write Attribute: The read/write lock attribute for a segment directory affects not only the segment directory itself, but also serves as the read/write lock for all of its members since segment directory members have no attributes of their own (except for file type).

However, PRIMOS still distinguishes between the segment directory and each of its members when it is called upon to open the directory or its members. Therefore, two users may have two different files within one segment directory open for writing at the same time, whereas an attempt by a user to open a segment directory member file that is already open may meet with failure.

The Format of the Read/Write Lock Attribute: The format of a read/write lock attribute is as follows:

```
dcl rwlock bit(2);
```

The four possible values for a read/write lock attribute are detailed in Table 3–2.

Table 3–2. Values for a Read/Write Attribute

<i>Value</i>	<i>Keyword</i>	<i>Meaning</i>
0	SYS	Use the system–wide default. The system default is set via the RWLOCK configuration directive, which is described in the <i>System Administrator's Guide, Volume 1: System Configuration</i> . Normally, the default is 1, corresponding to a file read/write lock of 1, or EXCL (described below). However, the system–wide read/write lock may be 0, meaning only 1 reader or 1 writer may have a file open at a time. The other possible value for a system–wide read/write lock is 3, corresponding to a file read/write lock of 2, or UPDT (described below).
1	EXCL	Exclusive control; <i>n</i> readers or 1 writer. This allows multiple processes to read a single file at a time, unless the file is being written. If the file is being written, no other user may open the file.
2	UPDT	Update control; <i>n</i> readers and 1 writer. This allows multiple processes to read a single file at a time even while it is being written by one process. It still prevents more than one process from writing to the same file at the same time. This setting is useful for command output (COMOUTPUT) files, for example.
3	NONE	No control; <i>n</i> readers and <i>m</i> writers. This provides no locking on a file at all. Using this setting is not recommended, as it decreases the performance of the file system, and can result in damage to your files.

The File Type Attribute

Every object in the PRIMOS file system has a file type. File types include the following:

- Sequential access method file (SAM)
- Direct access method file (DAM)
- Sequential access segment directory (SEGSAM)
- Access category (ACAT)
- Contiguous access method file (CAM)

The file type of an object is determined only when the object is created. It cannot be changed afterwards without deleting and recreating the object.

The file type of an object can be read by a user program along with other file system attributes. The file type attribute is declared as follows:

```
dcl type bit(8);
```

The seven possible values, and their corresponding keywords, are

<i>Keyword</i>	<i>Value</i>
SAM	0
DAM	1
SEGSAM	2
SEGDAM	3
Directory	4
ACAT	6
CAM	7 (ROAM files only)

Notice that file type value 5 is not defined. A value of 5, and any other undefined value, should be treated as an unrecognized file type. Prime reserves the right to use any or all of these undefined values.

The Dumped/Not-dumped Attribute

For backup service, the file system provides a **dumped bit** for all file system objects except access categories. The file system resets this bit whenever the corresponding object is modified. A backup utility can read the dumped bit to determine whether to make a backup copy of the object. If the dumped bit is reset, the utility can then make a backup copy, and set the dumped bit on for the object.

The dumped bit for a file system object is reset (turned off) whenever the date and time last modified attribute for the object is updated. Similarly, if a file is

deleted or renamed, the dumped bit of the parent directory is reset when the dtm attribute of the parent directory is updated.

Dumped Bits for Directories: When a file is modified, the resetting of dumped bits is not performed on all of the directories that intervene between the file and the MFD. Therefore, a backup program must walk through the entire contents of a directory, sensing the dumped bits for all of its members, before deciding that no recent modifications have been made to its members.

Dumped Bits for Segment Directories: File attributes exist only for members of file directories. Therefore, when a file within a segment directory is modified, the resetting of the dumped bit occurs on the parent segment directory, and not on the file, because the parent directory is a member of a file directory, and the individual files are not.

Therefore, only the top-level segment directory dumped bit need be tested to determine whether the contents of the segment directory have changed.

A corollary is that if the dumped bit for a segment directory is reset, the entire segment directory must be backed up, even if only one member of the segment directory has been modified.

The Special/Not-special Attribute

User programs that read directory entries may find the **special bit** useful. PRIMOS sets this bit on for all of the special files when it creates a new disk partition. Special files include the MFD, the BOOT file, the BADSPT file (if it exists), and the record allocation table for the disk partition (which has the name of the disk partition as its objectname).

PRIMOS does not allow user programs to change the special bit for a file system object, nor does it allow objects with the special bit set to 1 to be deleted.

Special files exist only in the MFD for a disk partition.

Quotas

PRIMOS allows you to set quotas on your directories and lower-level directories under certain conditions. Your programs can also make use of quota manipulation subroutines to do this. Quotas are expressed in terms of numbers of physical disk records, and must be assigned carefully if they are to be meaningful and useful.

Detailed explanations of quotas and their settings can be found in the *System Administrator's Guide, Volume I: System Configuration*, and in the *PRIMOS User's Guide*. Subroutines related to quotas are described in Chapter 11, Disk Quotas, and in *Subroutines Reference II: File System*.

Programmer Interfaces to the File System

4

.....

Chapter 2, The PRIMOS File System, introduced you to the concepts of the file system you need to know in writing programs that deal with files, access categories, and the various kinds of directories that the file system supports.

This chapter explains the file system interfaces that you as a programmer can use to communicate with the file system, what these interfaces allow you to do, and the principles involved in using them.

Communicating With the File System

As a programmer using PRIMOS programming tools like editors, compilers, and linkers, you have at your disposal a number of procedures by which you can communicate with the file system. From your terminal you can use **commands** to attach to directories, set access to file system objects, and create, open, close, and delete file system objects. These commands invoke PRIMOS programs that in turn call **subroutines** that perform the requested functions. Some PRIMOS programs invoke **command functions**, which in turn invoke subroutines to do their tasks.

Commands

Commands constitute the highest-level programmer interface to the PRIMOS operating system. This is the interface that you use to request the execution of PRIMOS programs stored in the standard command directory CMDNC0, and to execute any application program you have developed and installed in this directory. Descriptions of all PRIMOS commands are given in the *PRIMOS Commands Reference Guide*. You or someone in your organization should provide information on the execution of your application programs.

You can also request the execution of a program stored in a directory other than CMDNC0 by invoking the RESUME command, supplying the pathname of the program as an argument.

Command Functions

Command functions can be considered the second highest-level programmer interface after the command level. Command functions are used in a PRIMOS command line, and are analogous to subroutine calls in a program: during program execution, a subroutine call in a program statement requests the service of a precompiled procedure stored in a subroutine library; a command function requests the execution of a precompiled procedure at PRIMOS command level. A command function consists of a function name and zero or more arguments or options, all enclosed in square brackets ([]). It differs from a command in that it can return a value and store it in a variable for use by a subsequent command or command function. Command functions are explained in the *PRIMOS Commands Reference Guide* and the *CPL User's Guide*.

For repetitive operations at command level, you can build a series of commands and command functions into a **Command Procedure Language (CPL)** file. You can store a CPL program in one of your directories and execute it by invoking it from PRIMOS command level using the **RESUME** command (for detailed explanations, see the *CPL User's Guide*).

You can also store CPL programs in **CMDNCO** and invoke them directly as commands. However, for all but the simplest of routines, a CPL program's execution speed tends to be slower than that of the equivalent program stored in compiled form.

Subroutine Calls

As described in Chapter 3, *Accessing the PRIMOS File System*, your application programs can contain subroutine calls that perform a variety of functions involving the file system: opening and closing files, reading and writing data, as well as a number of operations involving pathnames, access control, and the like. You can make use of the extensive library of subroutines supplied by Prime, but you can also create your own libraries of subroutines tailored to the needs of your applications. Commands and command functions make extensive use of subroutines supplied by Prime during their execution; for example, the editor program uses subroutines to open, read, write, and close text files, as well as to create new files when necessary. These operations implicitly involve other subroutines that may, among other things, attach to top-level directories, evaluate access rights, and supply access control lists for newly created files. All of these actions are largely invisible to you as you sit at your terminal running the editor, unless you attempt to violate an access right, or PRIMOS detects some kind of abnormal condition such as a directory quota overflow.

System Primitives

Subroutine calls are not necessarily single-level operations, but may progress to one or more sublevels. There is a point at which no further sublevels are called

during a subroutine's execution. A subroutine that itself makes no calls to other subroutines is known as a **system primitive**; it is the lowest programmer-visible interface between a program and PRIMOS. The PRWF\$\$ subroutine, for example, is a system primitive that positions, reads, writes, or truncates a file; it can be called directly from a program, or indirectly through other subroutines such as SRCH\$\$ (used to open, close, delete, change access, or verify the existence of a file).

Arguments and Options

Arguments and options are additional elements of all of the programming interfaces described so far. They increase the flexibility of operations of commands, command functions, and subroutines by allowing variations in the ways in which they operate. An argument is usually a character string that defines the object to be operated on, such as a filename, a directory name, a file unit number, or one of the several forms of pathname. An option defines the way the object is operated on.

For a call to the SRCH\$\$ subroutine, for example, an argument would be the name of a file unit to be operated on, and an option could specify that the desired action is to open the file unit. Another option could specify whether the file unit was to be opened for reading, writing, or both. A subsequent call to SRCH\$\$ would be used to close the file unit, using the same file unit number (argument) and a different action (option).

For example, to open a new DAM file for writing on an unused file unit, perform some write operations on it, and then close it, you could use the following sequence of calls:

```
CALL SRCH$$ (K$WRIT+K$GETU+K$NDAM, NEWFILE, 7, UNIT, TYPE, CODE)
.
. /*Do some write operations
.
CALL SRCH$$ (K$CLOS, 0, 0, UNIT, 0, CODE)
```

The three K\$ options in the first call specify opening a DAM file (K\$NDAM) for writing (K\$WRIT) on an available file unit (K\$GETU). The K\$CLOS option in the second call causes the file to be closed. UNIT is a data element defined in the program to receive the file unit number returned by the subroutine when it opens the file; it also specifies the file unit to be closed. The zero (0) entries in the close call indicate that space must be reserved in the calling sequence for all elements of the call, even though some may be unused for certain actions.

At command level, arguments and options are similarly used. For example the SET_ACCESS command accepts both an argument to specify the name of the object on which the access control list is to be set, and an option to specify whether the list is to be obtained from an access category or set the same as another (existing) object.

Attach Points and Access Rights

All of the programming interfaces to the file system assume that you as a programmer at a terminal, or a user using one of your programs, can access the object or objects to be worked on. That is, the user ID of the person working on an object must exist (either explicitly or implicitly) on that object's access control list (ACL), and the ACL must include, for that user ID, the kind of access appropriate to what the person wants to do. (Refer to the *PRIMOS User's Guide* for details on access control lists.)

In order to gain access to a file system object, you (or your program) must also be attached to the directory that either directly or indirectly (by way of one or more lower-level directories) contains the object. You can attach to a directory from your terminal at command level by using the ATTACH command; your program can do the same thing by using one of the AT\$ subroutine calls. In both cases, Use (U) access is required at all directory levels that have to be passed through to get to the object.

The Three Attach Points: The initial, home, and current attach points identify your (or your user's) initial, home, and current directories. Other terms refer to these attach points as follows:

- The initial attach point identifies the initial, origin, or login directory.
- The home attach point identifies the home, or working directory.
- The current attach point identifies the current directory.

The terms attach point and directory are generally interchangeable. You establish an attach point by attaching to a directory.

The PRIMOS file system is heavily dependent on attach points. Most commands, command functions, and subroutines involving file access use the current attach point. Subroutines that accept pathnames to objects outside the home directory can temporarily change the current attach point during their execution. Some file system subroutines allow the attach points to be permanently changed.

There are specific uses for and restrictions on the three attach points, summarized as follows:

<i>Attach Point</i>	<i>Use</i>
Initial	Attaches you to your initial directory. The initial attach point is established when you first log in. From the terminal, you can attach to your initial directory at any time by issuing the PRIMOS command ORIGIN. Your program can attach to the initial directory by a call to the AT\$OR subroutine.

Home Establishes and attaches you to your home directory. This directory is your primary working directory. From the terminal, you can change the home directory by using the ATTACH command; a program uses a call to the AT\$HOM subroutine. Changing the home attach point also changes the current attach point. When commands such as LD and LIST_QUOTA are issued without arguments, the home directory is the implicit target directory. User programs may change the home attach point, but this is rarely done except when it is part of the function of the program to do this.

Current Establishes and attaches you to a current directory. The current attach point is normally the same as the home attach point. However, programs can change the current attach point by using one of the AT\$ subroutines to operate on objects outside the home directory without changing the home directory. Before returning the user to command level, programs should always reset the current attach point to the home attach point. Most PRIMOS subroutines that change the current attach point reset it to the home attach point before returning to their callers. Normally, you cannot explicitly, from command level, set the current attach point to be different from your home attach point. You can, however, explicitly reset the current attach point to be the same as the home attach point by issuing the ATTACH command with no arguments.

There are currently nine access rights that PRIMOS uses at various times to determine whether you (as a programmer) or your program (on behalf of its user) can do what you or your program want to do with a file system object. These rights and what actions they allow are explained in detail in the *PRIMOS User's Guide*. In brief:

<i>Access Right</i>	<i>Description</i>
O	Applies to files and directories; allows user to set access rights except for P and ALL; if the object is a file or a segment directory, the possessor is permitted to set the rlock.
P	Applies to directories; allows the access rights and attributes of the directory and its subordinate objects to be changed.
D	Applies to directories; allows subordinate objects to be deleted or renamed.
A	Applies to directories; allows subordinate objects to be added or renamed.
L	Applies to directories; allows their contents to be listed.

- U Applies to directories; allows the directory to be “used;” that is, attached to or passed through on the way to a subordinate object.
- R Applies to files; allows them to be read; allows EPFs to be executed.
- W Applies to files; allows them to be written.
- X Applies to local EPFs; allows them to be executed (not required if R is allowed).

Two other rights, represented by the character strings **ALL** and **NONE**, mean, respectively that all of the above individual rights, or none of them, apply to the user to whom these designations are given.

An important point to remember, when referring to a program's access to a file system object, is that it is not the program that must have access to the object, but the user on whose behalf the program is running. That is, the user ID by which a user is known to the system must exist on the access control list of the object on which an action is to be performed.

The ACL of a newly created object is always inherited from its containing directory. It is then said to have a **default ACL**. A newly created file or directory inherits all of the access rights of its parent directory (even though R, W, O, and X accesses are the only ones meaningful to a file). If you change the inherited ACL of a newly created directory, then the changed ACL becomes the default ACL for any objects subsequently created within the new directory.

The existence of the user ID on the ACL may be either explicit (the user ID itself) or implicit (the name of a group to which the user belongs or the special identifier \$REST). Each of these has its uses in particular circumstances. For example, if you are writing a program that creates a file for the exclusive use of its user, it would be appropriate for that program to create for the file an ACL that contains the user's name explicitly, and gives him the necessary rights to the file. On the other hand, if the program executes on behalf of a database group, and that group has a group ID, then it would be appropriate to create an ACL that contains the group ID and the rights applicable to the group. Any fine-tuning of this ACL with respect to specific users in the group can be done by using the ACL-related commands from PRIMOS command level.

Objectnames

The ways in which objectnames can be specified vary from command to command, command function to command function, and subroutine to subroutine. The allowable forms of objectnames (simple names, relative, full, or absolute pathnames) for the various levels of PRIMOS interfaces are defined in the appropriate manuals and guides. For subroutines that deal with the file system, they are given also in later chapters of this book.

You must keep in mind, when writing application programs that use file system subroutines, that the way you specify an objectname in a subroutine call (if you have a choice of method) can affect one or more of your attach points in some unexpected way. It may also determine whether or not the user on whose behalf your program is running has access to the object whose name is specified. Refer to the section titled *How and When Access Is Calculated* in Chapter 3, *Accessing the PRIMOS File System*, and remember that the same subtleties of the ACL mechanism that apply at command level can also apply at the command function and subroutine levels.

When interpreting objectname arguments, subroutines make a distinction between home and current directories that is not made at command level or command function level. For a subroutine, the current directory is the directory to which the process is currently attached. The home directory is either the one first attached to when the user logs in, or the one specified in a subroutine call such as AT\$HOM.

Assume, for example, that you have used the ATTACH command to attach to a directory MYDIR. Your home and current attach points are now MYDIR. Now, you invoke a command or program with a pathname as an argument:

MYPROG JANESDIR>MEMOS

The behavior of the home and current attach points is as follows:

1. The home attach point remains the same; from your point of view the attach point does not change.
2. MYPROG calls various subroutines that locate, check access, and open the MEMOS file in the JANESDIR directory. The subroutines change the current attach point to JANESDIR.
3. When the program terminates, the last subroutine executed (typically the one that closes the file) sets the current attach point back to MYDIR.

When you use a subroutine that accepts only a simple pathname, you must know the current attach point (and hence the current directory), because the current directory is the one that is used to determine the pathname of an object referred to by a simple name.

File Units and Attributes

When a file is opened using a subroutine call such as SRCH\$\$, it becomes associated with a file unit number, which is used in subsequent subroutine calls to manipulate the file data. A file can be read or written only by referring to its file unit number in read or write subroutine calls. File units are described more fully in the *Subroutines Reference II: File System*.

Files can be opened by specifying a file unit number explicitly or by allowing PRIMOS to allocate one (except in the FORTRAN language, which requires an explicit file unit number). If you are writing a program that is entirely self-contained (that is, it does not support, require support from, or otherwise communicate file information to another program), it makes little difference how you associate a file with its file unit number, other than to make sure that an explicitly defined number is not already in use by the same program. However, if your program is one element of a larger group of programs that make up a subsystem and that have to communicate file unit information among themselves, then it is more appropriate to let PRIMOS allocate file unit numbers, and to have the program that opens the file the first time store the returned file unit number in a program variable accessible to all components of the subsystem. This technique is particularly appropriate when a number of file units are opened at various times and in unpredictable order.

In programming a subsystem, once a file has been opened for the first time and associated with a file unit number, then that number should be used for all subsequent operations on that file, using the centrally stored file unit number returned from the first open call. In particular, if the same file is opened more than once during an application's execution, the file unit number resulting from the first open call should be used to explicitly define the number for subsequent open calls, rather than letting PRIMOS allocate a possibly different number and cause inconsistencies to arise among the members of the family of programs in the subsystem.

When your program has opened a directory containing a file system object, a set of attributes describing each object contained in the directory is available to the program. The attributes are read by the ENT\$RD subroutine call into a structure that your program provides, as described in detail in Chapter 10, File Attributes.

You must remember two things when using a subroutine that reads, sets, or changes the attributes of an object. First, the containing directory must be open and associated with a file unit number, since this is the argument that the subroutine uses to determine which directory to look in for the attribute list. Second, the object whose attributes are to be obtained, set, or changed must be immediately contained within that directory, since the argument specifying the object's name does not accept a pathname (that is, the object is assumed to be in the current directory).

The subroutine used to set or change attributes is SATR\$\$, which is fully described in the *Subroutines Reference II: File System*, along with the formats of the structures that your program needs to provide for its operation.

PRIMOS Responses (Return Codes)

Virtually all PRIMOS subroutines communicate with their callers in one consistent respect: they return a numeric code that informs the caller of the subroutine's success or failure in performing its task. For consistency,

subroutines that you write for your own applications should also follow this practice.

PRIMOS subroutines always place the return code in a 16-bit binary integer data item. If the subroutine was entirely successful in completing its requested function, the value of this integer is always zero (0). Other values are returned in case of total failure or partial success. Your program should always check the value of the return code upon returning from a subroutine call and take whatever action is appropriate to the reported condition.

A complete list of PRIMOS subroutine return codes is provided in the *Subroutines Reference II: File System*, with some examples of how a program might respond to a nonzero response code.

It is important that a subroutine call that can potentially change the current attach point be handled carefully when a nonzero code is returned. In order that the programmer can rely on some consistent current attach point even if a subroutine fails, most PRIMOS subroutines cause the current attach point to be set to the home attach point before returning to their callers, regardless of where the current attach point was before the call. Any programs, command functions, or subroutines that are to become part of a larger subsystem should handle nonzero return codes in a consistent way, and should be documented accordingly.

File System Operations: An Overview

This section gives you an overview of the five major operations (creating, opening, reading, writing, and deleting) that your programs can perform on file system objects and the general requirements that must be satisfied in order to do these operations. They are explained in more detail in subsequent sections.

General Requirements

In order to perform operations on file system objects, the users of your programs must be able to attach to the appropriate directories, and, in order to do this, they must have rights appropriate to what they want to do. A successful attach to a directory requires that the user have Use access to all directory levels from the MFD down to the level that contains (or will contain) the object. Additional rights required on the directory immediately containing the object depend on the action that is to be performed. For example, in order to change the name of a file, its owner must have both Add and Delete access to the directory containing the file.

Creating Objects

Programs that operate on files contain calls to subroutines that locate the files to be operated on, either in the user's home directory or in the current directory. If the attempt to locate a file that is to be opened for writing, or for both reading and writing, is unsuccessful, you can give the program the option of creating it in whichever directory it was being searched for. You do this by supplying a key that specifies the type of file to be created if it is not found. Your program can also create lower-level directories by using the same subroutine calls with the appropriate keys. Creating a new top-level directory requires a different subroutine from that which creates lower-level directories and files.

If a search for a file *for reading* is unsuccessful, the subroutine returns an error code; the program must decide how to handle this condition. It is fairly probable that the file is not found because the program is attached to a directory other than the one in which the file is expected to exist; in this case the user is most likely expected to have attached to the proper directory from PRIMOS command level before executing the program. However, if you suspect that the file to be read may *not* exist, then you should, by means of the appropriate key, test for the file's existence. The program should also report its nonexistence and provide for a graceful escape from the situation.

Opening Objects

Your programs open file system objects by using calls to any of several subroutines, depending on where the object is relative to the home directory, what kind of optional actions are desired (for example, creating new objects or retrying in case of initial failure), and whether your applications are more suited to using system library subroutines or application library subroutines. The *Subroutines Reference II: File System*, contains a chapter of all of the subroutines you can use to open file system objects.

In general, the subroutines in the application libraries (APPLIB or VAPPLB) are easier to use for application programs, as their user interfaces are comparatively simple and they return codes that are either true or false. In many cases, these subroutines call lower-level subroutines, taking care of supplying arguments with which you as a programmer need not concern yourself. They also perform all possible error detection and recovery tasks before returning to their callers, thus ensuring that everything that can be done to complete the requested function is done, and that whatever errors are encountered are reported.

Reading Objects

Assuming that your program has successfully opened an object for reading or for reading and writing, the object can then be read, using any of several subroutine calls. The call to be used depends on whether the object is a file, a file directory, or a segment directory; there are also calls intended expressly for reading ASCII

text files, getting characters or lines of text from command files or from the terminal, and getting characters from an array.

Positioning an object involves an implied read of the object, although no data is actually transmitted. Calls to position an object can be made either with a specified absolute position or with a position relative to the current position, either backward or forward. An object can also be positioned at its beginning or end.

Writing Objects

Assuming that your program has successfully opened an object for writing or for reading and writing, the object can then be written, using any of several subroutine calls. Generally, the writing of data files is done by explicit calls to write a line to a text file, a data file, or a command output file or terminal. You can also call a subroutine whose function is to store characters into an array.

The writing of other file system objects (file and segment directories and access categories) is done implicitly during many operations on files. File creation, ACL manipulation, and file renaming, for example, all implicitly involve writing to these objects, but there are no explicit subroutine calls that result in writing a specific character or string to them.

Files can be positioned to any arbitrary point before writing data to them. Normally, when additional data is written to a SAM or DAM file, the file is positioned to its end before writing is done; if the position is somewhere within the file (or at its beginning), existing data is overwritten. Indexed files, such as those used by MIDASPLUS, are capable of having records inserted into them; such subsystems take care of insertions in such a way as not to overwrite existing data.

Deleting Objects

Several subroutines are available to delete files and directories; the one you choose usually depends on whether the object is directly contained in the home directory or elsewhere.

The ability of your program to delete an object depends on the user's access rights not only on the object itself but also on the parent directory. The state of the delete-protect attribute on the object also affects the user's ability to delete an object, independent of access rights. The ability to set or reset this attribute, in turn, depends on the user's having Protect rights on the parent directory.

Having given you an overview of the programmer's interfaces to the file system and the kinds of things that can be done with file system objects, the rest of this chapter gives more details on file system operations at the command level and at the subroutine level.

Access Control to File System Objects

This section describes the requirements and procedures for attaching and controlling access to file system objects, both at command level and at the subroutine level.

As previously described, only file directories can be attached to; you cannot attach to segment directories, access categories, or files directly, but you can attach to the file directories that contain any of these objects.

Attach/ACL Requirements

Your user ID, or that of your program's user, must appear at all directory levels above the directory that is being attached to, and the access rights must include Use access. The user ID can be explicit, or it can be implied as the member of a specific group—id or the special group \$REST. Use access can be specified explicitly (U access) or implicitly (ALL access).

Access control lists and the subroutines for manipulating them are described more fully in Chapter 9, Access Control Lists (ACLs).

Attaching

At the command level you can attach to two of the three kinds of attach points: the home attach point and the initial, or origin, attach point. Remember that the initial attach point, the point at which you are attached when you first log in, cannot be changed except by your System Administrator or Project Administrator. You can change your home attach point, however, at any time; in fact, if the files you are working on (specifically, program files if you are a programmer) are in a directory other than your initial directory, you should use Attach commands to attach to the directory containing them.

At the subroutine level, your programs can set not only the initial and home attach points, but also the current attach point, a (usually) temporary attachment that is in effect only for the duration of the routine in which the subroutine is called.

Attach to Initial Directory

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
ORIGIN	None	AT\$OR

Attach to Origin Directory (Command): To attach to the initial, or origin, directory from PRIMOS command level, use the command:

**ORIGIN
OR**

Using the ORIGIN command sets both the home and the current attach points to the initial directory. The ORIGIN command requires no arguments.

Attach to Origin Directory (Command Function): There are no command functions that explicitly set the attach point to any directory. However, some command functions, such as OPEN_FILE, could implicitly attach temporarily to the initial directory. A CPL program that needs to attach specifically to the initial directory can use the ORIGIN command as one of its statements.

Attach to Origin Directory (Subroutine): To attach to the initial directory from a program, use the subroutine call:

AT\$OR (key, code)

The value of *key* is K\$SETH if both home and current attach points are to be set to the initial directory, or K\$SETC if only the current attach point is to be set.

Note that if only the current attach point is set, any subroutine that uses a simple objectname as an argument looks in the initial directory for the object, regardless of the setting of the home attach point.

The details of the calling sequence for the AT\$OR subroutine are given in Chapter 6, Attach Points.

Attach to Home Directory

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
ATTACH	None	AT\$HOM

Attach to Home Directory (Command): To define and attach to the home directory from PRIMOS command level, use the command:

ATTACH [directory_name]
A

Using the ATTACH command sets both the home attach point and the current attach point to the directory specified as the argument. If no argument is given, no change occurs (unless the current attach point has been left set at some other point in a previous operation, in which case it is reset to the home attach point).

directory_name can be any form of pathname that leads to a file directory.

Attach to Home Directory (Command Function): There are no command functions that explicitly set the attach point to any directory. A CPL

program that needs to specifically set the home directory can use as one of its statements the ATTACH command in the form just described.

Attach to Home Directory (Subroutine): To set the current attach point to the current home directory from a program, use the subroutine call:

AT\$HOM (*code*)

The details of the calling sequence for the AT\$HOM subroutine are given in Chapter 6, Attach Points.

Attach to Any Directory

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
ATTACH	None	AT\$ AT\$ABS AT\$ANY AT\$REL

Attach to Any Directory (Subroutine): To set the current and (optionally) the home attach points to a specific directory (other than the initial or home directory), use one of the following subroutine calls:

- AT\$ (*key, path, code*)
- AT\$ABS (*key, partition, directory, code*)
- AT\$ANY (*key, name, code*)
- AT\$REL (*key, name, code*)

Details of these calling sequences and their operations are given in Chapter 6, Attach Points.

In all of these calls, the value of *key* determines whether both the current and home attach points are to be set, or only the current attach point. A value of K\$SETH sets both; a value of K\$SETC sets only the current attach point. If you specify K\$SETH, the effect is the same as if the ATTACH command had been used at the terminal.

The AT\$ call is the most general of all of the attaching calls, in that it accepts a pathname in any form, and then calls one of the others, depending on the results it obtains from parsing the pathname. A null *name* argument (") means the home directory, and is equivalent to the AT\$HOM call or the ATTACH command with no argument. You can use the AT\$ call to attach to anywhere from anywhere, regardless of whether or not the current and home attach points were the same before the call.

In the AT\$ABS call, *partition* is the name or logical disk number of an active disk on the system on which your program is running, or on another system connected through a network. The partition argument can also be the null string, implying logical disk 0 (zero); or it can be '*', signifying the disk partition containing the directory to which the current attach point is set at the time of the call.

The *directory* argument is the name (and optional directory password, separated by a single space) of a top-level directory on the disk partition identified by *partition*. A null *directory* argument signifies the MFD of the disk partition.

The AT\$ANY call requires *name* to be an unqualified pathname, beginning with the name of a top-level directory. Remember the rules that were given in Chapter 3, Accessing the PRIMOS File System, for directory searching when using an unqualified pathname.

The AT\$REL call requires *name* to be the name of a directory immediately subordinate to the current directory. It can include a directory password, separated by a single space.

Access Control List (ACL) Functions

The ACL functions can be used at the command level to define, modify, list, and delete user access rights on file system objects. You can define ACLs by default from the object's containing directory, by specifying separate user IDs and their individual rights, or by specifying user groups and the rights that apply to them. You can also define access categories that protect any number of objects with the same ACL. The *PRIMOS Command Reference Guide* and the *PRIMOS User's Guide* explain the use of the various ACL-related commands in detail.

When using ACL-related subroutines in a program, your program must furnish the ACL entries in the form of a structure containing the user ID/access right pairs; the subroutine call supplies the address of the structure in the form of a pointer argument, *addr(acl_struct)*. Chapter 9, Access Control Lists (ACLs), gives the details of the calling sequences and operations of all of the ACL-related subroutines.

The ACL structure is shown pictorially in Chapter 9, Access Control Lists (ACLs), and in program declaration form in the *Subroutines Reference II: File System*.

The target object for any ACL-related command or subroutine can be a file, a file directory, or a segment directory. An access category is a special object that contains an ACL used to protect other objects; the ACL of the access category itself is the same as that of the group of objects it protects.

At both command and subroutine levels you, or your program's user, must have Protect and List access to the containing directory, and Protect access to the object on which an ACL is to be set.

Setting Default Access

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
SET_ACCESS	None	AC\$DFT

Setting Default Access (Command): PRIMOS gives a default ACL automatically to any object whenever the object is created; the ACL is the same as that of the containing directory. (The System Administrator or Project Administrator should set a specific ACL, as described later, on a top-level directory if it is to be different from that of the MFD.) Any objects created at levels below the top-level directory then gets this specific ACL by default.

To set a default ACL from PRIMOS command level, use the command

```
SET_ACCESS objectname  
SAC
```

In this form of the SET_ACCESS command, if the target object has an ACL different from the default, its ACL is reset to the default. A message may be returned indicating that there is already an ACL set on the object and asking whether it is to be replaced; the message can be suppressed by using the -NO_QUERY option.

Be careful, when you set the default access on an object, that the directory that is supplying the default ACL has rights appropriate to the object on which the default is being set. For example, Read and Write access as such are not meaningful to directories, but are usually included in directory ACLs so that they are inherited by subordinate files automatically.

Setting Default Access (Command Function): There are no command functions to set access control lists. A CPL program that needs to set an ACL can use the appropriate PRIMOS command as a program statement.

Setting Default Access (Subroutine): To set a default ACL from a program, use the subroutine call

```
AC$DFT (name, code)
```

The *name* argument can be any of the valid forms of pathname. The same precautions regarding propagated ACLs apply to the AC\$DFT subroutine as to the SET_ACCESS command described above.

Details of the calling sequence and its operation appear in Chapter 9, Access Control Lists (ACLs).

Setting Specific Access

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
SET_ACCESS	None	AC\$SET

Setting Specific Access (Command): To set a specific ACL from PRIMOS command level, use the command

```
SET_ACCESS objectname user-id:access-rights . . . [-NO_QUERY]
SAC
```

In this form of the SET_ACCESS command, the resulting ACL contains the list of users and access rights given as arguments to the command, plus, by default, \$REST:NONE if no other specific rights are given to the \$REST group. The ACL thus produced replaces any ACL already existing on the object. To modify an existing entry on an ACL without replacing the ACL, use the EDIT_ACCESS command, described later.

If *objectname* does not exist, PRIMOS assumes that you want to create an access category. If you do, refer to Creating an Access Category, described later; otherwise answer NO to the query returned by PRIMOS.

Setting Specific Access (Command Function): There are no command functions to set access control lists. A CPL program that needs to set an ACL can use the appropriate PRIMOS command as a program statement.

Setting Specific Access (Subroutine): To set a specific ACL from a program, use the subroutine call

```
AC$SET (key, name, addr(acl_struct), code)
```

In the AC\$SET subroutine call, *name* governs the creation and replacement of ACLs and specifies the error to return if AC\$SET is called to replace a nonexistent ACL or to create an ACL on an object that already has one. The AC\$SET description in *Subroutines Reference II: File System* lists the possible key values and their meanings. *name* specifies the object that is to receive the new ACL, as in the AC\$DFT call previously described. The structure of the ACL entries is shown in diagrammatic form in Chapter 9, Access Control Lists (ACLs). Each entry can have as many as 80 characters, and there can be as many as 32 entries in a given list.

Setting Category Access

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
SET_ACCESS	None	AC\$CAT

Setting Category Access (Command): To set the access of an object to that of an existing access category, use the command

```
SET_ACCESS  objectname -CATEGORY acatname
SAC                               -CAT
```

objectname argument can be any valid form of pathname. The access category specified by *acatname* must exist in the same directory as that of the object being protected. (Creating an access category is described later in this section.)

Setting Category Access (Command Function): There are no command functions to set access control lists. A CPL program that needs to set an ACL can use the appropriate PRIMOS command as a program statement.

Setting Category Access (Subroutine): To set the ACL of an object from a program, use the subroutine call

```
AC$CAT (name, category, code)
```

The *name* argument identifies the object to be protected; it can be any valid form of pathname. *category* is the simple name of the access category that is to protect *name*; the access category must exist and must reside in the same directory as *name*. The calling sequence and operation of the AC\$CAT subroutine are described more fully in Chapter 9, Access Control Lists (ACLs). Access requirements for using the AC\$CAT subroutine are described in the *Subroutines Reference II: File System*.

Setting Access Like That of Another Object

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
SET_ACCESS	None	AC\$LIK

Setting Access Like That of Another Object (Command): To set an object's access so that it is identical to that of another object from PRIMOS command level, use the command

```
SET_ACCESS  objectname1 -LIKE objectname2
SAC
```

Both *objectname1* and *objectname2* can be any valid form of pathname; objects need not be in the same directory. *objectname1* identifies the target object on which the access is to be set; *objectname2* identifies the object whose access is to be applied to the target object.

There is also no requirement that source and target objects be of the same type. If the source and target objects are of different types (for example, the source is a

directory and the target is a file), be sure that the source object includes access rights appropriate to the target, as described previously in this chapter.

When you use this form of the command, it does not matter whether the source object's ACL is derived from its superior directory, from an access category, or a specific ACL; the ACL of the target is always a specific ACL, since it is the ACL's values that are copied, not the location of its source.

Setting Access Like That of Another Object (Command

Function): There are no command functions to set access control lists. A CPL program that needs to set an ACL can use the appropriate PRIMOS command as a program statement.

Setting Access Like That of Another Object (Subroutine): To set an object's access so that it is identical to that of another object from a program, use the subroutine call

AC\$LIK (target, reference, code)

Both *target* and *reference* are any valid form of pathname; *target* identifies the object on which an ACL is to be set, while *reference* identifies the source of the ACL. The actions are the same as described in the command description just given; the calling sequence is described more fully in Chapter 9, Access Control Lists (ACLs). The *Subroutines Reference II: File System* gives information on the access rights required to use the AC\$LIK call.

Creating an Access Category

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
SET_ACCESS	None	AC\$SET

Creating an Access Category (Command): To create an access category from PRIMOS command level, use the command

SET_ACCESS *objectname user-id:access-rights . . .*
 SAC

This is the same form of SET_ACCESS command as you use to set a specific ACL on an object, as described previously under Setting Specific Access. The difference is that, in this case, *objectname* identifies a nonexistent object, and PRIMOS assumes that you want to create an access category. PRIMOS tells you that the access category does not exist and asks whether you want to create it. If you do, the access category is created and given the name *objectname.ACAT* and the specified ACL entry or entries. You can then use this access category in subsequent operations to set category access as described previously.

Be careful, if you really want to create an access category, that the named object does not exist; otherwise, PRIMOS locates the named object and apply the specified ACL entry or entries to it, with possibly unwanted results. If you know that an object whose name is, say, PRIVATE exists, you can still create an access category with the name PRIVATE.ACAT in the same directory by explicitly supplying the .ACAT suffix when giving *objectname*. PRIMOS recognizes this as a different object from PRIVATE, and creates the access category PRIVATE.ACAT.

The *objectname* argument can be any valid form of pathname, implying that you can create an access category anywhere. Remember, though, that an access category must be in the same directory as the object(s) it is intended to protect.

Creating an Access Category (Command Function): There are no command functions to create access categories. A CPL program that needs to create one can use the appropriate PRIMOS command as a program statement. It would be prudent for your CPL program to test for the existence of the named object using the [EXISTS] command function before attempting to use the command to create an access category. If the function returns a result indicating that the object exists, it should allow the user to specify what to do. Refer to the *CPL User's Guide* for information on the [EXISTS] command function and how to query the user and request a response.

Creating an Access Category (Subroutine): To create an access category from a program, use the subroutine call

AC\$SET (*key, name, addr(acl_struct), code*)

When using AC\$SET to create an access category, *name* must identify a nonexistent object (any valid form of pathname), and *key* must have a value of either 0 (zero) or K\$CREA. As before, *addr(acl_struct)* is a pointer to an area in your program that contains the structure of the ACL to be set on the access category.

The calling sequence and operation of the AC\$SET subroutine are more fully presented in Chapter 9, Access Control Lists (ACLs). The *Subroutines Reference II: File System* gives the access rights required to use the AC\$SET call.

Changing Access to an Object

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
EDIT_ACCESS	None	AC\$CHG

Changing Access to an Object (Command): To change an existing ACL on a file system object from PRIMOS command level, use the command

EDIT_ACCESS *objectname user-id:access-rights . . .*
EDAC

The *objectname* argument identifies a file system object that already has an ACL of any type: specific, category, or default. The object can be identified by any valid form of pathname. The ACL argument(s) identify one or more individual entries on the list that are to be added, deleted, or changed. Only the specified entries are affected; unreferenced entries are left on the list unchanged.

Changing Access to an Object (Command Function): There are no command functions to modify access categories. A CPL program that needs to modify one can use the appropriate PRIMOS command as a program statement.

Changing Access to an Object (Subroutine): To change an existing ACL on an object from a program, use the subroutine

AC\$CHG (name, addr(acl_struct), code)

In the AC\$CHG call, the *name* and *addr(acl_struct)* arguments have the same functions and requirements as in the AC\$SET call described earlier. This is the fundamental call used for changing access, and behaves in the same way as the EDAC command. There are other methods, which are described in Chapter 9, Access Control Lists (ACLs), used to change an existing ACL to that of another object and to make selective modifications to it afterwards.

Deleting Access Control Entries

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
SET_ACCESS	None	AC\$SET
EDIT_ACCESS	None	AC\$CHG

Deleting ACL Entries: There are no explicit commands, command functions, or subroutines that perform the sole function of deleting an ACL entry or entries; the basic approach to accomplish this is to use the SET_ACCESS or EDIT_ACCESS functions, and to include entries that contain the special access right NONE.

For example, if an ACL contains an entry BAKER:LUR and you want to exclude user BAKER from any access at all, you can use the EDIT_ACCESS command (or the AC\$CHG subroutine call), specifying the explicit entry BAKER:NONE. This explicitly states that user BAKER has access NONE, and an entry to this effect is placed on the ACL. Alternatively, you can use the EDIT_ACCESS command, specifying BAKER:, that is, the user ID and the colon, but no access rights. This results in the entry for user BAKER being deleted from the ACL entirely.

You can also use the SET_ACCESS command (or the AC\$SET subroutine call) and explicitly specify all of the entries on the existing ACL except the entry for BAKER.

Using the EDIT_ACCESS command is much the easier method, especially if the ACL is long and complex.

Creating File System Objects

File system objects are created in several different ways, depending on the type of object. In order to create any type of object, you (at command level) or your program's user must have Add access to the directory immediately containing the object, and Use access to any higher-level directories.

For those objects that can be created at command level (file directories, files, and access categories), you can specify either a simple name to create the object in the home directory, or a pathname to create the object in any other directory for which you have the appropriate access.

Creating Portals

A **portal** is a file system object, new at Rev. 23.0, that re-routes file system references from one directory to another. Portals are used to reference other common file system name spaces. Any references (for example, the AT\$-type subroutines) to the original directory are automatically redirected by a disk-directed portal to the MFD of the specified directory on the remote machine.

Creating Portals

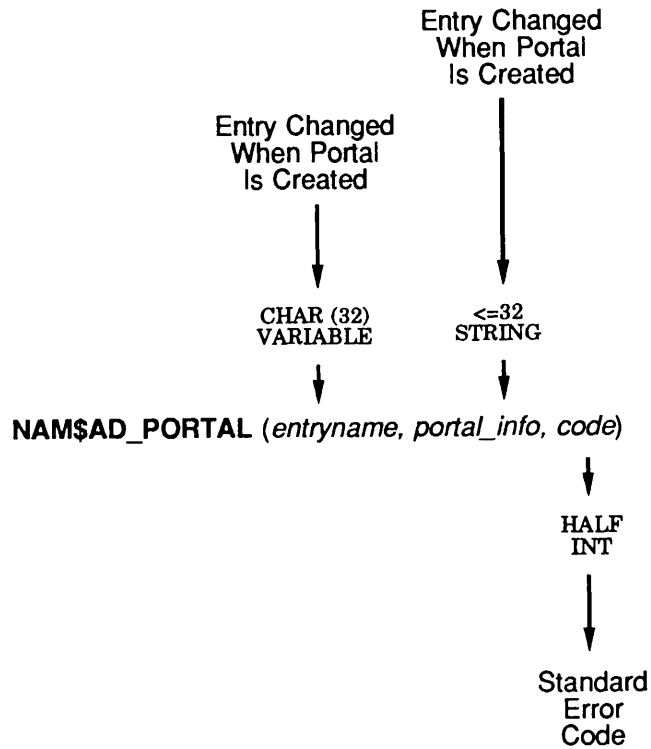
<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
ADD_PORTAL	None	NAM\$AD_PORTAL

Creating a Portal (Subroutine): To create a portal from a program, use the subroutine call

NAM\$AD_PORTAL (*entryname, portal_info, code*)

NAM\$AD_PORTAL converts an existing directory entry into a portal by mounting the defined portal over the directory. Future references to the original directory are redirected to the portal until you remove the portal with the NAM\$RM_PORTAL subroutine, described later in this chapter, or the REMOVE_PORTAL command. You may only use this subroutine at the supervisor terminal (User 1). Figure 4-1 shows the calling sequence of the NAM\$AD_PORTAL.

Creating a Portal



Q04.D1.D10056.3LA

Figure 4-1. Calling Sequence of NAM\$AD_PORTAL

- entryname* [char(32) var] The entry that is changed when you create the portal.
- portal_info* [string] The input structure that defines the attributes of the portal you are creating.

```

NAM$K_ROOT by 1, /* is root-portal */
NAM$K_NOROOT by 2; /* is disk-portal */
dcl 1 portal_info,
    2 version fixed bin(15),
    2 portal_target_key fixed bin(15),
  
```

```

2 portal_target,
3 node_name char(16) var, /* must be
   specified */
3 partition_name char(6); /* only for
   disk_portal */

```

code [fixed bin] The standard return code. See the next section for a list of error codes.

NAM\$AD_PORTAL Errors: The following list describes errors associated with the NAM\$AD_PORTAL subroutine.

<i>Error Name</i>	<i>Description</i>
E\$SCCM	This routine may only be used at the supervisor terminal.
E\$BVER	The portal structure version number that you specified is invalid.
E\$BKEY	The key that you specified is invalid.
E\$WRIT	You do not have access rights for this operation.
E\$BNAM	The entryname that you specified uses incorrect syntax.
E\$UNOD	The nodename that you specified is not in PRIMENET.
E\$BPOR	The portal target must be a remote node.
E\$IROO	A portal may not be mounted on a root directory.
E\$NTUD	The specified entryname must be a directory.
E\$MTPT	A portal already exists at the point where you are trying to mount a portal.
E\$RPMH	You cannot create a portal through another portal.
E\$IREM	The portal mount must be on a local directory.
E\$FMTF	No such entryname exists.

Creating File Directories

In order to create a directory, you (or your program's user) must have Add access to the directory (which may be the MFD) that contains the directory, and Use access to any directories that are superior to the one being created.

Creating File Directories

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
CREATE	None	DIR\$CR CREA\$\$ *

* The CREA\$\$ subroutine is documented in Appendix A of *Subroutines Reference II: File System*. It is considered obsolete at PRIMOS Rev. 20.2. Although CREA\$\$ is still supported, programs should use DIR\$CR beginning with Rev. 20.2.

Creating a File Directory (Command): To create a file directory from command level, use the command

```
CREATE directory_pathname [-MAX n] [-CATEGORY acatname]  
CR                                     -CAT
```

The *directory_pathname* argument can be any legitimate form of pathname, implying that you can create a file directory anywhere, provided, of course, that you have the appropriate access. The ACL of the new directory is the same as that of the containing directory; you can modify it once the directory exists by using any of the access control commands described previously.

Creating a File Directory (Command Function): You can include the CREATE command in a CPL program in the same form that you use when you enter the command at your terminal; if you invoke the CPL program from your terminal, the results are the same, including the return of error messages. However, if you invoke the CPL program as a phantom, no error messages are returned to your terminal. The program would not, for example, return a message if you were to try to create a directory that already existed. It would therefore be wise to check for the existence of the directory before attempting to create it; you can use the [EXISTS] command function for this purpose, as described in the *CPL User's Guide*.

Creating a File Directory (Subroutine): To create a file directory from a program, use the subroutine call

```
DIR$CR (dirname, addr(attributes), code)
```

The DIR\$CR subroutine creates a lower-level directory in the location indicated by the pathname. It creates a password directory if the current directory is a password directory; in this case, the owner and nonowner passwords are applied to the new directory. If the current directory is an ACL directory, the new directory is also an ACL directory; in this case, any passwords supplied in the call are ignored.

Note The CREPW\$ subroutine creates a password directory within an ACL directory. It is documented in Appendix A of the *Subroutines Reference II: File System*. CREPW\$ is considered obsolete at PRIMOS Rev. 20.2. Although CREPW\$ is still supported, programs should use DIR\$CR beginning with Rev. 20.2.

Creating Files

In order to create a file, you (or your program's user) must have Add access to the directory that is to contain the file, and Use access to all superior directories leading to this directory.

Creating Files

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
None	None	SRCH\$\$
		SRSFX\$
		TSRC\$\$ *

* The TSRC\$\$ subroutine is documented in Appendix A of the *Subroutines Reference II: File System*. It is considered obsolete at PRIMOS Rev. 20.2. Although TSRC\$\$ is still supported, programs should use SRSFX\$ beginning with Rev. 20.2.

Creating a File (Command): There is no command that explicitly creates a file; files are implicitly created by PRIMOS programs such as ED, the compilers, the PMA assembler, and the linkers SEG, LOAD, and BIND.

An empty file is implicitly created from PRIMOS command level if the OPEN command is given to open a nonexistent file for writing or for reading and writing. Opening file system objects is discussed in more detail later in this chapter and in Chapter 7, Text Storage and Retrieval, and Chapter 8, Data Storage and Retrieval.

Creating a File (Command Function): As at PRIMOS command level, there is no command function that explicitly creates a file; you can include the OPEN command as a CPL program statement if you want the program to create an empty file.

Creating a File (Subroutine): To create a file from a program, use one of the subroutine calls

SRCH\$\$ (*key, name, name_len, unit, type, code*)

SRSFX\$ (*key, name, unit, type, num_suffixes, suffixes, basename, suffix_used, code*)

These calls are described in greater detail in Chapter 7, Text Storage and Retrieval, and in *Subroutines Reference II: File System*.

In all cases, the *newfile* portion of *key* specifies the type of file (SAM or DAM) to be created if the object specified by *name* does not exist and the action to be performed is writing or reading and writing.

For the SRCH\$\$ call, the *name* argument is a simple name; the resulting file is created in the current directory and given the same protection as that of the current directory.

For SRSFX\$, *name* is any form of pathname; the resulting file is created in the directory specified by the directory portion of *name*, and given its protection.

Creating Access Categories: The creation of access categories was described earlier in the section entitled Access Control Functions.

OPENING FILE SYSTEM OBJECTS

To open a file system object, you (or your program's user) must have Use access to all directory levels leading to the object to be opened. Additional rights required on the object itself and its containing directory depend on the action to be performed on the opened object.

As described previously, attempting to open a nonexistent file normally results in that file being created in an empty state; the discussion in the following subsections assumes that the object already exists.

Opening File Directories

File directories can be opened at both command level and at subroutine level; however, they can be opened only for reading. File directories are written to implicitly whenever some action on or within the directory requires that information in the directory be updated (such as the date-time-last-modified or access control information).

Opening File Directories

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
OPEN	OPEN_FILE	SRCH\$\$ SRSFX\$ TSRC\$\$ *

* The TSRC\$\$ subroutine is documented in Appendix A of the *Subroutines Reference II: File System*. It is considered obsolete at PRIMOS Rev. 20.2. Although TSRC\$\$ is still supported, programs should use SRSFX\$ beginning with Rev. 20.2.

Opening File Directories (Command): There is not much to be gained from opening a file directory interactively, since there are no commands that enable you to read the directory interactively. However, PRIMOS does not prevent your doing this; if you want to open a file directory from PRIMOS command level, use the command

OPEN *pathname funit key*
O

pathname can be any form of pathname leading to a file directory to which you have Read access. You must specify a file unit number *funit*; PRIMOS does not look for an unused file unit when an object is being opened from command level. The *key* argument must specify a value of 1 (read). See the *PRIMOS Commands Reference Guide* for a full description of the OPEN command.

Opening File Directories (Command Function): PRIMOS allows a file directory to be opened by the OPEN_FILE command function, but does not allow any other operations (other than CLOSE) to be performed on it. Use the following form in a CPL program:

&SET_VAR *unit* := [OPEN_FILE *pathname status* -MODE R]

In this CPL statement, *unit* is a local or global variable that receives the file unit number assigned to the opened directory by PRIMOS; *status* is a local or global variable that receives the status code resulting from the operation. *pathname* can be any of the valid forms. See the *PRIMOS Commands Reference Guide* and the *CPL User's Guide* for more detailed descriptions of the OPEN_FILE command function.

Opening File Directories (Subroutine): To open a file directory from a program, use calls to the subroutines described previously for creating file system objects:

SRCH\$\$ (*key, name, name_len, unit, type, code*)

SRSFX\$ (*key, name, unit, type, num_suffixes, suffixes, basename, suffix_used, code*)

In all cases, the action portion of *key* specifies the action(s) to be performed (read, write, or read and write).

For the SRCH\$\$ call, *name* can be only a simple name, the name of the directory being searched for in the current directory.

For SRSFX\$, *name* is any form of pathname.

Opening Files

Files contained in file and segment directories can be opened for reading, writing, or reading and writing at command, command function, and subroutine levels. In all cases, Use access is required on the containing directory and superior directories, and Read, Write, or Read and Write access is required on the file, depending on the actions to be performed.

Opening Files		
<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
OPEN	OPEN_FILE	SRCH\$\$
		SRSFX\$
		TSRC\$\$ *

* The TSRC\$\$ subroutine is documented in Appendix A of the *Subroutines Reference II: File System*. It is considered obsolete at PRIMOS Rev. 20.2. Although TSRC\$\$ is still supported, programs should use SRSFX\$ beginning with Rev. 20.2.

Opening Files (Command): To open a file (either text or data) from command level, use the command

OPEN *pathname funit key*
O

The *pathname* argument can be any form of pathname leading to a file. You must specify a file unit number *funit*; PRIMOS does not look for an unused file unit when opening a file from command level. The *key* argument must specify a value indicating the action to be performed. Refer to the *PRIMOS Commands Reference Guide* for details on the use of the OPEN command and its arguments.

Opening Files (Command Function): To open a file (either text or data) from a CPL program, use a statement of the form

&SET_VAR *unit* := [OPEN_FILE *pathname status -MODE x*]

In this CPL statement, *unit* is a local or global variable that receives the file unit number assigned to the opened file by PRIMOS; *status* is a local or global variable that receives the operation's status code. *pathname* can be any of the valid forms. The mode argument *x* specifies the action(s) for which the file is being opened: R (Read), W (Write), or RW or WR (Read and Write). Note that if the file is being opened in any mode that allows writing and the file does not exist in the directory indicated by *pathname*, the file is created with no indication of an error. Therefore, if proper operation of your CPL program depends on a pre-existing file of the specified name, it would be wise to test for its existence before opening it for writing. See the *PRIMOS Commands Reference Guide* and the *CPL User's Guide* for more detailed descriptions of the OPEN_FILE command function.

Opening Files (Subroutine): To open a file from a program, use calls to the subroutines described previously for creating and opening file system objects.

SRCH\$\$ (*key, name, name_len, unit, type, code*)

SRSFX\$ (*key, name, unit, type, num_suffixes, suffixes, basename, suffix_used, code*)

In all cases, the action portion of *key* specifies the action(s) to be performed (read, write, or read and write).

For the SRCH\$\$ call, *name* can be only a simple name, the name of the file being searched for in the current directory.

For SRSFX\$, *name* is any form of pathname.

Segmented files (members of a segment directory) can be opened by the SGD\$OP subroutine call, described in Chapters 7, Text Storage and Retrieval and 8, Data Storage and Retrieval.

Reading File System Objects

After an object has been opened, it can be read under certain conditions and from some, but not all, programmer interface levels. From the command level, directories cannot be read, nor can fixed-length data records; variable-length text records can be read and displayed on the terminal, but only indirectly through a command function. Any kind of object can be read from program level

by use of several special-purpose subroutines, as well as some of the general-purpose subroutines already described. In all cases, Read access is required on the object to be read, and Use access is required to all superior directories.

Reading Directories

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
None	None	DIR\$LS
		DIR\$SE
		DIR\$RD
		ENT\$RD
		SGDR\$\$

Reading Directories (Command and Command Function): There is no mechanism by which directory entries can be read from command level or from command function level. This applies to both file and segment directories. (Directory contents can, of course, be displayed or written to a COMO file by using the LD command.)

Reading Directories (Subroutine): Your program can read file directories in several ways using any of the following subroutine calls:

DIR\$LS (*dir-unit, dir-type, initialize, desired-types, wild-ptr, wild-count, return-ptr, max-entries, entry-size, ent-returned, type-counts, before-date, after-date, code*)

DIR\$SE (*dir-unit, dir-type, initialize, sel-ptr, return-ptr, max-entries, entry-size, ent-returned, type-counts, max-type, code*)

DIR\$RD (*key, unit, return-ptr, max-return-len, code*)

ENT\$RD (*unit, name, return-ptr, max-return-len, code*)

DIR\$LS is a general-purpose directory searcher that takes arguments used to select entries to be searched for. Selection criteria can be object types, wild-card names, date and time last modified, or combinations of these. Selection can not be by date and time last accessed or date and time created. Either file or segment directories can be read. Selection can begin at the beginning of the directory or at the current position; entries are returned in a structure provided by the program that is capable of holding *max-entries* entries, and are pointed to by *return-ptr*. This call is fully described in the *Subroutines Reference II: File System*.

DIR\$SE extends the functionality of DIR\$LS by using a structure to contain additional selection criteria, including date and time last accessed and date and time created. DIR\$SE is fully described in the *Subroutines Reference II: File System*.

DIR\$RD reads the contents of a directory sequentially, one entry at a time, and returns each entry read in a program-provided structure pointed to by *return_ptr*. It returns only named file system objects, and therefore cannot be used to read subentries in a segment directory. It returns names for files, file directories, and access categories. This call is described more fully in Chapter 8, Data Storage and Retrieval, and in the *Subroutines Reference II: File System*.

ENT\$RD is used to read the contents of a specific directory entry whose name is given as the *name* argument. The entry is returned in a structure identical to that used by DIR\$RD, and pointed to by *return_ptr*. The entry being searched for must exist in the current directory, since *name* is defined as having a length of 32 characters. This call is described in detail in the *Subroutines Reference II: File System*.

Segment directories can be read by using either of the following calls

DIR\$LS (*dir-unit, dir-type, initialize, desired-types, wild_ptr, wild-count, return_ptr, max-entries, entry-size, ent-returned, type-counts, before-date, after-date, code*)

SGDR\$\$ (*key, unit, start_position, end_position, code*)

DIR\$LS is used as described for file directories, except that *dir-type* must have a value of 2 for a SAM segment directory, or 3 for a DAM segment directory.

SGDR\$\$ returns an integer representing the position in the directory of the first or next full or free position in the segment directory, depending on the values of *key* and *start_position*. *key* is K\$FULL or K\$FREE to look for full or free entries, respectively. A *start_position* value of zero (0) looks for the first entry; a value equal to the position of the last full or free entry plus 1 looks for the next entry. The position integer is returned in *end_position*. The SGDR\$\$ call is described in detail in Chapter 8, Data Storage and Retrieval, and in the *Subroutines Reference II: File System*.

Reading Files

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
None	READ_FILE	RDLIN\$ PRWF\$\$

Reading Files (Command): There are no commands that enable you to read a file directly from PRIMOS command level. However, a text file can be read indirectly and displayed to your terminal (or written to a COMO file), one line at a time, by using a TYPE command whose argument is a [READ_FILE] command function, described next.

Reading Files (Command Function): You can read an ASCII (text) file from a CPL program by including a statement of the form:

&SET_VAR *read_data* := [READ_FILE *unit status_var*]

In this CPL statement, *unit* is the decimal number of the file unit on which the file has been previously opened. You supply local or global variable names for the variables *read_data* and *status_var*. The former receives the line of text read from the file, while the latter stores the return code from the execution of the read. (The setting and evaluating of variables, and the use of the READ_FILE command function, are described in the *CPL User's Guide*).

Reading Files (Subroutine): To read a file from a program, use one of the following subroutine calls

RDLIN\$ (*unit, input_line, max_line_length, code*)

PRWF\$\$ (*key, unit, addr(buffer), size, pre_posn, halfwords_read, code*)

The RDLIN\$ call is used to read variable-sized records from a file open on *unit* into a buffer, pointed to by *input_line*. Reading ends when a new-line character is encountered. If the number of characters read is less than *max_line_length*, the remaining buffer characters are blank-filled. The RDLIN\$ calling sequence is illustrated in Chapter 7, Text Storage and Retrieval; the subroutine's operation is further explained in Chapter 7, Text Storage and Retrieval, and in the *Subroutines Reference II: File System*.

Use the PRWF\$\$ call to position and read fixed-length data files. Positioning and reading are only two of many functions that PRWF\$\$ can perform; its complete functionality is described in Chapter 7, Text Storage and Retrieval, and in the *Subroutines Reference II: File System*.

In addition to RDLIN\$ and PRWF\$\$, there are subroutines whose functions are to read from other than disk devices: RDASC reads ASCII characters from any device, while RDBIN reads binary data from any device. These subroutines are described in the *Subroutines Reference II: File System*.

Reading the Global Mount Table

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
LIST_MOUNTS	None	NAM\$L_GMT

Reading the Global Mount Table (Command): LIST_MOUNTS reads the contents of the Global Mount Table and returns a list of both the currently-mounted disk partitions and the currently-mounted portals which the calling program can access. You must be the System Administrator, or you must use the supervisor terminal, in order for NAM\$L_GMT to return the remote private partitions (partitions on other machines that were created with the ADDISK -PRIVATE command). The LIST_MOUNTS command is discussed in the *PRIMOS User's Release Document*.

Reading the Global Mount Table (Subroutine): NAM\$L_GMT reads the contents of the Global Mount Table and returns a list of both the currently-mounted disk partitions and the currently-mounted portals which the calling program can access. You must be the System Administrator, or you must use the supervisor terminal, in order for NAM\$L_GMT to return the remote private partitions (partitions on other machines that were created with the ADDISK -PRIVATE command).

NAM\$L_GMT(*index*, *ret_ptr*, *max_items*; *ret_items*, *code*)

- index* (fixed bin) A number that indicates the starting Global Mount Table entry in the list to be returned; use *index* when filling in the structure to which *ret_ptr* points. The GMT list of entries may be referenced as an array
[0 ... (N-1)]
where N is the total of the number of entries in the GMT. Use an array to call the NAM\$L_GMT subroutine as many times as there are GMT entries if the declaration of the structure is too small.
- ret_ptr* A pointer to the structure that NAM\$L_GMT fills in (the items for each GMT entry).
- max_items* (fixed bin) The maximum number of entries to be declared as GMT entries in the *index* field. If the *max_items* field is smaller than (N-1), structure overflow occurs.
- ret_items* (fixed bin) The number of entries filled in the structure.

code (fixed bin) The standard return code (NoError indicates successful completion).

BadIndex (error) No such entry exists at the index given in the GMT.

Writing File System Objects

Writing Directories		
<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
None	None	SGDR\$\$
		SGD\$DL

File and segment directory objects are most often written to implicitly, as a result of performing some function on a subordinate object that reflects a need to add or update control information in its containing directory. Each time a file open for writing is closed, for example, the date-time-last-modified information in the containing directory needs to be changed; this is done as an implicit byproduct of the close operation.

No writing to directories of either type can be done explicitly by commands or command functions, and only a limited number of writing operations can be done to directories at subroutine level, and these only on segment directories. Likewise, there are no commands by which you can explicitly write records to a file from command level; you can, however, write variable-length text records using a command function in a CPL program.

Write access is required on any object to be written to; Use access is required to all superior directories, and Add access is required to the containing directory if a previously nonexistent file is being written into that directory. (If the name of a file or other object in a directory is being changed, Delete as well as Add access is required on the containing directory.)

Writing Segment Directories (Subroutine): You can effectively write to a segment directory from program level by using the subroutine calls

SGDR\$\$ (*key, unit, new_size, ignored, code*)

SGD\$DL (*unit, code*)

The SGDR\$\$ call is used to extend or truncate a segment directory open on *unit* by specifying the *key* value K\$MSIZ and the new number of members in the *new_size* argument. The ignored argument is not used, and should be zero (0).

The SGD\$DL call is used to delete a member of the segment directory open on *unit*. If the member deleted is not the last member of the directory, effectively the size of the directory does not change; it changes only if the member deleted is the last one.

Both of these subroutines and their calling sequences are described in Chapter 8, Data Storage and Retrieval.

Writing Directories

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
None	WRITE_FILE	WTLIN\$ PRWF\$\$

Writing Files (Command): There is no direct command by which a text line or data file record can be written from command level. You can, however, write a text line using the WRITE_FILE command function described next.

Writing Files (Command Function): You can write text files (but not data files) from a CPL program by using the command function

[WRITE_FILE unit text]

The *unit* argument is the file unit number of a text file previously opened for writing or for reading and writing. The text to be written, represented by *text*, can be either literal text (enclosed in quotes if it contains spaces or special characters), or the current contents of a local or global variable previously set by a command function such as RESPONSE. Refer to the *CPL User's Guide* for further information on the WRITE_FILE command function.

Writing Files (Subroutine): To write a file from a program, use one of the following subroutine calls

WTLIN\$ (*unit, output_line, max_line_length, code*)

PRWF\$\$ (*key, unit, addr(buffer), size, rel_posn, lhalfwords_read, code*)

The WTLIN\$ call is used to write variable-sized (usually ASCII text) records to a file open on *unit* from a buffer, pointed to by *output_line*. Writing ends when a new-line character is encountered. If the number of characters written is less than *max_line_length*, the remaining characters in the buffer are blank-filled. The WTLIN\$ calling sequence is illustrated in Chapter 7, Text Storage and Retrieval; the subroutine's operation is further explained in Chapter 7, Text Storage and Retrieval, and in the *Subroutines Reference II: File System*.

Use the PRWF\$\$ call to position and write fixed-length data files. Positioning and writing are only two of many functions that PRWF\$\$ can perform; its complete functionality is described in Chapter 7, Text Storage and Retrieval, and in the *Subroutines Reference II: File System*.

In addition to WTLIN\$ and PRWF\$\$, there are subroutines whose functions are to write to other than disk devices: WRASC writes ASCII characters to any device, while WRBIN writes binary data to any device. These subroutines are described in the *Subroutines Reference IV: Libraries and I/O*.

Closing File System Objects

Any file system object that is capable of being opened from command, command function, or subroutine level is also capable of being closed. Objects can be closed only by the CLOSE command or a subroutine; there is no CLOSE_FILE command function to match the OPEN_FILE command function. However, the CLOSE command can be included in a CPL program either with or without the enclosing brackets ([]); the results are identical.

Closing File System Objects

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
CLOSE	CLOSE	CLO\$FU CLO\$FN SRCH\$\$ SRSFX\$ TSRC\$\$*

* The TSRC\$\$ subroutine is documented in Appendix A of the *Subroutines Reference II: File System*. It is considered obsolete at PRIMOS Rev. 20.2. Although TSRC\$\$ is still supported, programs should use SRSFX\$ beginning with Rev. 20.2.

Closing Objects (Command and Command Function): To close an object from command or command function level, use one of the following

CLOSE *objectname*

[CLOSE *objectname*]

objectname is any valid form of pathname. The CLOSE function does not return a code indicating that an object is not open; it does, however, return a code if the object is not found.

Closing Objects (Subroutine): To close a file system object from program level, use one of the subroutine calls

CLO\$FU (*unit, code*)

CLO\$FN (*pathname, code*)

SRCH\$\$ (*key, objectname, name_length, unit, type, code*)

SRSFX\$ (*key, pathname, unit, type, n-suffixes, suffix-list, basename, suffix-used, code*)

CLO\$FU and CLO\$FN are simplified interfaces to close file system objects by file unit number and pathname, respectively. Their calling sequences and operations are described more fully in Chapter 7, Text Storage and Retrieval.

SRCH\$\$ and SRSFX\$ both require a key value of K\$CLOS to close an object. SRCH\$\$ accepts only a simple objectname, and closes the named object in the current directory. SRSFX\$ can close an object anywhere in the file system (assuming appropriate access, of course). These subroutines are fully described in the *Subroutines Reference II: File System*.

See also the description of the CLOS\$A subroutine, part of the Application Library package, given in the *Subroutines Reference IV: Libraries and I/O*.

Deleting File System Objects

Any file system object that has been created, by whatever means, can also be deleted. Not all types of objects, however, can be deleted from all interface levels: you cannot, for example, delete an individual segment from a segment directory from command or command function level.

Delete access is required for the directory containing the object to be deleted; Use access is required for all superior directory levels.

Deleting File System Objects

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
DELETE	None	SGD\$DL
		SRCH\$\$

Deleting File System Objects

<i>Command</i>	<i>Command Function</i>	<i>Subroutine</i>
		SRSFX\$
		FIL\$DL
		TSRC\$\$ *
REMOVE_PORTAL	None	NAM\$RM_PORTAL

* The TSRC\$\$ subroutine is documented in Appendix A of the *Subroutines Reference II: File System*. It is considered obsolete at PRIMOS Rev. 20.2. Although TSRC\$\$ is still supported, programs should use SRSFX\$ beginning with Rev. 20.2.

Deleting Objects (Command): To delete a file, file directory, segment directory, or access category from command level, use the command:

DELETE *objectname [options]*

objectname is any valid form of pathname in which you have the appropriate access rights; you can therefore delete an object anywhere in the file system. The values that you can supply for *options* are described in the *PRIMOS Commands Reference Guide*.

Note that there is no abbreviated form of the DELETE command.

Deleting Objects (Command Function): There is no command function to delete a file system object. However, the DELETE command can be included in a CPL program.

Deleting Objects (Subroutine): To delete a file system object from a program, use one of the following subroutine calls

SGD\$DL (*unit, code*)

SRCH\$\$ (*key, objectname, nam_length, unit, type, code*)

SRSFX\$ (*key, pathname, unit, type, n-suffixes, suffix-list, basename, suffix-used, code*)

FIL\$DL (*pathname, code*)

The SGD\$DL call is used only to delete members of a segment directory. The program must first position to the desired segment number. See the section How to Position a Segment Directory in Chapter 8, Data Storage and Retrieval. The

unit argument gives the file unit number on which the segment directory was previously opened.

For SRCH\$\$ and SRSFX\$, the value of *key* is K\$DELE to delete an object. For SRCH\$\$, *objectname* is the simple name of an object in the current directory; if the object is a directory, the deletion occurs only if the directory is empty.

These calls are described further in Chapters 7, Text Storage and Retrieval and 8, Data Storage and Retrieval, and in *Subroutines Reference II: File System*.

Removing Portals (Command): To remove a portal from command level, use the command

`REMOVE_PORTAL mount_point_pathname [-HELP]`

mount_point_pathname is the fully-qualified pathname of the local directory where the portal is mounted.

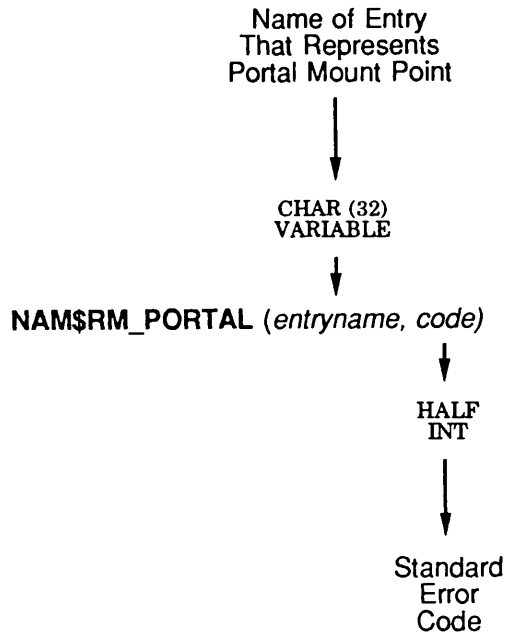
Removing Portals (Subroutine): To remove a portal by means of a program, use the following subroutine call

`NAM$RM_PORTAL (entryname, code)`

NAM\$RM_PORTAL deletes a portal entry in the specified directory pathname. This subroutine may only be used at the supervisor terminal. Figure 4-2 shows the calling sequence of NAM\$RM_PORTAL.

<i>entryname</i>	(char(32) var) The entry that represents the portal mount point.
<i>code</i>	(fixed bin) The standard return code. See the next section for a list of error codes.

Deleting a Portal



Q04.02.D10056.3LA

Figure 4-2. Calling Sequence of NAM\$RM_PORTAL

NAM\$RM_PORTAL Errors:

<i>Error Name</i>	<i>Description</i>
E\$SCCM	This routine may only be used at the supervisor terminal.
E\$WRIT	You do not have access rights for this operation.
E\$BNAM	The entryname that you specified uses incorrect syntax.
E\$FWTF	The specified portal was not found; delete operation failed.
E\$IREM	The specified portal mount must be on a local directory.

Search Rules

5



This chapter describes the PRIMOS search rules facility. It provides a conceptual overview of the search rules facility and describes how you can both modify system-supplied lists of search rules and create your own search lists. The search rules facility permits you to invoke a runtime search operation to locate an object, rather than specifying the exact location of the object. It is an important programming tool to enhance the generality, flexibility, and performance of many types of operations.

Search Rules and Search Lists

The PRIMOS search rules facility is a general-purpose mechanism for specifying a search sequence. It enables you to prespecify locations for PRIMOS to use when conducting a search. Each prespecified location is known as a **search rule**. A search rule names a location that may contain the object of the search. For example, a directory name would be a search rule when the object of the search is a file.

Search rules are grouped into sequences known as **search lists**. A search list is an area in memory that contains search rules, listed in sequential order. You initially write the sequence of search rules into a text file known as a **search rules file**. Before these search rules can be used, they must be copied from the search rules file into a search list. The process of copying search rules into a search list is known as setting the search list.

When using a search list, PRIMOS searches the first search rule in the search list, then the second search rule in the list, and so forth until PRIMOS either finds the object of the search or encounters the end of the search list.

One common use of search rules is to locate file system objects without requiring the user to enter the fully qualified pathname. You can create different search lists for different kinds of search operations. For example, you can establish a search list to search multiple disk partitions for a top-level directory or establish a search list to search multiple directories for a file.

You can invoke such a search by using a PRIMOS command, a CPL function, or a subroutine call. The `EXPAND_SEARCH_RULES` command, for example, takes a filename as input and uses the search rules facility to determine the

absolute pathname of that file. The search rules facility is invoked automatically by system software, such as the PRIMOS command processor and the BIND program linker.

PRIMOS maintains a separate group of search lists for each process. This means that users can customize their search lists to meet individual requirements. Because a group of search lists is specific to a process, a program uses the search lists of the user (or phantom) currently executing the program. To avoid possible mismatches between programs and search lists, you can include in the program calls to search rule subroutines that check or set your search lists. You cannot use, read, or set search lists that belong to other users' processes. The use of search lists is not affected by the user's current command level or attach point.

Default Search Lists

PRIMOS provides system default rules for five special-purpose search lists. These five search lists are included in the search lists of every user on the system. These search lists and their default rules are automatically set when a user logs in or otherwise initializes a process. The five special-purpose search lists are the following:

ATTACH\$	Searches directories to locate specified directory.
COMMAND\$	Searches directories to locate executable code files.
INCLUDE\$	Searches directories to locate source code files.
BINARY\$	Searches directories to locate binary object code files.
ENTRY\$	Searches EPF library files to locate entrypoints.

In addition to these five special-purpose search lists, you can set other, general-purpose search lists for the duration of a process. These search lists are referred to as **user-defined search lists**. During a process you can add, delete, or modify the search rules in any of your search lists. Search rules that you add to a search list (of any type) are referred to as **user-specified search rules**.

Advantages of Search Rules

Search rules provide several benefits:

- Search rules enable users to locate items at runtime without knowing their exact location. You specify this location information when you create the search list. When the search list is used, PRIMOS searches these listed locations for the object of the search. Once these search lists have been set, you do not have to specify (or even know) the full pathname in order to

retrieve each item. Naive users can be supplied with search lists that make knowledge of the file system architecture unnecessary.

- PRIMOS searches the rules in a search list in the listed order. By rearranging the search rules in a list, you can improve performance in searching for an item. This is particularly significant when searching multiple disk partitions for a directory.
- PRIMOS stops searching when it finds a match. Because a search operation uses the search rules to find the first occurrence of an item, you can maintain multiple items with identical filenames on the system and sequence the search rules to find the desired instance of that item. For example, if you have several revisions of the same file in different directories, you could list your search rules so that they always locate the directory containing the most recent version of the file. When you create a new version of the file, you simply add the name of that version's directory to the top of the search list.
- PRIMOS searches only those items that are specified in the search list. By changing the contents of a search list, you can restrict the scope of a search to only those locations where the desired item is likely to be found. For example, if a program always accesses a directory located on one of a small group of disk partitions, you would create a search list to search only those partitions, thus avoiding a search of all partitions on the system and preventing access to inappropriate directories.

The use of search rules can greatly simplify program and terminal operations, can increase the flexibility of programs and thus reduce maintenance overhead, and can improve the performance of search operations. However, note that failing to set a search list or modifying the rules in a search list can result in unexpected changes to the execution of programs.

Search Rule Types

A search list can consist of three types of search rules: administrator rules, system rules, and user-specified rules.

Administrator and System Search Rules

PRIMOS sets a group of search lists when you log in or otherwise initialize a process. These search lists are initialized with administrator search rules and system search rules. In each search list, administrator rules appear first, followed by system rules. PRIMOS assigns the same administrator and system rules to every process on the system.

Administrator search rules permit the System Administrator to regulate the use of search rules throughout the system. **System search rules** provide all users on the system with the same default search environment for normal PRIMOS operations. The search lists that PRIMOS sets when you initialize a process can contain just administrator rules, just system rules, or both administrator and system rules.

When you set a search list to user-specified rules, PRIMOS automatically prefaces your user-specified search rules with administrator and system rules. You can override the placement of system rules in a search list. You cannot override the placement of administrator rules in a search list.

Administrator and system search rules are located in search rules files found in directory SEARCH_RULES* on the command device. This directory provides search rules for ATTACH\$, COMMAND\$, ENTRY\$, BINARY\$, and INCLUDE\$. The System Administrator can modify these search rules files and can add administrator or system search rules files to this directory for other search lists. If either an administrator or system search rules file exists in SEARCH_RULES*, PRIMOS automatically sets a corresponding search list whenever a process is initialized. Refer to the *System Administrator's Guide, Volume I: System Configuration* for further details on administrator and system search rules.

User-specified Rules

You can specify new search rules to add to existing search lists. You can also specify search rules for new, user-defined search lists.

When adding rules to an existing search list, you can specify whether you wish the system rules to preface your user-specified rules (the default), to be excluded from the search list, or to be placed in a designated location in the search list. If administrator rules have been established for a search list, they always precede the user-specified rules and system rules. User-defined search lists have no corresponding administrator or system search rules.

Search List Types

PRIMOS permits you to create your own search lists. It also provides support for five special-purpose search lists: ATTACH\$, COMMAND\$, ENTRY\$, BINARY\$, and INCLUDE\$.

User-defined Lists

You can use a user-defined search list to search directories for file system objects (files, subdirectories, segment directories, and access categories). You

create a search list that consists of the pathnames of the directories that you wish to search for these file system objects. Each directory pathname is a separate search rule. The following are typical search rules for a user-defined search list.

```
glenn
glenn>project
alan>project
glenn>project>tests
glenn>status
```

How to create and name a user-defined search list is described later in this chapter, in the section named *Creating and Setting Search Rules*.

You can use the `EXPAND_SEARCH_RULES` (ESR) command or a subroutine call to search a user-defined search list. You specify the full name (name and suffix) of the file system object that is the object of the search, and the name of the search list. The ESR command returns the object's absolute pathname. The `OPSR$` and `OPSR$` subroutines locate and open the file.

You can use the `SR$SETL` subroutine to define the locator pointer values for rules in user-defined search lists. This advanced operation permits you to freely define the objects of a search. For further details, refer to the `SR$SETL` subroutine in the *Subroutines Reference II: File System*.

You must set user-defined search lists during the process in which they are used. User-defined search lists are automatically deleted at the conclusion of the process.

ATTACH\$

`ATTACH$` search rules let you predetermine the locations of file system objects when you use unqualified pathnames. Before Rev. 23.0, PRIMOS searched only the directories in the MFD of each specified disk partition. At Rev. 23.0, PRIMOS can search any directory, no matter at what level in the file system hierarchy the directory resides.

An `ATTACH$` search list is, in effect, a list of pathname prefixes. When encountering an unqualified pathname, PRIMOS transforms it into a fully-qualified pathname by using `ATTACH$`. PRIMOS adds each `ATTACH$` search rule, one at a time, to the beginning of the unqualified pathname and then checks the validity of the new pathname. If the now fully-qualified pathname is an actual file system object, the search is over. If the pathname is not valid, the search continues until the file system object is found or the `ATTACH$` search list is exhausted.

Before Rev. 23.0, the only valid `ATTACH$` search rules were disk partition names and valid keywords, including the special search rule called `-added_disks` (described later in the section *The -added_disks Keyword*). In order to make

<wrkdsk>myproj>mywork

The lower-level directory mywork is searched next. Note that this search rule is a fully-qualified pathname. This provides you with the ability to attach to lower levels in the file system hierarchy.

<bckdsk>

Another disk partition is searched next. <bckdsk> is interpreted as <bckdsk by PRIMOS.

If you have not set your own ATTACH\$ search list, the ATTACH\$ search list defined by the System Administrator is in effect. This list resides in the file <0>SEARCH_RULES*>ATTACH\$.SR on the command device. If this file does not exist, PRIMOS simply uses the added_disks keyword.

The EXPAND_SEARCH_RULES (ESR) command can be used to determine the result of using the ATTACH\$ search list to convert an unqualified pathname into a fully-qualified pathname. For example, issuing the command ESR MYDIR might yield the fully-qualified pathname <SYSDSK>MYDIR.

The ATTACH\$ search list can be invoked automatically by other search lists. This use of ATTACH\$ is described in the section ATTACH\$ Invoked by Other Search Lists.

COMMAND\$

You use the COMMAND\$ search list to search directories for command files. A command file is any executable code file, such as a runfile or CPL file. A COMMAND\$ search list should contain the pathnames of the directories that you wish to search for executable code files. The following are typical search rules for a COMMAND\$ search list:

```
cmdnc0
glenn
glenn>project
alan>project
glenn>project>tests
glenn>status
```

The default for COMMAND\$ is the directory CMDNC0, which contains the executable code files for PRIMOS commands. This default permits you to execute PRIMOS commands without supplying complete pathnames.

Once you have created a COMMAND\$ search list, you can execute a command file by simply typing its name, as if it were a PRIMOS command. For example, if you include the search rule mydir>subdir in your COMMAND\$ search list, you can execute the file mydir>subdir>myfile.run from any attach point by simply typing the value for myfile. You do not have to specify the RESUME

command or the filename suffix. PRIMOS searches each listed directory in sequence. PRIMOS stops searching when it finds the first file with the name you requested and (in order of preference) the suffix .RUN, .SAVE, .CPL, or a static-mode runfile with no suffix.

You can also use the EXPAND_SEARCH_RULES (ESR) command to search the COMMAND\$ search list. If you instruct ESR to use the COMMAND\$ search list, you do not have to specify the .RUN, .SAVE, or .CPL filename suffix. If you instruct ESR to find a filename with a .RUN, .SAVE, or .CPL suffix, you do not have to specify use of the COMMAND\$ search list. ESR returns the absolute pathname of the command.

INCLUDE\$

Some language compilers use the INCLUDE\$ search list to search directories for source code files that are to be included during program compilation. An INCLUDE\$ search list should contain the pathnames of the directories that you wish to search for source code files. The following are typical search rules for the INCLUDE\$ search list:

```
glenn  
glenn>tools  
glenn>project>tests  
alan>subsystem>tests
```

The compiler uses this search list when you specify the name of an include file during program compilation. You do not have to specify the filename suffix.

The following compilers support INCLUDE\$: F77, C, Pascal, CBL, VRPG, and PL/I. If no INCLUDE\$ search list is set, or a compiler does not support INCLUDE\$, the compiler assumes the include file is a source code file in the current directory. Refer to the individual language manuals for further details.

BINARY\$

The BIND linker uses the BINARY\$ search list to search directories for binary (.BIN) files. A BINARY\$ search list should contain the pathnames of the directories that you wish to search for binary files. The following are typical search rules for a BINARY\$ search list:

```
glenn  
glenn>compiles  
glenn>project>compiles  
alan>subsystem>compiles
```

When running BIND, you specify the filename of the BIND load file, and PRIMOS searches the directories listed in BINARY\$ for that file. You do not have to specify the .BIN filename suffix.

If no BINARY\$ search list is set, BIND assumes the load file is a binary file in the current directory.

ENTRY\$

You use the ENTRY\$ search list to search executable program format (EPF) or static-mode libraries for entrypoints. Each of these libraries can contain one or more entrypoints. The ENTRY\$ search list should contain the pathnames of the library files that you wish to search for entrypoints. The following are typical search rules for an ENTRY\$ search list.

```
-primos_direct_entries
LIBRARIES*>SYSTEM_LIBRARY.RUN
LIBRARIES*>TTYCK$.RUN
LIBRARIES*>FORTRAN_IO_LIBRARY.RUN
LIBRARIES*>PASCAL_LIBRARY.RUN
GLENN>PRIV_LIB.RUN
```

The ENTRY\$ search list is used automatically when you execute a program that contains a dynamic link to an entrypoint. This dynamic link is established using BIND. During the BIND operation, you use the -DYNT option to specify the name of the entrypoint. Then, during program execution, PRIMOS searches the libraries listed in ENTRY\$ for the named entrypoint. For further details on this use of ENTRY\$, refer to the *Programmer's Guide to BIND and EPFs*.

Creating and Setting Search Rules

Establishing user-specified search rules is a two-step process. First, you create a search rules file. A search rules file is a standard text file in which you write one or more search rules. After you create a search rules file, you use that file to set a search list. This set operation copies the rules in the search rules file into an area in memory established for the search list. All search operations are performed against the search list, not against the search rules file.

Creating a Search Rules File

You create a search rules file as a standard text file using EMACS or EDITOR. The naming conventions for search rules files are as follows:

- Use the name format: *xxx.listname.SR*. In this format, *xxx* can be any name, *listname* is the name of the search list, and *.SR* is a suffix indicating a search rules file.
- Do not use dollar signs (\$) in the listname of user-defined search rules files. Dollar signs are reserved for the listnames of special-purpose search rules files.

For example, you would use a search rules file with the name

```
mylist.command$.sr
```

to set the special-purpose search list `COMMAND$`. You would use a search rules file with the name

```
yourlist.lookup.sr
```

to set the search list `LOOKUP`.

You can create multiple search rules files that can be used to set the same search list. Only one file at a time can be used to set a particular list. (This file can, however, contain keywords that draw upon the contents of other search rules files.)

To place rules in a search rules file, use `EMACS` or `EDITOR` to specify one search rule per line in the sequence that the items should be searched. A search rule can be up to 128 characters in length. A search rule can include the disk partition name, or it can begin with the top-level directory. If the disk partition name is omitted, the search rules facility uses the `ATTACH$` search list to locate the appropriate partition. This use of `ATTACH$` is described later in the section `ATTACH$ Invoked by Other Search Lists`.

You can include comments, blank lines, and leading and trailing blanks in a search rules file. A comment begins with `/*` and continues to the end of the line. Comments and blanks in the search rules file are not copied into the search list during a set operation.

When creating a search rules file, you should avoid duplicating administrator rules or system rules in your file. The one exception to this is if you plan to override the automatic inclusion of system rules when you set the search list.

Setting Search Lists

A search rules file is used to set a search list. Search lists are set when

- You initialize a process
- You invoke a set operation

In both cases, the set operation copies search rules from one or more search rules files into an area in memory allocated for the search list. Because the set operation is a copy operation, the subsequent deletion or modification of the search rules file does not affect the search list.

When a process is initialized, PRIMOS automatically performs set operations that copy the search rules from the search rules files in the directory <0>SEARCH_RULES* into search lists in memory. This creates a group of default search lists for that process. PRIMOS sets each search list with search rules copied from the administrator search rules file and the system search rules file for that list. If one of these search rules files does not exist, PRIMOS sets the search list with the contents of whichever of these search rules files does exist. If a list has neither type of search rules file, no search list is set during process initialization.

You can set a search list by using the SET_SEARCH_RULES (SSR) command or the SR\$SSR subroutine. You supply the pathname of your search rules file to these set operations. You can also specify a name for the search list, or have the set operation derive the search list name from the name of the search rules file.

If the search list did not previously exist, the set operation creates that search list. If the search list did exist previously, the set operation either overwrites the old search rules or appends the new search rules to the search list. The set operation copies the rules in your search rules file into the search list. It may also copy administrator and system rules into the search list, if the appropriate search rules files are present in <0>SEARCH_RULES*.

A set operation does not check your search rules against the contents of the file system. Therefore, you can set search rules that refer to partitions, directories, and so on, that do not yet exist in your file system. When a search operation is performed, PRIMOS uses each rule in a search list independently. An invalid reference in one search rule does not affect other search rules or halt the search operation. If a search rule names a nonexistent object, PRIMOS proceeds to the next rule in the search list.

The SSR command returns a message if your search list has been set with duplicate rules. The SSR command sets the search list regardless of the presence of duplicate rules. A duplicate search rule in a search list can result in redundant searches but does not otherwise affect the search operation.

The SSR command has an option that permits you to reset a search list to system defaults. You can also use the SR\$INIT subroutine to reset search lists to system defaults. Other search rule subroutines are available to add or delete individual search lists and search rules. These subroutines act directly upon the search lists in memory and do not affect the corresponding search rules files.

Once you have set a search list, you can use the LIST_SEARCH_RULES (LSR) command to display the search list. You can also use the SR\$READ and SR\$NEXTR subroutines to read the rules set in a search list. The SSR and LSR commands are further described in the *PRIMOS Commands Reference Guide*.

Search rule subroutines are further described in the *Subroutines Reference II: File System*.

Search Rule Keywords

A search rules file can contain keywords that perform specific operations. Keywords that begin with a hyphen are directions to the search rules facility. These directions are carried out either when you set the search list or when you perform a search operation on that search list. Keywords enclosed in square brackets are variables for which the appropriate literal is supplied when the search list is used. The following are the available search rule keywords:

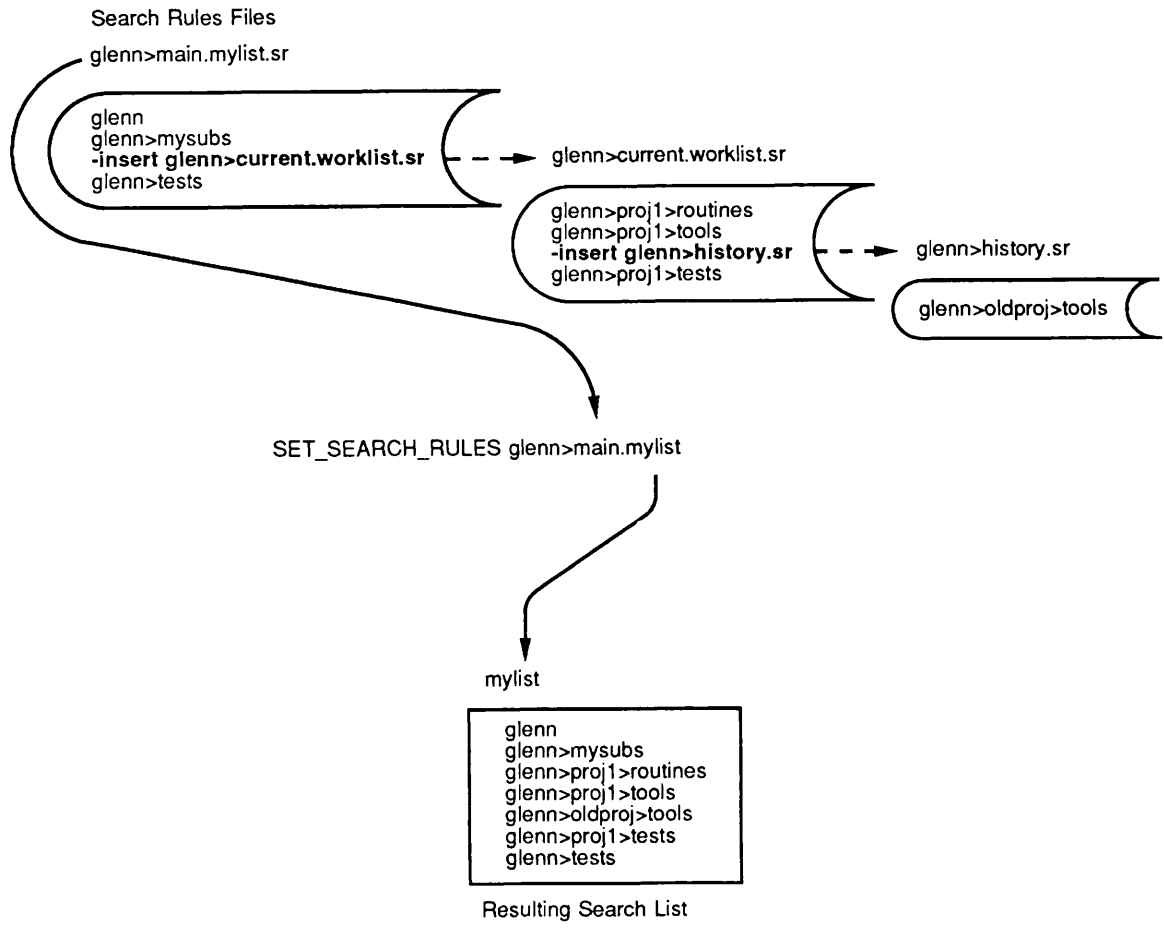
```
-insert  
-system  
-optional  
-added_disks  
-public  
-static_mode_libraries  
-primos_direct_entries  
[origin_dir]  
[home_dir]  
[referencing_dir]
```

You should place each keyword on its own line in a search rules file. Keywords and search rules can be intermixed in any sequence within a search rules file. Keywords can be written in either uppercase or lowercase.

The -insert Keyword

The `-insert` keyword specifies the pathname of another search rules file. When you set the search list, PRIMOS inserts the contents of that search rules file at the point indicated by the `-insert` keyword. By using this keyword, you can set a large search list using several small search rules files. Search rules files can be nested. The `SET_SEARCH_RULES` command rejects circular references, such as a search rules file that includes itself.

Figure 5-1 is an example of the `-insert` keyword. In this example, nested `-insert` keywords cause the contents of three search rules files to be included in the MYLIST search list.



Q05.01.D10056.3LA

Figure 5-1. Setting a Search List From Nested Search Rules Files Using the *-insert* Keyword

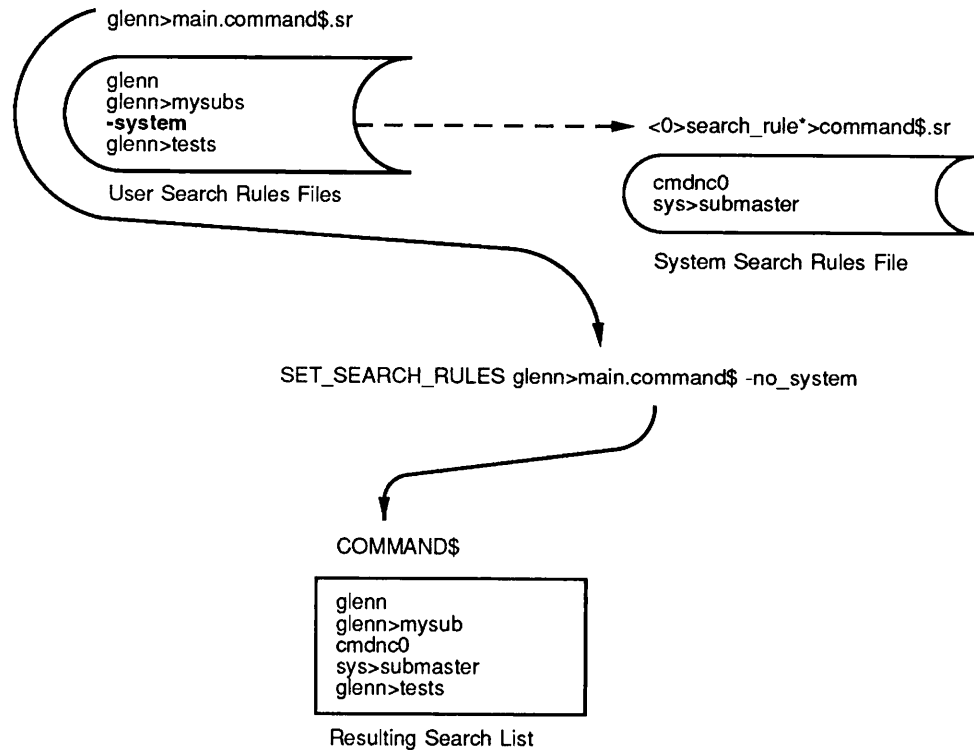
The *-system* Keyword

The *-system* keyword allows you to change the placement of system rules in a search list. By default, PRIMOS automatically places the system rules at the beginning of the search list. To place the system rules elsewhere in the search list, you specify the *-system* keyword at the desired location. When you set the search list, the complete sequence of system rules is placed in your search list at the location indicated by the *-system* keyword.

If you set the search list using the SET_SEARCH_RULES command, it is necessary to suppress the automatic inclusion of the system rules at the top of the list. To suppress automatic inclusion of system rules, use the

SET_SEARCH_RULES command `-no_system` option. If you set the search list using the SR\$SR subroutine, just specify the `-system` keyword at the desired location. You do not have to suppress inclusion of system rules at the beginning of the search list.

The example in Figure 5-2 inserts the system rules at the location indicated by the `-system` keyword. The SET_SEARCH_RULES `-no_system` option suppresses inclusion of the system rules at the beginning of the list.



Q05.02.D10056.3LA

Figure 5-2. Setting a Search List With User and System Default Search Rules Using the `-system` Keyword

If you do not suppress the prefacing of system rules (by using the SET_SEARCH_RULES `-no_system` option) PRIMOS ignores the `-system` keyword, and places the system rules at the beginning of the file.

Do not use the `-system` keyword in a search rules file for the ATTACH\$ search list. Instead, use the `-added_disks` keyword or set your own attach search rules to perform the equivalent operation.

The `–optional` Keyword

The `–optional` keyword specifies a rule that must be enabled before it can be used by PRIMOS. In your search rules file, you write the `–optional` keyword and the rule that must be enabled on the same line, as shown in the following search rules file:

```
glenn>tools
–optional glenn>tests
glenn>routines
```

When you set a search list, all optional search rules are disabled. PRIMOS skips over those rules when searching the list. The `LIST_SEARCH_RULES` command and most subroutines do not display the disabled search rules in the search list. For example, if you set a search list using the search rules file above, and then issue a `LIST_SEARCH_RULES` command for that search list, the following search rules are displayed:

```
glenn>tools
glenn>routines
```

You can enable optional search rules in a search list by using the `SR$ENABL` subroutine. When enabled, an optional search rule appears in the search list as an ordinary rule. For example, if you enable the `glenn>tests` optional search rule and then issue a `LIST_SEARCH_RULES` command, the following search rules are now displayed:

```
glenn>tools
glenn>tests
glenn>routines
```

Optional search rules can be set in any search list, including system and administrator search lists. You can specify any search rule or search rule keyword as an optional search rule, except for the keywords `–system` and `–insert`.

Optional rules in a search list can be repeatedly enabled and disabled. One application of optional search rules is to establish search rules that are used only by a particular program. You enable the optional rules at the beginning of program execution and disable the optional rules at the end of program execution. For further details, refer to `SR$ENABL` in the *Subroutines Reference II: File System*.

The `–added_disks` Keyword

The `–added_disks` keyword is used in `ATTACH$` search lists. Before Rev. 23.0, the PRIMOS file system used `–added_disks` to search potentially all of the added

disk partitions in order to resolve an unqualified pathname. PRIMOS did this by using the disk table in order to determine the list of disk partitions to search.

At Rev. 23.0, however, the file system name space is organized as a singly-rooted tree hierarchy, which has the ability to accept disk partitions mounted at different levels in its structure. This means that the use of `-added_disks` is no longer as straightforward as it once was. In addition, if the Name Server is running on your system, the number of added disk partitions in the common file system name space can grow to be quite large.

Therefore, the `-added_disks` keyword at Rev. 23.0 has different meanings, depending upon whether the Name Server is running or not. The different meanings of `-added_disks` are discussed in the sections following.

-added_disks Without the Name Server: If the Name Server is not running, the file system uses the disk table to determine which disks to search. First, the local disks are searched in the order in which they appear in the disk table. Second, the remote disks are searched in the order in which they appear in the disk table. This functionality is unchanged from previous revisions of PRIMOS, with one exception: at Rev. 23.0 and beyond, disk partitions that are not mounted in the root directory are not searched.

-added_disks With the Name Server: If the Name Server is running, the file system uses the disk table and the Global Mount Table (GMT) to determine which disks to search. (Use the `LIST_MOUNTS` command, described in the *PRIMOS User's Release Document*, to examine the contents of the GMT.) First, the local disks are searched in the order in which they appear in the disk table. Second, the file system uses the GMT to determine the remote disks to search.

There are several factors to consider when you are deciding upon the use of `-added_disks` on a system that has the Name Server running:

- The order of the GMT is determined by the Name Server's internal replication algorithm. This order may change over time as disks on local and remote systems are added and shut down. Thus, it is not possible to directly affect the order in which remote disks are searched. If your system is running the Name Server, and the order in which the disks are searched is important to you, then you should explicitly define those disks in an `ATTACH$` search list without the `-added_disks` keyword. If you do not define your own `ATTACH$` search list, then `-added_disks` is used by default.
- The disk partitions that are mounted at a level lower than the root directory are not searched. If you want lower-level mounts searched, then you must explicitly add them to your `ATTACH$` search list.
- If the Name Server is running, the number of disk partitions in the common file system name space is probably greater than the number of disks added manually to the local machine. At Rev. 23.0, a common file system name space can contain up to 1280 disks, and this size can affect the performance of attach-scan operations. The `-added_disks` keyword is usually specified

as the last search rule in an ATTACH\$ search list, and this keyword searches all of the disk partitions mounted in the root directory, including those that have already been searched using previous ATTACH\$ search rules.

In light of the above factors, it is recommended that you do not use the `-added_disks` keyword when the Name Server is running. Instead, use the ATTACH\$ search list without `-added_disks`.

The `-public` Keyword

Registered EPFs do not reside in the file system, but instead are contained in a special database. In order to be able to execute registered EPFs, use the `-public` search rule.

- To execute registered program EPFs, put `-public` in your COMMAND\$ search list so that PRIMOS searches the registered EPF database for command names.
- To execute an EPF that dynamically links to registered library EPFs, put the `-public` search rule in your ENTRY\$ search list, so that PRIMOS searches the registered library EPFs for entrynames.

PRIMOS searches the named registered EPFs using the order you have specified, then searches all other registered EPFs. For more information on the `-public` search rule, see the *Advanced Programmer's Guide I: BIND and EPFs*.

The `-static_mode_libraries` Keyword

The `-static_mode_libraries` keyword causes PRIMOS to search the static-mode libraries. The `-static_mode_libraries` keyword is only used in the ENTRY\$ search list. When you set an ENTRY\$ search list, the set operation copies the `-static_mode_libraries` keyword from the search rules file into the search list. When PRIMOS uses the ENTRY\$ search list and encounters the `-static_mode_libraries` keyword, it searches the static-mode libraries for the desired entrypoint. Refer to the *Programmer's Guide to BIND and EPFs* for further details on the use of ENTRY\$.

The `-primos_direct_entries` Keyword

The `-primos_direct_entries` keyword causes PRIMOS to search the PRIMOS system calls. The `-primos_direct_entries` keyword is only used in the ENTRY\$ search list. Normally, this keyword is set as an administrator rule in the ENTRY\$ search list. When PRIMOS uses the ENTRY\$ search list and encounters the `-primos_direct_entries` keyword, it searches the PRIMOS system

calls for the desired endpoint. Refer to the *Programmer's Guide to BIND and EPFs* for further details on the use of ENTRY\$.

The [origin_dir] Keyword

The [origin_dir] keyword causes PRIMOS to search the user's origin directory (that is, the user's initial attach point). This keyword is executed when the search list is used. When you set a search list, the set operation copies the [origin_dir] keyword from the search rules file into the search list. When PRIMOS uses the search list and encounters the [origin_dir] keyword, it searches the user's origin directory. The [origin_dir] keyword can be used in all search rules files (including search rules files for administrator and system rules) with the exception of ATTACH\$.

The [origin_dir] keyword can be used as a complete search rule or as a component of a pathname in a search rule, as shown in the following sample search rules file:

```
[origin_dir]
glenn>tools
[origin_dir]>tools
glenn>subr
```

The [home_dir] Keyword

The [home_dir] keyword causes PRIMOS to search the user's home directory (that is, the user's current attach point). This keyword is executed when the search list is used. When you set a search list, the set operation copies the [home_dir] keyword from the search rules file into the search list. When PRIMOS uses the search list and encounters the [home_dir] keyword, it searches the user's current attach point at the time of the search operation. The [home_dir] keyword can be used in all search rules files (including search rules files for administrator and system rules) with the exception of ATTACH\$. Using [home_dir] in the ENTRY\$ search list can produce unexpected results, and is therefore not recommended.

The [home_dir] keyword can be used as a complete search rule or as a component of a pathname in a search rule, as shown in the following sample search rules file:

```
[home_dir]
glenn>tools
[home_dir]>tools
glenn>subr
```

The [referencing_dir] Keyword

The [referencing_dir] keyword causes PRIMOS to search a pathname supplied by the user. When you set a search list, the set operation copies the [referencing_dir] keyword from the search rules file into the search list. When the search list is used, the operation that uses the search list should also supply a pathname to substitute for the [referencing_dir] keyword. If an operation that uses the search list does not supply a pathname, PRIMOS ignores the [referencing_dir] keyword and proceeds to the next rule in the search list. The [referencing_dir] keyword can be used in all search rules files (including search rules files for administrator and system rules) with the exception of ATTACH\$.

The EXPAND_SEARCH_RULES (ESR) command and the OPSR\$ and OPSRS\$ subroutines have optional arguments that supply a pathname to the [referencing_dir] keyword. PRIMOS substitutes this pathname for every instance of [referencing_dir] in the search list and then performs the search operation. The [referencing_dir] keywords revert to null values at the completion of the search operation.

Compilers that use the INCLUDE\$ search list automatically supply values to the [referencing_dir] keyword. For further details concerning the use of [referencing_dir] in INCLUDE\$ search lists, refer to the individual language manuals.

The [referencing_dir] keyword can be used as a complete search rule or as a component of a pathname in a search rule, as shown in the following sample search rules file:

```
[referencing_dir]
glenn>tools
[referencing_dir]>tools
glenn>subr
```

Accessing Search Lists

You can use search lists to conduct searches from the PRIMOS command environment, from CPL programs, or through subroutine calls from user programs. The five system-defined search lists are also accessible by specific system software. The ATTACH\$ search list can be accessed by other search lists.

PRIMOS Command Environment

The EXPAND_SEARCH_RULES (ESR) command uses a search list to locate the requested item and returns the absolute pathname of the object to the user's

terminal. When you issue an ESR command, you specify which search list should be used for the search. If you do not specify which search list to use, ESR selects a search list, based on the suffix of the sought item. If the object of the search is not located, ESR returns the value \$ERROR\$. The ESR command is further described in the *PRIMOS Commands Reference Guide*.

CPL Programs

EXPAND_SEARCH_RULES (ESR) can be issued as a CPL function from within a CPL program. The ESR CPL function has the same syntax and options as the ESR PRIMOS command. When issued as a CPL function, ESR returns the absolute pathname to a variable within the CPL program.

Program Subroutines

The search rules facility supports 18 search rule subroutines. Most of these subroutines perform operations on the search lists themselves. However, two subroutines (OPSR\$ and OPSR\$\$) use the search rules to locate and open a file. These two subroutines can also check for the existence of a file system object and, under certain circumstances, create a new file system object if the specified object does not exist.

The available search rule subroutines are as follows:

<i>Routine</i>	<i>Function</i>
OPSR\$	Locates a file using a search list and opens the file. Creates the file if the file sought does not exist.
SR\$ABSDS	Disables an optional search rule. Used to disable rules that have been enabled using SR\$ENABL. This subroutine absolutely disables an enabled rule, regardless of how many times the rule has been enabled. Compare with SR\$DSABL.
SR\$ADDB	Adds a rule to a search list before a specified rule.
SR\$ADDE	Adds a rule to the end of a search list, or after a specified rule.
SR\$CREAT	Creates a search list.
SR\$DEL	Deletes a search list.
SR\$DSABL	Disables an optional search rule that was enabled by SR\$ENABL. Disables a single SR\$ENABL operation. Compare with SR\$ABSDS.
SR\$ENABL	Enables an optional search rule. Enabled rules can be disabled using SR\$DSABL or SR\$ABSDS.

SR\$EXSTR	Determines if a search rule exists.
SR\$FR_LS	Frees list structure space allocated by SR\$LIST or SR\$READ.
SR\$INIT	Initializes all search lists to system defaults.
SR\$LIST	Returns the names of all defined search lists.
SR\$NEXTR	Reads the next rule from a search list.
SR\$READ	Reads all of the rules in a search list.
SR\$REM	Removes a search rule from a search list.
SR\$SETL	Sets the locator pointer for a search rule.
SR\$SSR	Sets a search list using a user-defined search rules file.

These subroutines are further described in the *Subroutines Reference II: File System*.

ATTACH\$ Invoked by Other Search Lists

The only search list which requires that pathnames be fully-qualified is the ATTACH\$ search list. The other system search lists can contain pathnames which are unqualified. In order to resolve unqualified pathnames found in other system search lists, PRIMOS uses the ATTACH\$ search list.

Adding unqualified pathnames to search lists can greatly affect their performance, especially if there are many partitions to search. Searching for a file system object with an unqualified pathname is always slower. You should carefully consider the tradeoff between the flexibility of unqualified pathnames and the better performance of fully-qualified pathnames.

Attach Points

6



This chapter describes, in detail, the initial, home, and current attach points, and then describes subroutines that are used to manipulate attach points.

The Initial Attach Point

When a new user is added to the system, the System Administrator or the Project Administrator specifies an initial attach point and usually creates an origin directory for the new user.

PRIMOS attaches the user's process to the origin directory during the login procedure; when the procedure terminates, the user's initial, home, and current attach points are all set to the origin directory, unless the login procedure itself (or an external program that it may call) has changed the home or current attach point, or both.

During a terminal session, the user may reset his or her home and current attach points to the origin directory by issuing the ORIGIN command. Your program may also reset the home and current attach points by using the AT\$OR subroutine. The AT\$OR subroutine allows your program to reset just the current point or both the current and home attach points to the origin directory. Figure 6-1 illustrates the calling sequence for the AT\$OR subroutine.

If the *key* argument is K\$SETH, both the home and current attach points are reset to the origin directory. If the *key* argument is K\$SETC, only the current attach point is reset to the origin directory.

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$NATT	7	No top-level directory attached. This error usually occurs only when the disk on which the origin directory resides has been removed from the system, as in disk shut down. Once a disk has been shut down, all origin directories residing on that disk are lost. These users can reestablish their origin directories only by logging in after start up.
E\$SHDN	121	The disk has been shut down. The disk on which the origin directory resides has been shut down. The disk is no longer available for use until the System Operator uses the ADDISK command to add the disk again. After this is done, the user must log in again to reestablish his or her origin directory.

The Home Attach Point

The home attach point essentially identifies the user's working directory. Initially, following user login, the home attach point is the same as the initial attach point. To reset the current attach point to the home attach point from within your program, use the AT\$HOM subroutine, shown in Figure 6-2.

Reset Current Attach Point to Home Directory

AT\$HOM (*code*)



HALF
INT



Standard
Error
Code

Q06.D2.D10056.3LA

Figure 6-2. Calling Sequence of AT\$HOM

An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was entirely successful. Otherwise, *code* is always positive. After a call to AT\$HOM to attach to the home directory, *code* may have one of many values. The Advanced Programmer's Guide: Appendices and Master Index contains a comprehensive list of all standard file system error codes.

Error codes specific to the AT\$HOM subroutine are

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$NATT	7	No top-level directory attached. This error usually occurs only when the disk on which the home directory resides has been removed from the system, as when a disk is shut down. Once a disk has been shut down, all home directories residing on that disk for all currently logged-in users are lost. These home directories can be reestablished by the users only by issuing an ATTACH command after the disk is started up again.
E\$SHDN	121	The disk has been shut down. The disk on which the home directory resides has been shut down (using the SHUTDN command as described in the <i>Operator's Guide to System Commands</i>). The disk is no longer available for use, until the System Operator uses the ADDISK command to add the disk again. After this is done, the user must issue the ATTACH command again to reestablish his or her home directory.

The Current Attach Point

The current attach point is essentially the program's working directory. Initially, the current attach point is the same as the initial and home attach points. A program can change the current attach point by calling one of many file system subroutines:

<i>Subroutine</i>	<i>Use</i>
AT\$	Attaches the current (optionally home) attach point to the directory specified by pathname. Similar to the ATTACH <i>pathname</i> command.
AT\$ABS	Attaches the current (optionally home) attach point to the specified directory on the specified root entry disk partition. Similar to the ATTACH <i><partition>dirname</i> command.

- AT\$ANY** Attaches the current (optionally home) attach point to the specified directory on the first disk partition found to have the directory that was specified. The search is done using the `ATTACH$` search rules. Similar to the `ATTACH dirname` command.
- AT\$HOM** Attaches the current attach point to the home directory, as described earlier in this chapter. Similar to the `ATTACH` command.
- AT\$OR** Attaches the current (optionally home) attach point to the origin directory, as described earlier in this chapter. Similar to the `ORIGIN` command.
- AT\$REL** Attaches the current (optionally home) attach point to the specified lower-level directory of the current directory. Used to attach downward in a directory tree. Similar to the `ATTACH *>dirname` command.
- AT\$ROOT** Attaches the current (optionally home) attach point to the root directory. Synonymous with the `ATTACH <` command.

All of the above subroutines replace an obsolete subroutine named `ATCH$$` that performed all of the attach functions in one (rather complicated) interface. The subroutines listed above are described later in this chapter; the `ATCH$$` subroutine is described in detail in Appendix A of *Subroutines Reference II: File System*.

Operations That Reset the Current Attach Point

Because the current attach point is used in so many file system operations, it is often reset even when errors occur. For example, if a call to `AT$` is made to set the current attach point to `FRODO>FINGER>FOOD`, and the `FINGER` lower-level directory does not exist in the `FRODO` directory, an error code of `E$FNTF` (Not found) is returned, and the current attach point is reset to the home directory, independent of what it was before the call was made.

Similarly, a mistyped command resets the current attach point to the home directory. In fact, the only way to avoid resetting the current attach point while at `PRIMOS` command level is to use only internal commands, such as `OPEN`, `STATUS`, `DUMP_STACK`, and so on. (The *PRIMOS Command Reference Guide* lists internal commands.)

Commands such as `LD`, `DELETE`, `COPY`, `EMACS`, and `USAGE` reset the current attach point. In most cases, resetting the current attach point is usually not a problem. Resetting the current attach point is a problem if a program activation has been suspended (as with, for example, `Control-P`) just when the current attach point is different from the home attach point. In this case, restarting the suspended program may produce irrational behavior. Programs

that make heavy use of the current attach point can expect to encounter problems resulting from program interruptions; even programs that do not explicitly use the current attach point can possibly encounter problems when calling subroutines that handle pathnames (such as SRSFX\$), because these subroutines use the current attach point and may also be interrupted.

In addition, anytime a pathname is processed by the file system, the current attach point is reset to the home directory. For example, if the DIR\$CR subroutine, described in Chapter 8, Data Storage and Retrieval, is called with the pathname FRODO>THUMB, the current attach point is implicitly reset to the home directory.

File system subroutines that accept filenames but not pathnames assume that the specified file is in the current directory. Similar subroutines perform their operations in the current directory, although they do not actually accept filenames as arguments. In both cases, these subroutines are frequently referred to as file system **primitives**. The use of these primitives rarely changes the current attach point. Among the PRIMOS file system primitives are the following subroutines:

AC\$RVT	PHANT\$
CNAM\$\$	PHNTM\$
COMI\$\$	REST\$\$
COMO\$\$	RESU\$\$
FIL\$DL	SATR\$\$
GPAS\$\$	SAVE\$\$
GPATH\$	SPAS\$\$
SRCH\$\$	

Note CREA\$\$ and CREPW\$, which accept only filenames, are considered obsolete at PRIMOS Rev. 20.2. Although CREA\$\$ and CREPW\$ are still supported, programs should use DIR\$CR, which accepts pathnames, beginning with Rev. 20.2.

All other subroutines that operate either explicitly or implicitly on a pathname (any file system name containing at least one < or > character) reset or change the current attach point.

Functions Used To Manipulate Attach Points

In addition to the subroutines described earlier in this chapter, several subroutines are provided to allow a running program to manipulate the user's attach points. These are:

<i>Subroutine</i>	<i>Action</i>
AT\$	Attaches to a pathname
AT\$ABS	Attaches to a directory on a specified disk partition
AT\$ANY	Attaches to a directory on any started-up disk partition
AT\$REL	Attaches to a subordinate directory (relative attach)
AT\$ROOT	Attaches to the root directory
GPATH\$	Returns the complete pathname of the initial, home, or current directories
SRCH\$\$	Opens the current directory for reading

The AT\$ Subroutine

To attach to a specific directory by pathname, use the AT\$ subroutine. The AT\$ subroutine parses a pathname, and passes the call to the appropriate AT\$-type subroutine (AT\$ABS, AT\$ANY, AT\$HOM, AT\$ROOT, or AT\$REL, described below) to perform the actual attaching.

The AT\$ subroutine may be used to change only the current directory or both the home and current directories. It may return any of the error codes that the other four subroutines can return, with one additional error code — E\$ITRE (Illegal treename). This error code indicates an invalid pathname.

The subroutine to which the call is passed by AT\$ depends on the form of the pathname. The several forms and their corresponding implementations are:

<i>Form</i>	<i>Result</i>
<	Passed to AT\$ROOT to attach to the root directory.
<*>	Passed to AT\$ABS to attach to the current partition's MFD (the MFD containing the home directory in effect at the time of the AT\$ call). The < in this special syntax does <i>not</i> indicate the root.
<dir>...	
<dir	Passed to AT\$ROOT to attach to the specified disk partition, followed by calls to AT\$REL to attach to directories following the <dir> portion of the pathname.
*>...	Passed to AT\$HOM to attach to the home directory, followed by calls to AT\$REL to attach downward.
dir	Passed to AT\$ANY to attach to a top-level directory, that is, a directory immediately subordinate to a partition's MFD.
dir>...	Passed to AT\$ANY to attach to an absolute pathname, the first element being a top-level directory.
(null)	A null pathname has the same effect as using the AT\$HOM call, described later in this chapter.

Note PRIMOS treats a single (simple) objectname in one of two ways, depending upon whether or not the objectname is a directory. When you use the ATTACH command with simple object name, that object is a root directory entry. With other commands, a simple objectname identifies that object as a file in the current directory.

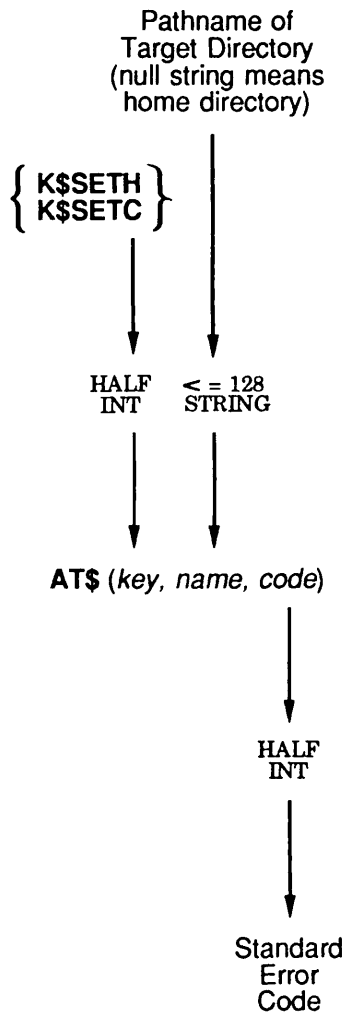
This distinction is seen when comparing the following two PRIMOS commands:

```
ATTACH FRODO
SLIST FRODO
```

The ATTACH command searches all root directory entries looking for an entry named FRODO. The SLIST command searches for a file named FRODO in the home (current) directory. (When the SLIST command is issued, the current attach point is reset to the home directory by the operation of searching the command directory, CMDNC0, for the SLIST program.)

Figure 6-3 illustrates the calling sequence of AT\$.

Attach to Directory by Pathname



Q06.03.D10056.3LA

Figure 6-3. Calling Sequence of AT\$

The AT\$ABS Subroutine

Before Rev. 23.0, AT\$ABS allowed you to set the attach point to a specified top-level directory on a given partition. At Rev. 23.0, however, a disk partition may be logically mounted anywhere on the file system tree, not just directly below the root. This means the directory you are seeking may no longer be “top-level.” Therefore, AT\$ABS cannot always be used to reference top-level directories.

In the pre-Rev. 23.0 file system, the partition and directory name arguments were treated as if the pathname <partition_name>directory_name was used. This is still true at Rev. 23.0 as long as the disk partition is mounted in the root directory as directory partition_name. However, if the disk is mounted in a directory below the root directory, you cannot use AT\$ABS to attach to top-level directories on that lower-mounted partition. Instead, use AT\$.

You may specify the root-entry argument to the AT\$ABS subroutine by using any of the following:

- A partition whose name is mounted in the root
- The partition on which the current directory resides
- The partition corresponding to logical disk 0
- The partition corresponding to a particular logical disk number

When your program calls AT\$ABS, it provides:

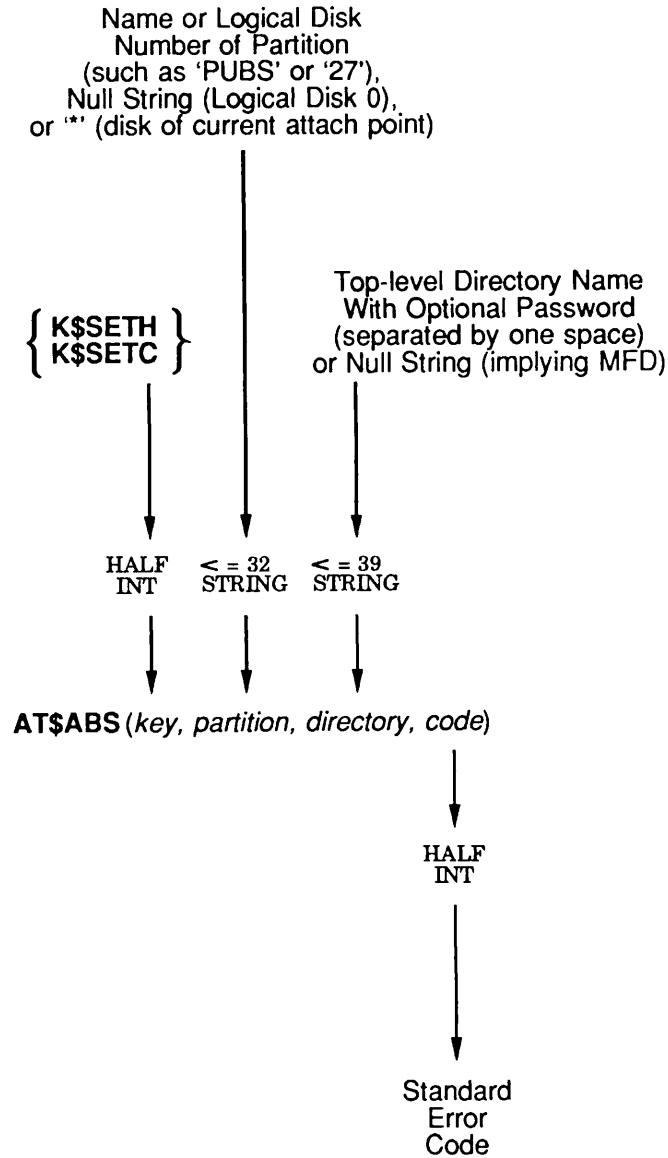
- A key that specifies whether the home attach point is to be set
- The identity of the directory's partition, in any of the forms listed above
- The name of the directory itself

The AT\$ABS subroutine attempts to set the current attach point to the specified directory on the specified partition, and returns a code indicating whether the operation was successful. You supply the partition argument and the directory argument, and AT\$ABS supplies the root-directory symbol (<) and the subordinate object symbol (>).

If the operation fails, no changes are made to the attach points. If the operation succeeds, the home attach point is also set to the current attach point if specified by the key.

Figure 6-4 illustrates the calling sequence of the AT\$ABS subroutine.

Attach to Top-level Directory of Specified Partition



Q06.D4.D10056.3LA

Figure 6-4. Calling Sequence of AT\$ABS

The Key: Your program sets *key* to one of the following:

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
K\$SETC	0	Set only the current attach point.
K\$SETH	1	Set both the current and home attach points.

The Partition: Your program passes, as a character string, the name of the directory which represents the partition. A null partition name specifies logical disk 0 (the command device). A partition name of * specifies the partition on which the current directory resides. (Note that * means the root when you are attached to the root.) A character string that is an unsigned octal number specifies the logical disk number. Otherwise, *part_name* identifies the desired partition. Since *part_name* is the name of a root entry directory, its name can contain up to 32 characters.

The Directory: Your program passes the name of the directory to attach to as a character string. To specify a password, append it to the directory name with a single space separating the directory name and the password.

If your program passes a null directory name, AT\$ABS attaches to the MFD of the specified partition.

The Error Code: An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was entirely successful. Otherwise, *code* is always positive. After a call to AT\$ABS to attach to a directory, *code* may have one of many values. The Advanced Programmer's Guide: Appendices and Master Index contains a comprehensive list of all standard file system error codes. Error codes specific to this operation are

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$BPAR	6	Bad parameter. The length of the directory name as passed by the calling program is a negative number or is greater than 39 (including an optional directory password).
E\$NATT	7	No top-level directory attached. This error usually occurs when the partition name is * and the partition on which the current directory resides is removed from the system, as when a disk is shut down. Use one of the subroutines described in this chapter to reestablish a current attach point.
E\$FNTE	15	Not found. The specified partition does not exist, or the specified directory does not exist on that partition.
E\$BNAM	17	Illegal name. The partition name must be between 0 and 32 characters in length. The directory name must also be between 0 and 32 characters in length (inclusive), optionally followed by a single space and a password from 1 to 6 characters long (inclusive).

Example: The following PL/I statement sets the home and current attach points to the directory named ORANGE on the partition named RHYMES:

```
call at$abs(k$seth,'RHYMES','ORANGE',code);
```

The AT\$ANY Subroutine

AT\$ANY attaches to the first instance of a particular directory by searching through a list of directories. AT\$ANY uses ATTACH\$ search rules to determine which directories are examined to find the specified directory. (See the discussion on ATTACH\$ search rules in Chapter 5, Search Rules.) Before Rev. 23.0, this list of directories could only be MFDs on specified disk partitions. At Rev. 23.0 and beyond, the list of directories can be directories at any level in the file system hierarchy.

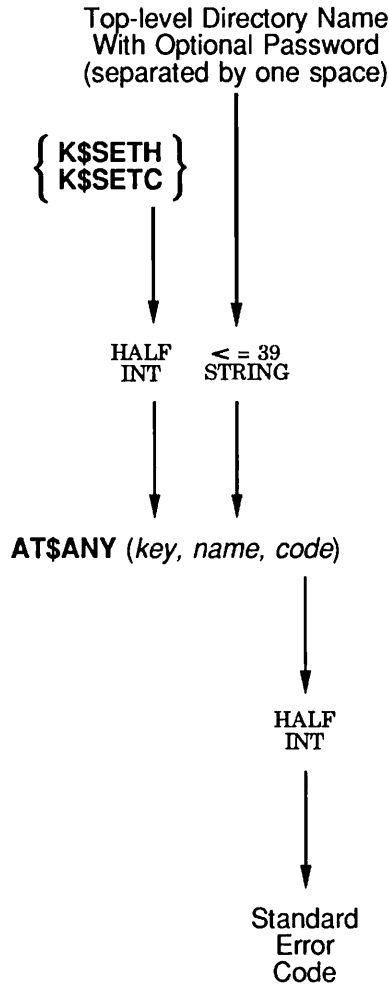
When calling AT\$ANY, your program provides

- A key that specifies whether the home attach point is to be set
- The name of the directory to search for

The AT\$ANY subroutine attempts to set the current attach point to the specified directory in the first directory it finds that has the specified directory. It returns a code indicating whether the operation was successful. If the operation fails, no changes are made to the attach points. If the operation succeeds, the home attach point is also set to the current attach point if specified by the key.

Figure 6-5 illustrates the calling sequence of the AT\$ANY subroutine.

Attach to Top-level Directory of Any Partition



Q06.05.D10056.3LA

Figure 6-5. Calling Sequence of AT\$ANY

The Key: Your program sets *key* to one of the following:

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
K\$SETC	0	Set only the current attach point.
K\$SETH	1	Set both the current and home attach points

The Directory: Your program passes the name of the directory to attach to as a character string. To specify a password, append it to the directory name with a single space separating the directory name and the password.

The Error Code: An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was entirely successful. Otherwise, *code* is always positive. After a call to AT\$ANY to attach to a partition-level directory, *code* may have one of many values. The Advanced Programmer's Guide: Appendices and Master Index contains a comprehensive list of all standard file system error codes. Error codes specific to this operation are:

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$BPAR	6	Bad parameter. The length of the directory name as passed by the calling program is a negative number or is greater than 39 (including an optional directory password).
E\$BNAM	17	Illegal name. The syntax of the directory name as supplied by the calling program is not correct. The directory name must be between 0 and 32 characters in length, optionally followed by a single space and a password. See the <i>PRIMOS User's Guide</i> for a description of the legal syntax for objectnames.
E\$NFAS	189	Top-level directory not found or inaccessible. The specified directory could not be found, or resides on a disk partition that cannot be accessed by the user.

Example: The following PL/I statement sets the home and current attach points to the directory named ORANGE on the first partition found to contain a directory named ORANGE:

```
call at$any(k$seth, 'ORANGE', code);
```

The AT\$REL Subroutine

Use the AT\$REL subroutine to attach down to a directory that is subordinate to the current directory. The subroutine searches through the current directory for the specified lower-level directory, and attaches to it as the new current (and optionally home) directory.

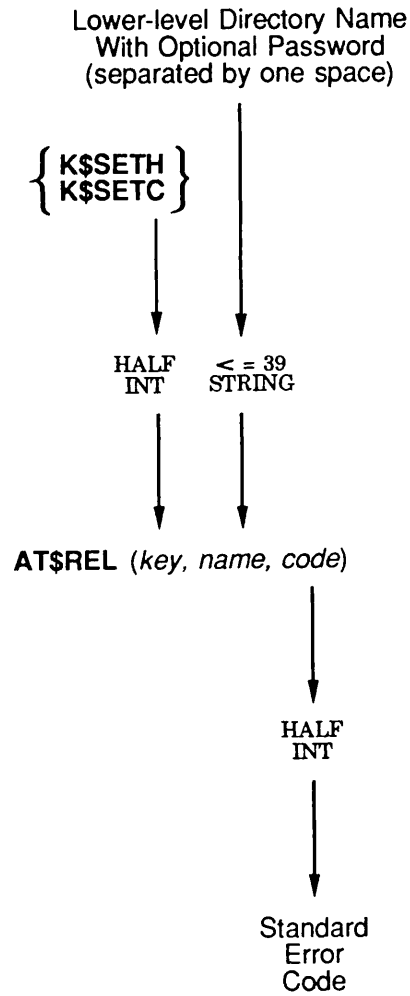
When calling AT\$REL, your program provides

- A key that specifies whether the home attach point is to be set
- The lower-level directory to be attached to

The AT\$REL subroutine attempts to set the current attach point to the specified lower-level directory of the current directory, and returns a code indicating success or failure. If the operation fails, the attach points are not changed. If the operation succeeds, the home attach point is also set to the current attach point if specified by the key.

Figure 6-6 illustrates the calling sequence of the AT\$REL subroutine.

Attach to Subdirectory of Current Directory



Q06.06.D10056.3LA

Figure 6-6. Calling Sequence of AT\$REL

The Key: Your program sets *key* to one of the following

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
K\$SETC	0	Set only the current attach point.
K\$SETH	1	Set both the current and home attach points.

The Lower-level Directory: Your program passes the name of the lower-level directory to attach to as a character string. To specify a password, append it to the lower-level directory name with a single space separating the lower-level directory name and the password.

The Error Code: An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was entirely successful. Otherwise, *code* is always positive. After a call to AT\$REL to attach to a lower-level directory, *code* may have one of many values. The *Advanced Programmer's Guide: Appendices and Master Index* contains a comprehensive list of all standard file system error codes. Error codes specific to this operation are

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$BPAR	6	Bad parameter. The length of the lower-level directory name as passed by the calling program is a negative number or is greater than 39 (including an optional lower-level directory password).
E\$NATT	7	No top-level directory attached. This error can occur only when the partition on which the current directory resides is removed from the system, as when a disk is shut down. Use one of the AT\$ subroutines to reestablish a current attach point.
E\$BNAM	17	Illegal name. The syntax of the lower-level directory name as supplied by the calling program is not correct. The lower-level directory name must be between 0 and 32 characters in length, optionally followed by a single space and a password. See the <i>PRIMOS User's Guide</i> for a description of the legal syntax for objectnames.

Example: The following PL/I statement sets the home and current attach points to the lower-level directory named JUICE of the current directory:

```
call at$rel(k$sseth, 'JUICE', code);
```

The AT\$ROOT Subroutine

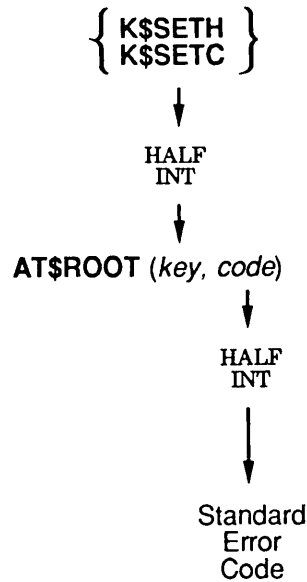
AT\$ROOT lets you attach to the root directory by means of the calling program. Figure 6-7 illustrates the calling sequence of AT\$ROOT.

The Key: Your program sets *key* to one of the following

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
K\$SETC	0	Set only the current attach point.
K\$SETH	1	Set both the current and home attach points.

The Error Code: An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was successful.

Attach to the Root Directory



Q06.07.D10056.3LA

Figure 6-7. Calling Sequence of AT\$ROOT

Example: The following PL/I statement sets the current attach point to the root directory:

```
call at$root(k$setc,code);
```

The GPATH\$ Subroutine

It is sometimes useful for your program to be able to determine the full pathname of the initial, home, or current directories. The GPATH\$ subroutine provides this function. This subroutine is also capable of determining the full pathname of a file open on any file unit, including the command output unit. File numbers for member files within segment directories are returned when appropriate.

To determine the full pathname of one of the three directories, your program calls GPATH\$ and provides it with

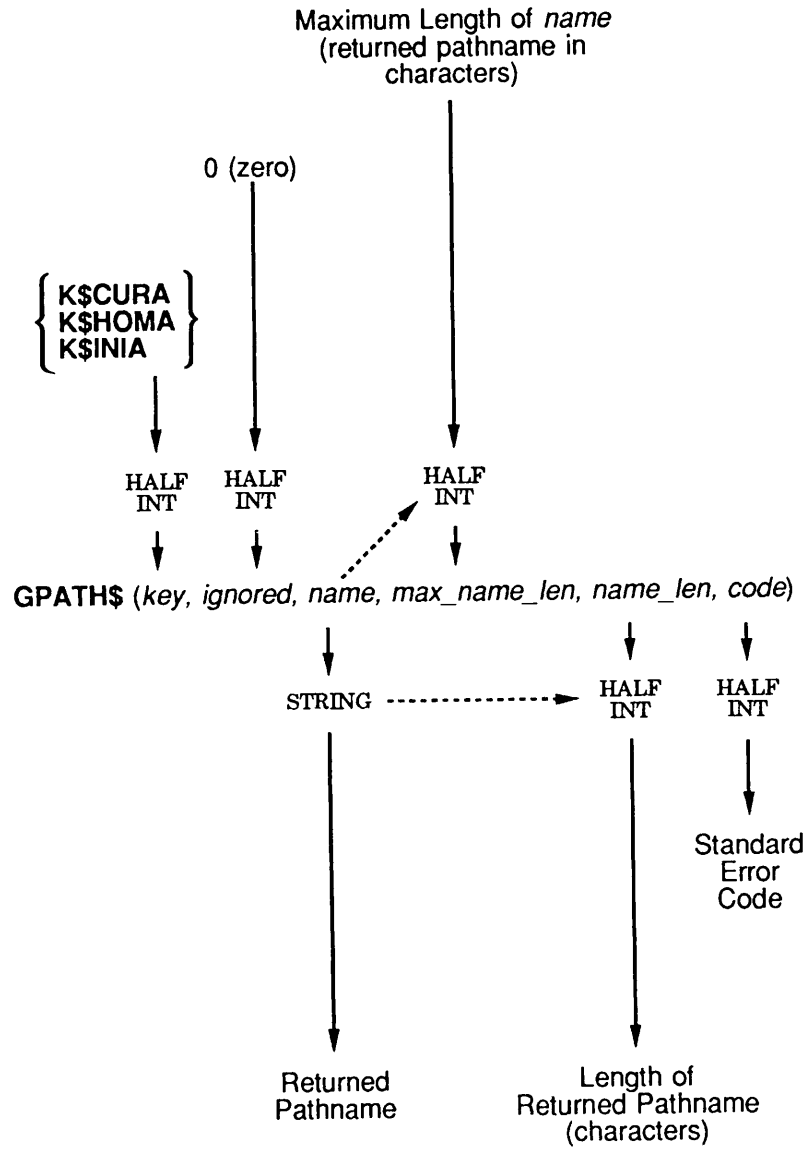
- A key that specifies which directory pathname is to be obtained
- The size of the buffer into which the pathname is to be stored

The GPATH\$ subroutine determines the appropriate directory pathname and returns to your program

- A buffer containing the resulting pathname
- The actual length of the pathname
- An error code indicating whether the operation was successful

Figure 6–8 illustrates the calling sequence for the GPATH\$ subroutine to determine the pathname of one of the three attach points.

Determine Pathname of an Attach Point



Q06.D8.D10056.3LA

Figure 6-8. Calling Sequence of GPATH\$ to Determine the Pathname of an Attach Point

The Key: Your program sets the *key* argument to one of three values:

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
K\$CURA	2	Determine the pathname of the current directory.
K\$HOMA	3	Determine the pathname of the home directory.
K\$INIA	4	Determine the pathname of the initial directory.

Maximum Size of the Returned Pathname: Your program sets the *max_name_len* argument to the size of the *name* argument in bytes. If the resulting pathname is longer than *max_name_len* characters, the operation fails, and an error code of E\$BFTS is returned.

The Returned Pathname: The GPATH\$ subroutine sets the *name* argument to the resulting pathname if the operation succeeds (*code* is 0). GPATH\$ stores the operational length of the returned pathname in *name_len*. No characters beyond character number *name_len* in *name* contain valid data.

The Actual Length of the Returned Pathname: GPATH\$ sets the *name_len* argument to the length of the resulting pathname in bytes if the operation succeeds (*code* is 0).

The Error Code: An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was entirely successful. Otherwise, *code* is always positive. After a call to GPATH\$ to determine the pathname of an attach point, *code* may have one of many values. The *Advanced Programmer's Guide: Appendices and Master Index* contains a comprehensive list of all standard file system error codes. Error codes specific to this operation are

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$NATT	7	No top-level directory attached. This error usually occurs only when the directory to which the user is attached is removed from the system, as when a disk is shut down. Use one of the subroutines described in this chapter to reestablish a current attach point.
E\$BFTS	35	Buffer too small. The supplied buffer is too small to hold the information. The <i>buffer</i> argument contains no useful data.
E\$PTHU	357	Pathname unavailable. In certain cases it is possible that GPATH\$ may not return the desired pathname. For example, if GPATH\$ encounters a remote portal reference to another directory, and that reference has not been propagated to the GMT, the Path Unavailable error is returned.

Example: The following PL/I statements display the full pathname of the home directory:

```
call gpath$(k$homa,0,pathname,80,pathlen,code);
if code=0 then call tnou(pathname,pathlen);
  else call errpr$(k$irtn,code,'Cannot get home pathname',24,
    'MYPROGRAM',9);
```

The SRCH\$\$ Subroutine

Use the SRCH\$\$ subroutine to open the current directory. When calling SRCH\$\$, your program provides

- An indicator that the current directory is being opened
- A key that specifies how the directory is to be opened

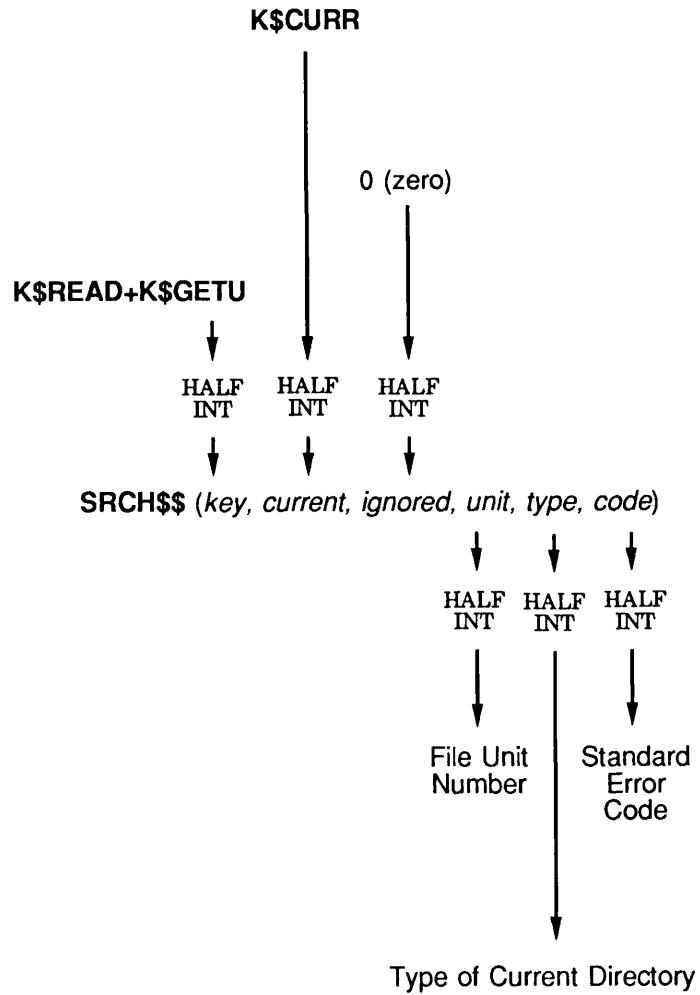
The SRCH\$\$ subroutine attempts to open the current directory and returns to your program

- An error code indicating whether the operation was successful
- A file unit number that identifies the open directory. This number is used when reading directory entries.
- The file type, indicating the type of file just opened (currently always top-level directory)

This section describes the input and output parameters that apply when calling SRCH\$\$, and then shows a sample call to SRCH\$\$.

Figure 6-9 illustrates the calling sequence of the SRCH\$\$ subroutine to open the current directory.

Open Current Directory



Q06.09.D10056.3LA

Figure 6-9. Calling Sequence of SRCH\$\$ to Open the Current Directory

The Error Code: An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was entirely successful. Otherwise, *code* is always positive. After a call to SRCH\$\$ to open the current directory, *code* may have one of many values. The *Advanced Programmer's Guide: Appendices and Master Index* contains a comprehensive list of all standard file system error codes. Error codes specific to this operation are:

<i>Error Code</i>	<i>Value</i>	<i>Meaning</i>
E\$NATT	7	No directory attached. This error usually occurs only when the directory to which the user is attached is removed from the system, as when a disk is shut down. Use one of the subroutines described in this chapter to reestablish a current attach point.
E\$NRIT	10	Insufficient access rights. You do not have List access to the current directory.
E\$MTPT		Directory is a mount point. You cannot reference a directory that is the mount point of another directory, so that tape backup procedures do not cross mount points. This error is returned by the <i>ref</i> key K\$NMNT.

Example: The following example shows how a FORTRAN program opens the current directory for reading:

```
CALL SRCH$$ (K$READ+K$GETU, K$CURR, 0, UNIT, TYPE, CODE)
  IF (CODE.NE.0) GO TO 1000
  .
  .
  .
1000 CALL ERRPR$ (K$IRTN, CODE, 'Current directory', 15, 'MYPROGRAM', 9)
      RETURN
```

Questions and Answers About Attach Points

This section answers some typical questions about attach points.

- If the current attach point gets reset by a mistyped command, or by the execution of a non-internal command, how does this affect my running program?

If a user quits (by typing CONTROL-P) while running a program, resets the current attach point by mistyping a command or executing a non-internal command, and then restarts the program (by using the START command), the

program may not continue working properly; its behavior when it resumes execution may be unpredictable, because it is suddenly performing file system operations in a different directory than intended. This is the case only if the user happened to quit while the program was using the current attach point separately from the home attach point.

Attach points are not a part of the recursive command environment. You must consider this when you write programs that tend to disassociate the home and current attach points while allowing the user to quit.

Even a call to SRSFX\$ or CP\$ can involve the specific use of the current attach point. (For example, calling CP\$ to invoke DELETE causes an attach to CMDNC0 to search for the DELETE program.) Because these subroutines are also interruptible, they may not continue execution properly if the current attach point is reset during an interruption, and improper execution of these subroutines may affect the operation of your program.

- Can I use the GPATH\$ subroutine to record my current attach point during a quit, and then when the user types START, call AT\$ with the pathname returned by GPATH\$ to preserve my attach point?

Yes, with the following caveat: the pathname returned by GPATH\$ contains no passwords. If your system is using password directories, it is possible that the mechanism proposed by this question might complicate matters, but if your system uses ACL directories throughout, then the proposed solution should work.

There are two points to consider:

- Make certain that the mechanism not only catches the original CONTROL-P, but also catches the subsequent START command. The recommended way to do this is to resignal the QUIT\$ condition from within the handler for QUIT\$. *Subroutines Reference II: File System* describes the condition signalling mechanism.
- The access to the specified directory is recalculated whenever the mechanism is engaged. This is rarely a problem, but it is possible that the attempt to reattach to the current directory could fail due to insufficient access, whereas the original attach to the current directory succeeded, if a change to the access control of that directory is made between the original attach and the subsequent attach.

Text Storage and Retrieval

7



Many applications must be able to store and retrieve text strings on disk. Under PRIMOS, text strings consist of 8-bit characters in ASCII format. Each text string is considered to be a **line** of text. A **text file** consists of one or more lines of text. This chapter describes how programs create and operate on text files.

Using PRIMOS, text storage and retrieval is straightforward. PRIMOS offers two methods of organizing lines of text on disk:

- Variable-length records
- Fixed-length records

Each method of organization has advantages and disadvantages, described later in this chapter. PRIMOS provides a unified interface to both types of files. In particular, the opening and closing of variable-length record and fixed-length record files is identical. The only difference is in the way data are actually read and written to the file.

This chapter describes

- The differences between variable-length record files and fixed-length record files
- How to open, extend, truncate, and close text files
- How to read and write variable-length files
- How to read, write, and position fixed-length record files
- The format of a variable-length record file
- The format of a fixed-length record file

This chapter closes with questions and answers about text files.

Subroutines for Accessing Files

The subroutines most often used when accessing text files are:

SRCH\$\$	Accepts a filename; opens, closes, deletes, changes access on, or verifies the existence of the file as requested by the key. Most commonly used to open and close files.
SRSFX\$	Allows the calling program to specify a list of legal file suffixes in order to find a file with one of them. Each supplied suffix is appended to the base pathname, until the file is found or the list of suffixes is exhausted. This subroutine is used by Prime software such as the RESUME command.
SGD\$OP	Opens a file within a segment directory. The segment directory must already be open.
RDLIN\$	Reads a line from an open variable-length record file, returning a fixed-length record buffer. The buffer is appropriately padded with spaces (240 octal).
WTLIN\$	Writes a fixed-length record to an open variable-length record file. The length of the line is calculated by subtracting the number of trailing spaces (240 octal) from the length of the record.
PRWF\$\$	Used to truncate a file after writing it, in case the file is to be made shorter. For fixed-length record files, PRWF\$\$ is also used to read, write, and position the file, as described later.

All of these subroutines are thoroughly described in *Subroutines Reference II: File System*.

Difference Between Variable-length and Fixed-length Record Files

The organization of data within a file is defined by the program or programs that use the file. PRIMOS does not maintain a description of the contents of any file. This allows flexibility in accessing files, because one program can treat a file as a collection of lines of text, while another program can treat the same file as binary data.

All programs that use a file must agree on the organization of data within the file, because PRIMOS does not impose restrictions on the access method. This means that all programs must know whether a text file consists of variable-length records or fixed-length records when operating on text files.

Therefore, you should decide early in any project whether to use variable-length records or fixed-length records. This prevents confusion and program misbehavior. If such a decision cannot be made early enough, you should build a small subroutine library that can perform fixed-length operations on variable-length record files, and vice versa. You rarely need such a subroutine

library, however, because the advantages and disadvantages of the two organizations are so distinct.

Variable-length Records

Text files under PRIMOS normally consist of variable-length records. Each line of text in a file is terminated by a new-line character, ASCII LF (212 octal). The lengths of lines in the file vary from 0 to an application-defined maximum. No prefix defining the record length is present; the new-line character delimits each record.

Variable-length records offer the following advantages:

- All utilities supplied by Prime that operate on text files accept variable-length records. Such utilities include SLIST, ED, EMACS, RUNOFF, SPOOL, and others. Only a few utilities, such as SORT, support fixed-length records.
- Using variable-length records usually saves disk space. This is true when the lengths of lines in a file vary, or when three or more contiguous spaces (a space is 240 octal) occur frequently in the file.
- There is no inherent limit to the length of a line in a variable-record file. Each particular application limits the operational length of lines in a text file to a specific quantity. This is also true of utilities supplied by Prime. The maximum line length in ED is different from that of EMACS, for example.
- The length of each line in a variable-record file is defined by a new-line character that follows it. Thus, a utility needs no information outside the file to use the file. On the other hand, any program that operates on a fixed-length record file must know the record size of the file.

Variable-length record files are sometimes referred to as **compressed** files. The term “compressed” refers to the compression of contiguous spaces. Another term, **uncompressed**, identifies a similar file format that does not include space compression. PRIMOS itself cannot distinguish between compressed and uncompressed files; your application must make this distinction.

Fixed-length Records

The alternate organization of text files under PRIMOS stores lines of text in fixed-length records. Here, the length of each record, or line of text, is known by the program using the file. Records are stored side-by-side in the file, with no intervening control information (such as a new-line character).

Fixed-length records offer the following advantages:

- Accessing a particular record is significantly faster when all of the records are fixed-length, since the location of the record is defined by only two variables — the record length for the file and the record number to be accessed. A variable-length record file must be searched sequentially until the desired record is found.
- There are no restrictions on the character set. Characters such as the ASCII line feed (212 octal), DC1 (221 octal), and null (000 octal) can be read and written without any special consideration.
- The execution speed of a program that expects records to contain fixed-length fields of information may be superior when fixed-length records are used. Fixed-length records can be read directly into PL/I structures or FORTRAN EQUIVALENCE areas without going through an intermediate parsing stage (as is necessary when reading variable-length records).
- Programs that use fixed-length record data can often be more easily moved from one large-scale computer system to another. Fixed-length record organization has been in use for approximately a century, beginning with the punched card. Variable-length record organization is comparatively recent, and is still the second choice in languages such as COBOL, FORTRAN, and PL/I.

Hybrid Approaches

As described previously, PRIMOS itself places no restrictions on the organization of data in a file. It is up to the programs that access the file to use the same access method on a file. Therefore, it is possible to construct hybrid file organizations that include advantages from both the fixed-length and variable-length record approaches.

For example, if you use fixed-length records separated by an ASCII LF (212 octal) and a NUL (000 octal) byte, you can display a fixed-length record file using SLIST. (It cannot be edited by using ED or EMACS, however.)

To solve the problem of having to hard-code the record length into programs that use fixed-length records, the first two bytes of a file can be defined to contain the length of records in the file in bytes. However, this prevents the file from being directly sorted by Prime's SORT facility — when sorting fixed-length record files, SORT does not expect the first two bytes to contain such information.

You may decide to have a variable-length record file to save disk space, using the PL/I language as a model. You can represent the individual records as PL/I CHARACTER(*) VARYING variables, rather than ending each record with a new-line code. This has two disadvantages. First, it renders such files inaccessible via PRIMOS utilities such as SLIST, ED, and EMACS. Second, records would presumably not be written using space compression techniques,

and therefore one might take up extra disk space. The important advantage of this approach is that of fixed-length records; the entire character set may be used within each record.

Although you can use approaches such as those described above instead of the variable-length and fixed-length record organizations, such approaches are not described in this book. If you find that you need a nonstandard organization for a text file, you must treat the file as a data file. The manipulation of data files is described in Chapter 8, Data Storage and Retrieval.

Maximum Length of a File

Currently, the maximum number of characters that can be stored in one file under PRIMOS is 465 million. This assumes a single file stored on a 30-head partition residing on a 675MB disk drive (also known as a 600MB disk), with the minimal housekeeping and directory information in the partition.

How to Open, Extend, Truncate, and Close Text Files

To read a text file, your program normally

1. Opens the file for reading.
2. Reads the file until the end of file is reached.
3. Closes the file.

To write a text file, your program normally

1. Opens the file for writing.
2. Positions the file to end-of-file if new data is to be written to the end of the file; otherwise, your program overwrites the existing data.
3. Writes the file, automatically extending the file length when necessary.
4. Truncates the file at the current position to insure that old data originally in the (longer) file is deleted.
5. Closes the file.

PRIMOS does not impose restrictions on the order of these operations except that your program must open a file before it can read, write, extend, or truncate the opened file.

The subjects of this section are how to open, position to the end of, truncate, and close a text file. The subsequent two sections describe how to actually read and write text files.

Opening a File

Before your program can access data in a file, it must open the file. Your program opens a file by using the SRSFX\$, SGD\$OP, or SRCH\$\$ subroutines. When your program calls these subroutines, it provides

- The name of the file to be opened.
- A key that specifies how the file is to be opened.

The SRSFX\$, SGD\$OP, or SRCH\$\$ subroutine attempts to open the specified file and returns the following information to your program:

- An error code indicating whether the operation was successful.
- A file unit number that identifies the open file. Your program uses this number when performing operations (such as read and write) on an open file.
- The file type, indicating the type of file just opened (including SAM, DAM, CAM, SEGSAM, SEGDAM, and Directory).

This section contains the input and output parameters applicable when you call SRSFX\$, SGD\$OP, and SRCH\$\$, and shows a sample call to SRCH\$\$\$. Figure 7-1 illustrates the calling sequence of SRSFX\$ to open a file; Figure 7-2 illustrates the calling sequence of SGD\$OP; Figure 7-3 illustrates the calling sequence of SRCH\$\$ to open a file.

The Name of the File: The rules for the filename depend on the system subroutine being called.

Your program may supply a pathname if it is using SRSFX\$. The pathname may identify segment directory members (such as FRED>XYZ.SEG>0).

For SGD\$OP, your program must first position the segment directory to the desired entry by using SGDR\$\$; then, your program calls SGD\$OP, providing the file unit number of the open segment directory.

When calling SRCH\$\$, the filename must be an objectname; that is, it cannot contain a > symbol. SRCH\$\$ searches the current directory for the specified file system object.

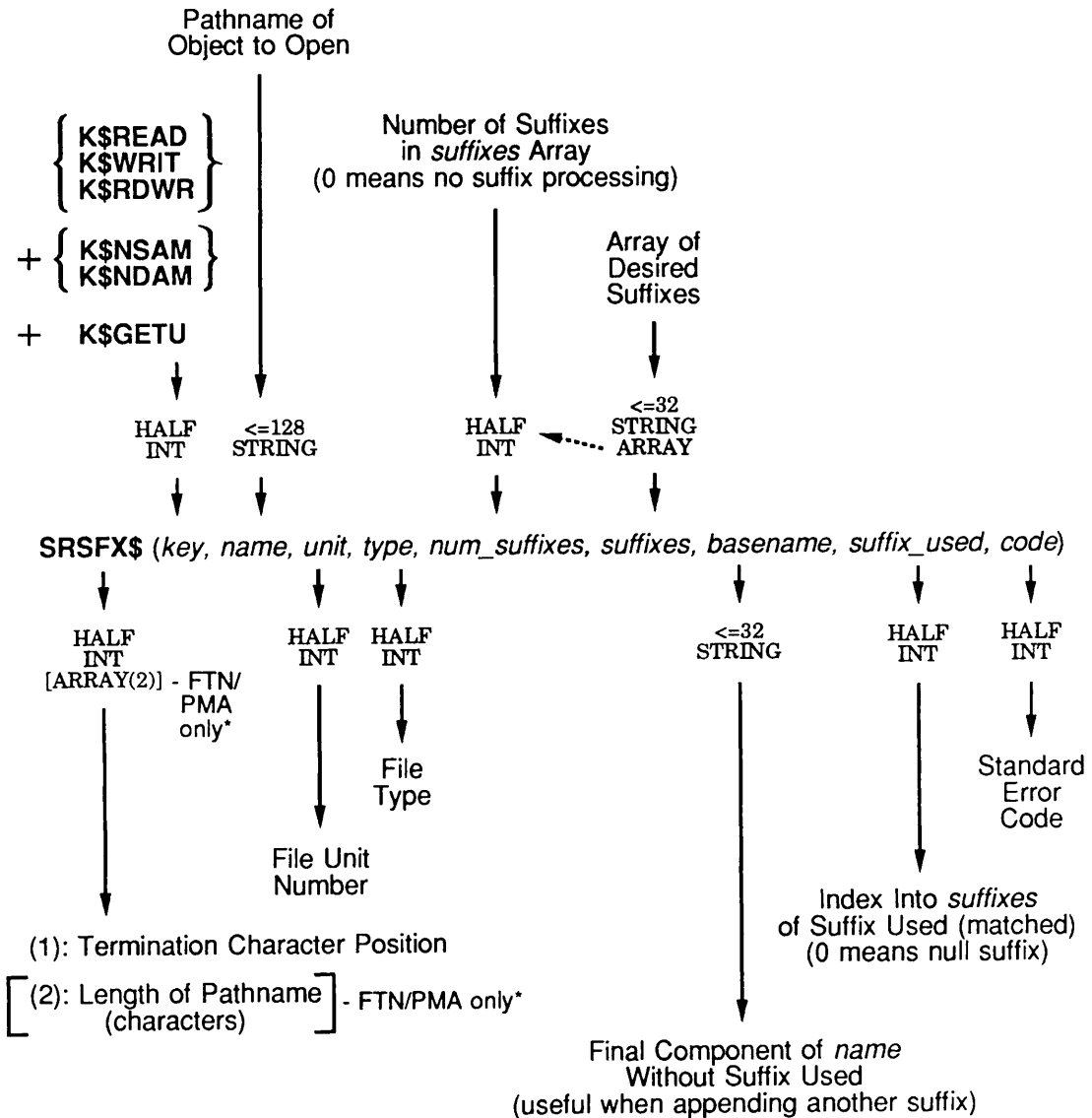
The Key: In the case of SRSFX\$ and SRCH\$\$, your program sets *key* as follows:

$$key = action + newfile + K$GETU$$

For SGD\$OP, your program sets *key* as follows:

$$key = action$$

Open a File, With Possible Suffix



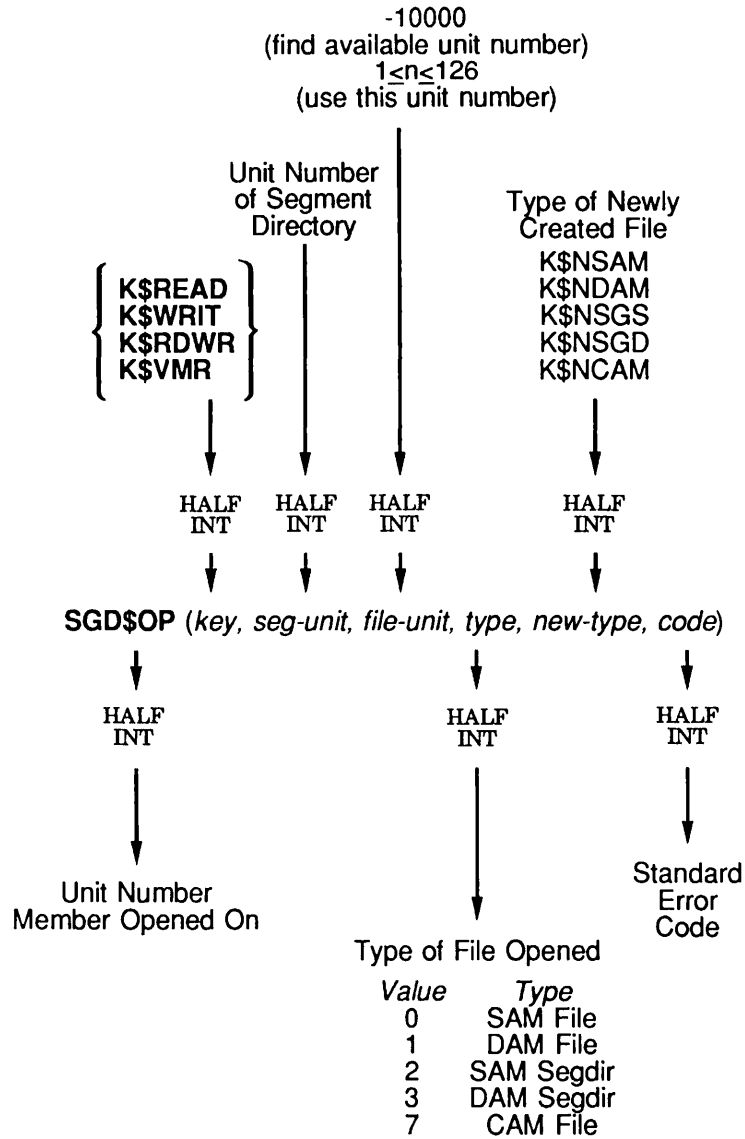
Side Effects: May reset current attach point.

* Function value is returned in L-register; typically, you need only to declare as HALF INT, because first datum is all you need and is in A-register. Otherwise, you must declare it as FULL INT to make it work.

Q07.01.D10056.3LA

Figure 7-1. Calling Sequence of SRSFX\$ to Open a File

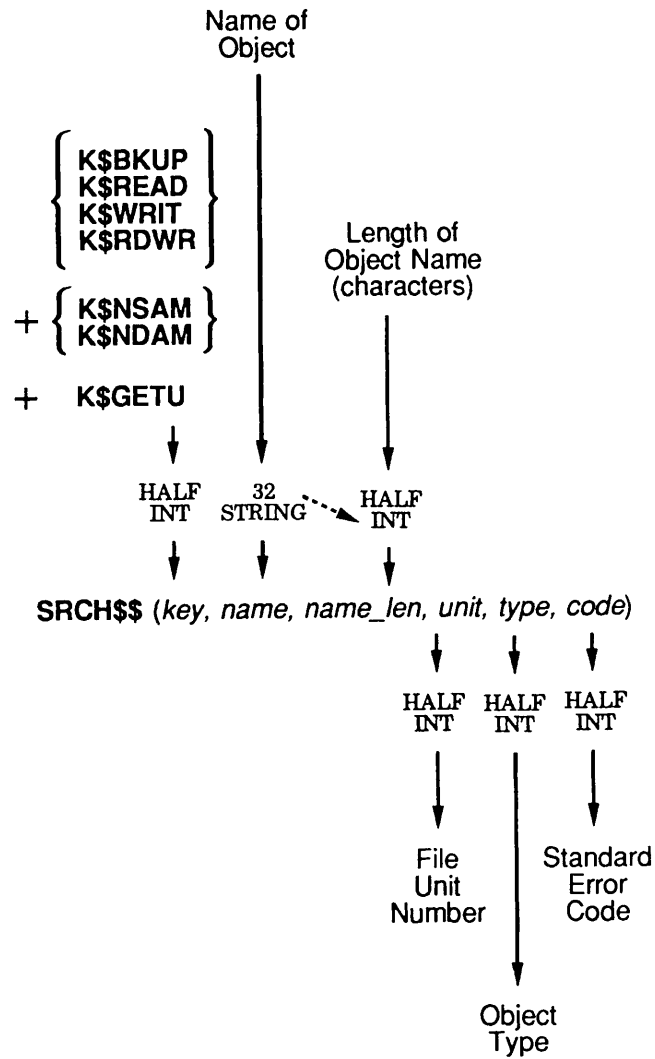
Open Member of Segment Directory



Side Effects: If *seg-unit* is at end of segment directory and *key* is K\$WRIT or K\$RDWR, SGD\$OP attempts to automatically extend segment directory by one entry, which also repositions *seg-unit* to new *end-of-segdir* position; otherwise, size of segment directory and position of *seg-unit* remain unchanged.

Figure 7-2. Calling Sequence of SGD\$OP

Open File in Current Directory



Q07.03D10056.3LA

Figure 7-3. Calling Sequence of SRCH\$\$ to Open a File

The values and meanings of *action* and *newfile* are:

<i>Value</i>	<i>Meaning</i>
<i>action</i>	Specifies how the file is to be opened. This distinguishes between a file being open for reading, writing, or both reading and writing. These states are often identified by the mnemonics R, W, and RW (or WR), respectively. The keywords used when opening files are

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
K\$READ	1	Open the file for reading.
K\$WRIT	2	Open the file for writing.
K\$RDWR	3	Open the file for both reading and writing.
K\$BKUP	7	Open for reading by backup facility.
K\$VMR	16	Open for VMFA read.

If your program attempts to write to a file that is open for reading, an error code of E\$UNOP (Unit not open) is returned to your program. This same error code is returned if your program attempts to read a file that is open for writing.

<i>newfile</i>	Specifies what type of file should be created if the file does not already exist. (The file is created only if your program is opening the file for writing or for reading and writing.)The keywords used for text files are
----------------	--

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
K\$NSAM	0	Create a new threaded (SAM) file. (This is the default.)
K\$NDAM	1024	Create a new directed (DAM) file.

For SGD\$OP, the *newfile* value is replaced by the *new-type* argument in the calling sequence. This argument may also include:

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
K\$NSGS	2048	Create a new SAM Segment Directory.
K\$NSGD	3072	Create a new DAM Segment Directory.
K\$NCAM	4096	Create a new contiguous (CAM) file.

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
K\$GETU		Specifies that PRIMOS is to use an available file unit, and return the selected file unit number in the unit parameter of the calling sequence. For SGD\$OP, your program specifies that PRIMOS is to use an available file unit by supplying a unit number of -10000. If you want your program to specify the unit number instead of letting PRIMOS select the number, your program supplies a unit number between 1 and 126 (or 1 and 15 for a program running under PRIMOS II).

The Error Code: An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was entirely successful. Otherwise, *code* is always positive. After a call to SRSFX\$, SGD\$OP, or SRCH\$\$ to open a file, *code* may have one of many values. The *Advanced Programmer's Guide: Appendices and Master Index* contains a comprehensive list of all standard file system error codes. Error codes specific to this operation are:

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$FIUS	5	File in use. The file being opened is already open on another file unit, or by another user. Normally, a file that is open for reading cannot be opened for writing, nor can a file open for writing be opened for reading. A file that is open for writing can have only one file unit open to it, whereas a file open for reading can have many file units open to it. If you expect your program to open a file that may occasionally be in use by another process for a short period of time, consider having your program repeatedly attempt to open an in-use file for 30 seconds or a minute, sleeping one second in between each attempt by calling SLEEPS. See Chapter 10, File Attributes, for more information on the read/write lock.
E\$DKFL	9	The disk is full. This error can occur only if a new file is being created, and hence cannot occur if the action portion of the <i>key</i> argument is K\$READ.
E\$NRIT	10	Insufficient access rights. If the file being opened already exists, this means that the user running your program does not have sufficient access rights to the file. If the file does not exist, then the user does not have Add rights to the directory in which the file is to be created.

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
		For calls to SRSFX\$, this error code may indicate a problem attaching to the directory that was specified by the pathname argument of the calling sequence. In this case, the user does not have Use access to at least one directory in the pathname.
E\$FNTE	15	Not found. The file being opened does not exist. The action portion of the <i>key</i> argument is probably K\$READ; otherwise, the file would be created.
		For calls to SRSFX\$, this error code may indicate a problem attaching to the directory that was specified by the pathname argument of the calling sequence. In this case, at least one directory in the pathname does not exist. Even if the action portion of <i>key</i> is K\$WRIT or K\$RDWR, no directory is ever created via a call to SRSFX\$. You must use the DIR\$CR subroutine to create a directory.
E\$ITRE	57	Illegal treename. (SRSFX\$ only.) This indicates that the pathname supplied to SRSFX\$ does not conform to the syntax rules for a pathname. See the <i>PRIMOS User's Guide</i> for a description of the syntax of a pathname.
E\$MXQB	143	Maximum quota exceeded. This error can occur only if a new file is being created, and hence cannot occur if the action portion of the <i>key</i> argument is K\$READ.
E\$IACL	150	Entry is an access category. The specified file system object is an access category. See Chapter 9, Access Control Lists (ACLs), for information on access categories.
E\$NINF	159	No information. This indicates that some error occurred, but the user running your program does not have List access to the directory involving the error. In such a case, the E\$NINF error code is always returned to prevent the user or calling program from getting any information about the directory. Therefore, this error code indicates any possible error, in addition to a simple case of insufficient access.

The File Type: The returned file type is valid when the returned error code is 0. It is not valid in any other case.

The file type is one of the following five values:

<i>Value</i>	<i>Meaning</i>
0	A SAM file has been opened. Use RDLIN\$, WTLIN\$, PRWF\$\$, and similar subroutines to read or write it.
1	A DAM file has been opened. Use RDLIN\$, WTLIN\$, PRWF\$\$, and similar subroutines to read or write it.

- 2 A SAM segment directory (SEGSAM) has been opened. Use SGDR\$\$ to operate on members of this segment directory. See Chapter 8, Data Storage and Retrieval, for information on how to do this.
- 3 A DAM segment directory (SEGDAM) has been opened. Use SGDR\$\$ to operate on members of this segment directory. See Chapter 8, Data Storage and Retrieval, for information on how to do this.
- 4 A top-level directory has been opened. Use DIR\$SE, DIR\$RD, ENT\$RD, and RDEN\$\$ to read information on files in this directory. See Chapter 8, Data Storage and Retrieval, for information on how to do this.
- 7 A CAM file has been opened. Use RDLIN\$, WTLIN\$, PRWF\$\$, and similar subroutines to read or write it.

Examples: The following example shows how a FORTRAN program would open the file MYFILE in the current directory for reading:

```

      CALL
SRCH$$ (K$READ+K$GETU, 'MYFILE', 6, UNIT, TYPE, CODE)
      IF (CODE.NE.0) GO TO 1000
      .
      .
      .
1000 CALL ERRPR$ (K$IRTN, CODE, 'MYFILE', 6, 'MYPROGRAM', 9)
      RETURN

```

The next example illustrates the use of the *newfile* value in the *key* argument of the calling sequence to SRCH\$\$. The file ANOTHER_FILE is opened for reading and writing in the current directory. If it does not exist, it is created as a DAM (directed) type file. Only the subroutine call itself is shown; the error code would be examined in the same fashion as shown in the example above.

```

      CALL
SRCH$$ (K$RDWR+K$NDAM+K$GETU, 'ANOTHER_FILE', 12, UNIT,
&     TYPE, CODE)

```

Positioning a File to End-of-file

When your program is writing data to a text file, you may want it to add new data to the end of the existing file and leave the previously entered data intact. To position a newly opened file to the end-of-file location, have your program call a subroutine named POSIT, shown below, with the HALF INT file unit number of the file, and a HALF INT returned error code. Your program then

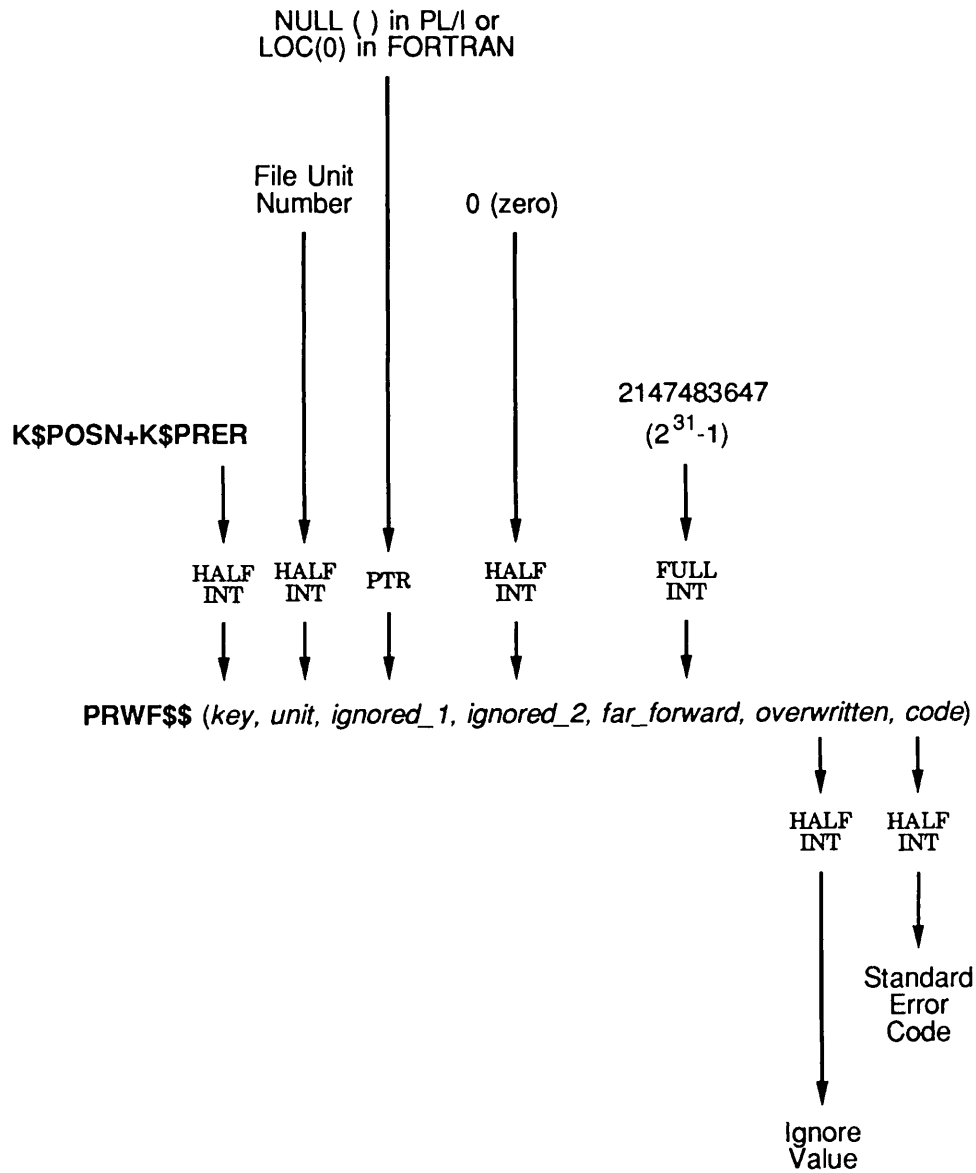
checks the returned error code to make certain the operation succeeded. If it did not, your program closes the opened file, produces an error message, and aborts.

The following FORTRAN statements illustrate the procedure.

```
CALL SRCH$$ (K$WRIT+K$GETU, 'MYFILE', 6, UNIT, TYPE, CODE)
IF (CODE.NE.0) GO TO 1000
C
CALL POSIT(UNIT, CODE) /* Position to end of file.
IF (CODE.NE.0) GO TO 1001
C
1000 CALL ERRPR$(K$IRTN, CODE, 'MYFILE', 6, 'MYPROGRAM', 9)
RETURN
C
1001 CALL SRCH$$ (K$CLOS, 0, 0, UNIT, TYPE, I) /* Don't overwrite CODE!
GO TO 1000
```

The POSIT subroutine mainly uses a form of the PRWF\$\$ subroutine that positions a file. Figure 7-4 illustrates the calling sequence of the PRWF\$\$ subroutine to position toward end-of-file.

Position Toward End-of-file



Q07.04.D10056.3LA

Figure 7-4. Calling Sequence of PRWF\$\$ to Position Toward End-of-file

The POSIT subroutine might be written as follows:

```

SUBROUTINE POSIT (UNIT, CODE)
  INTEGER*2 UNIT, CODE
C
C This subroutine positions the specified file unit to the
C end-of-file location. It returns the success or failure
C of the operation in the CODE parameter.
C
$INSERT SYSCOM>ERRD.INS.FTN
$INSERT SYSCOM>KEYS.INS.FTN
C
      INTEGER*2 RNW
C
10  CALL PRWF$$ (K$POSN+K$PRER, /* Position relative.
& UNIT, /* Pass the file unit number.
& LOC(0), /* This pointer is unused during a
& /* position-only operation.
& 0, /* Another unused value in this case.
& 2147483647, /* Largest positive INTEGER*4 number.
& RNW, /* Unused, but may be overwritten anyway.
& CODE) /* The error code.
C
C CODE should never be 0. If it is, it means the file is
C very large. Loop until we reach the end of the file.
C
      IF (CODE.EQ.0) GO TO 10 /* File is very big!
C
C However, if the returned error code is E$EOF (End of file),
C then we succeeded, so set it to 0. In any case, return.
C
      IF (CODE.EQ.E$EOF) CODE=0 /* Success.
RETURN
END

```

Truncating a File

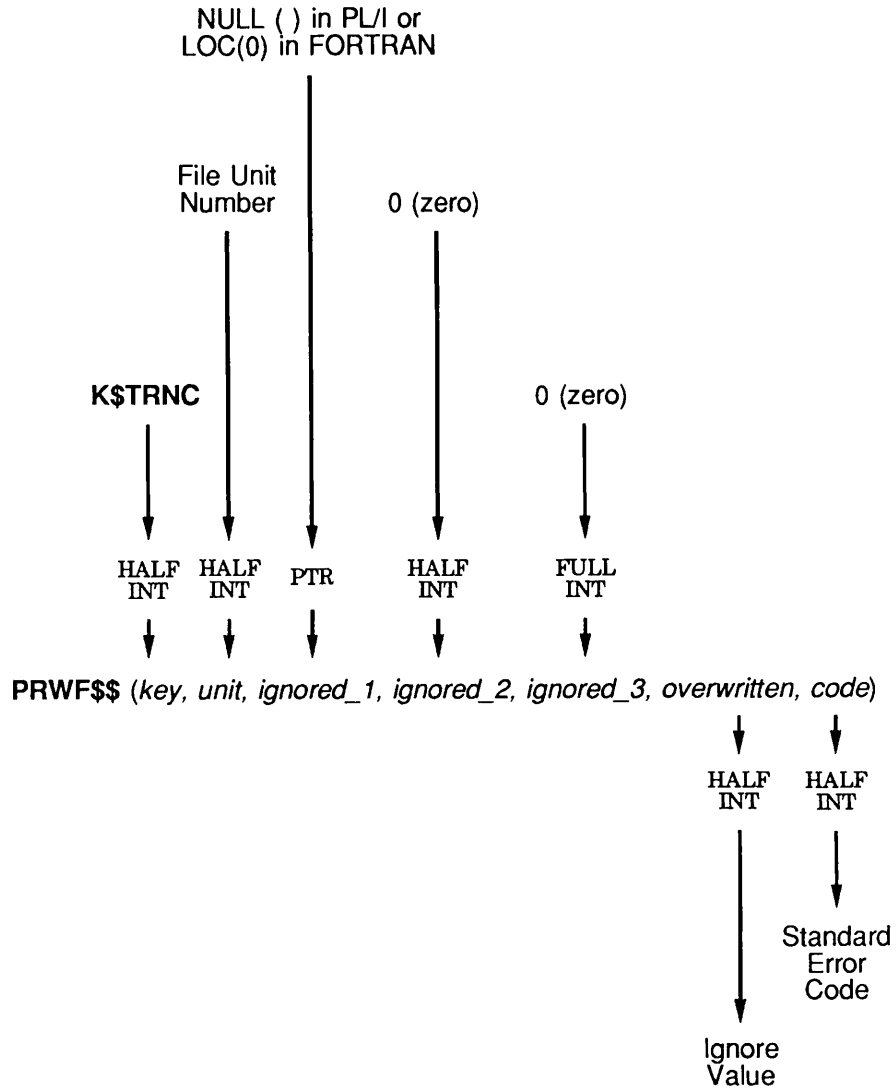
Before closing a file you have written to, it is good practice to have your program truncate the file. This tells PRIMOS to make the current position of the file the new end-of-file location.

If your program just created the file, this operation is not necessary. However, if your program has opened an existing file and overwritten it, your program may not have written up to the current end-of-file location of the file. If this is the case, and your program performs no truncation, the file ends up having more data than was intended, and the chances are good that one record of the old file data has been partially overwritten by the new data.

The truncation operation is simple, and is always done by PRWF\$\$\$. Figure 7-5 illustrates the calling sequence of the PRWF\$\$\$ subroutine to truncate a file. A sample use of PRWF\$\$\$ follows:

```
CALL PRWF$$$ (K$TRNC, /* Truncate the file.  
& UNIT, /* The file unit number.  
& LOC(0), /* Ignored when truncating.  
& 0, /* Ignored when truncating.  
& 000000, /* Truncate at the current position.  
& RNW, /* Ignored when truncating, but play it safe.  
& CODE) /* The error code.  
C  
IF (CODE.NE.0) CALL ERRPR$(K$IRTN, CODE, /* Not fatal.  
& 'Cannot truncate file', 20, 'MYPROGRAM', 9)
```

Truncate File at Current Position



Q07.05.D100563LA

Figure 7-5. Calling Sequence of PRWF\$\$ to Truncate a File

As shown in the example above, most programs do not regard an inability to truncate a text file as an error, although they do produce an error message. For example, the PRIMOS editor ED treats an inability to truncate a file as a nonfatal error. This is because the data have been written, but there exists the possibility of extraneous data in the file.

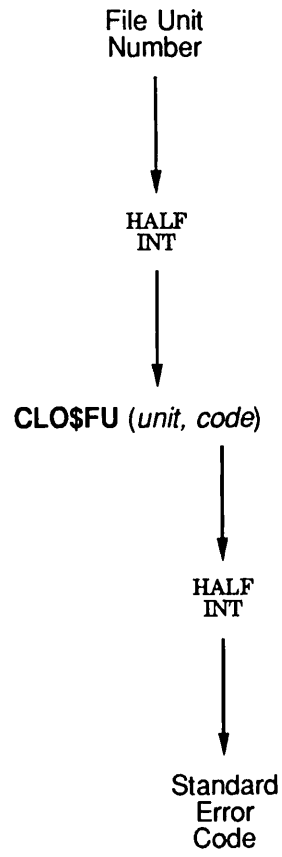
The *Advanced Programmer's Guide: Appendices and Master Index* contains a comprehensive list of all standard file system error codes. Error codes that may typically be returned as a result of attempting to truncate a file follow.

<i>Value</i>	<i>Keyword</i>	<i>Meaning</i>
E\$EOF	1	End of file. This can occur only if the call to PRWF\$\$ inadvertently specifies that the position of the file be changed. The fifth argument in the call to PRWF\$\$ should always be an INTEGER*4 zero (000000 in FORTRAN, 0L in PMA). If it is not, this error code may be returned.
E\$BOF	2	Beginning of file. This can occur only if the call to PRWF\$\$ inadvertently specifies that the position of the file be changed. The fifth argument in the call to PRWF\$\$ should always be an INTEGER*4 zero (000000 in FORTRAN, 0L in PMA). If it is not, this error code may be returned.
E\$UNOP	3	Unit not open. The specified file unit is not open, or is open only for reading. This usually indicates a program error, although it can also be the result of the user exiting the program via CONTROL-P, typing CLOSE ALL, and then typing START.
E\$FIUS	5	File in use. The file being truncated is already open on another file unit, or being used by another user. This error code usually indicates that the open file has a read/write lock setting of UPDT or NONE, because the program has the file open for writing, and yet at least one other file unit is open to the file for reading or writing. Chapter 10, File Attributes, contains information on the read/write lock.

Closing a File

It is very easy to close a file. The best method is to close the file by unit number. This means that only the file unit specified in the call to CLO\$FU is closed. Figure 7-6 illustrates the calling sequence of the CLO\$FU subroutine.

Close a File Unit Number



Q07.D6.D10056.3LA

Figure 7-6. Calling Sequence of CLO\$FU

A sample use of CLO\$FU is

```
CALL CLO$FU (UNIT, CODE)
IF (CODE.NE.0) CALL ERRPR$(K$IRTN, CODE, 'Cannot close', 12,
& 'MYPROGRAM', 9)
```

If a nonzero error code is returned, your program should treat it as a fatal error, because subsequent operations on the file by the same program, other programs, or the user may fail. In addition, a failure to close may indicate a program error or a disk error, both of which suggest that the program should not attempt to continue processing.

Your program may also close a file by name by using the CLO\$FN subroutine by passing the same pathname that was used to open the file. Figure 7-7 illustrates the calling sequence of the CLO\$FN subroutine.

When a file is closed by name, all file units opened to the file by the user are closed. This function is rarely needed, because most programs open a file on only one file unit at a time. However, interactive users often find this ability useful, such as when they try to write out an edited command input file, and receive a File in use message from the editor. At that point, a CLOSE *filename* command fixes the problem, no matter what the file unit number for *filename* is, as long as no other users have the file open.

Here is a sample use of closing a file by name:

```
call clo$fn('MYDIR>MYFILE', code);
if code ^= 0 then call errpr$(k$irtn, code, 'Cannot close', 12,
' MYPROGRAM', 9);
```


How to Read and Write Variable-length Text Files

Use the RDLIN\$ subroutine to read a variable-length record file (compressed or uncompressed), and use the WTLIN\$ subroutine to write a variable-length record file (compressed).

Note In most cases, variable-length record files are compressed. In other words, contiguous spaces in a line are compressed into a two-byte code to save disk space. Therefore, this section describes only the reading and writing of compressed files. For information on the format of compressed variable-length record files, see the section entitled Format of a Variable-length Record File, later in this chapter.

This section describes

- The RDLIN\$ and WTLIN\$ interfaces
- Sample uses of RDLIN\$ and WTLIN\$

The RDLIN\$ and WTLIN\$ Interfaces

The subroutine interfaces for RDLIN\$ and WTLIN\$ are identical. Each subroutine has the following arguments:

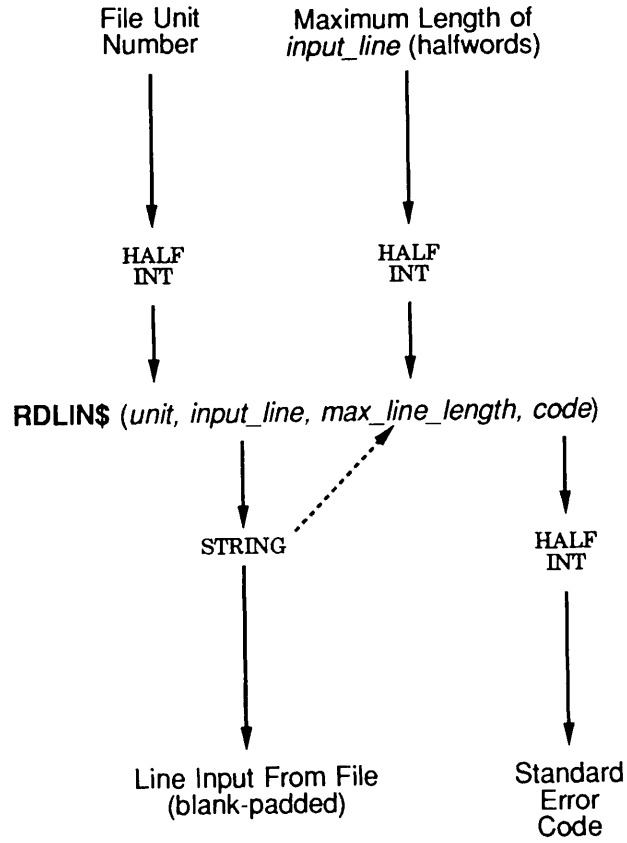
- A file unit
- A character buffer
- The length of the buffer (in halfwords)
- An error code

For RDLIN\$, the input arguments are the file unit and the length of the buffer, and the output arguments are the character buffer and the error code. For WTLIN\$, the input arguments are the file unit, the character buffer, and the length of the buffer, and the only output argument is the error code. Figure 7-8 illustrates the calling sequence of the RDLIN\$ subroutine; Figure 7-9 illustrates the calling sequence of the WTLIN\$ subroutine.

File Unit: Each subroutine takes the file unit number as an input argument. The file unit must be open for reading (RDLIN\$), writing (WTLIN\$), or reading and writing (either subroutine). Further, the file that is open on the file unit must be a SAM or DAM file; it cannot be a segment directory or file directory.

The reading or writing of the line (or record) begins at the current position in the file for that file unit. After the line is successfully read or written, the current position of the file unit immediately follows the line. Therefore, a subsequent call to RDLIN\$ or WTLIN\$ reads or writes the next line. Your program does

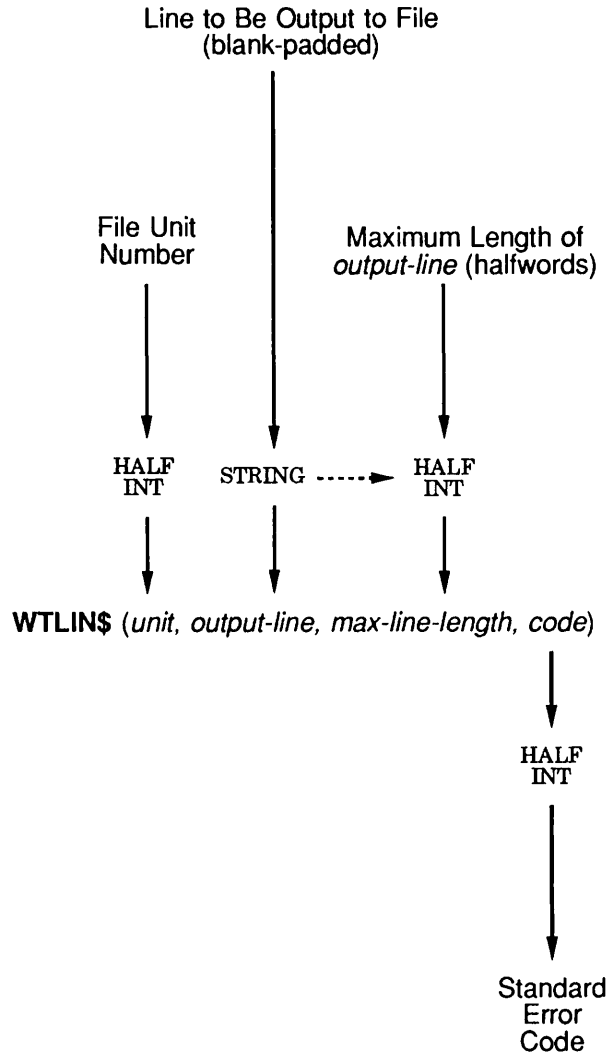
Read a Line of Text



Q07.08.D10056.3LA

Figure 7-8. Calling Sequence for RDLIN\$

Write a Line of Text



Q07.D9.D10056.3LA

Figure 7-9. Calling Sequence for WTLIN\$

The Error Code: An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was entirely successful. Otherwise, *code* is always positive. After a call to RDLIN\$ or WTLIN\$, *code* may have one of many values. The *Advanced Programmer's Guide: Appendices and Master Index* contains a comprehensive list of all standard file system error codes. Error codes specific to this operation are:

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$EOF	1	End of file (RDLIN\$ only). The end of the file was reached. The contents of the character buffer are undefined. Normally, this means that there is no more data in the file, but it could mean that there is an incomplete line at the end of the file, that is, data without a following new-line character (ASCII 212).
E\$UNOP	2	Unit not open. When calling RDLIN\$, this means the file unit is open only for writing, or is not open at all. When calling WTLIN\$, this means the file unit is open only for reading, or is not open.
E\$BPAR	6	Bad parameter. The length of the buffer as passed by the calling program is a negative number.
E\$DKFL	9	The disk is full (WTLIN\$ only). The line could not be completely written to the file because the disk was full. The amount of data successfully written to the file is undefined, so the only way to recover from this error is to call PRWF\$\$ to read the file position before calling WTLIN\$, and reposition the file after the E\$DKFL error occurs before trying to write the line again or truncating the file.
E\$MXQB	143	Maximum quota exceeded (WTLIN\$ only). The line could not be completely written to the file because the quota for the directory was exceeded. The amount of data successfully written to the file is undefined. The only way to recover from this error is to call PRWF\$\$ to read the file position before calling WTLIN\$. Then, before trying to write the line again or truncating the file, reposition the file after the E\$MXQB error occurs.

Sample Programs Using RDLIN\$ and WTLIN\$

Here is a sample FORTRAN subroutine that uses WTLIN\$ to write lines to an open file unit. If a disk-full or quota-exceeded error occurs, the user is given an opportunity to delete files and restart the program, and the subroutine retries the write in the correct fashion. This subroutine has the same calling sequence as WTLIN\$.

```

SUBROUTINE WRITE(UNIT,BUFFER,BUFLEN,CODE)
      INTEGER*2 UNIT,BUFFER(1),BUFLEN,CODE
C
C BUFFER can be dimensioned to just 1, even though it is probably
C larger, because this subroutine does not reference its contents;
C it simply passes the buffer on to WTLIN$.
C
$INSERT SYSCOM>ERRD.INS.FTN
$INSERT SYSCOM>KEYS.INS.FTN
C
      INTEGER*2 RNW,PATHNM(40),PATHLN,CODE2
      INTEGER*4 POSITN /* A fullword variable.

C
C First use PRWF$$ to determine our current position in the file.
C
      CALL PRWF$$ (K$RPOS, /* Special "read-position" function.
& UNIT, /* The file unit.
& LOC(0), /* Unused during a read-position.
& 0, /* Also unused during a read-position.
& POSITN, /* Report position in POSITN.
& RNW, /* Unused, but always play it safe.
& CODE) /* The error code.
      IF (CODE.NE.0) RETURN /* Failure.

C
C Now that we know the position of the file before the attempt to
C write the line, we attempt to write the line.
C
1 CALL WTLIN$(UNIT,BUFFER,BUFLEN,CODE) /* Simple enough.
C
C Examine the return code. If disk-full or quota-exceeded, do
C special processing. Otherwise, return.
C
      IF (CODE.NE.E$DKFL.AND.CODE.NE.E$MXQB) RETURN

C
C A disk-full or maximum-quota error has occurred. We want to tell
C the user to clean up the directory and type START. First, output
C the error message, along with the full pathname of the file being
C written (so the user knows where to delete files!).
C
      CALL GPATH$(K$UNIT,UNIT,PATHNM,80,PATHLN,CODE2) /* Get the
& /* full pathname of the file open on the file unit.
      IF (CODE2.EQ.0) GO TO 10 /* Error?

C
C If we can't get the treename, ignore the error, but make the
C error message useful.
C
      CALL ERRPR$(K$IRTN,CODE,'Unknown file name',17,

```

```

& 'WRITE',5)
GO TO 20
C
C Otherwise, produce an error message showing the treename.
C
10 CALL ERRPR$(K$IRTN, CODE, PATHNM, PATHLN, 'WRITE', 5)
C
C Now explain to the user what must be done.
C
20 CALL TNOU(0,0) /* Blank line.
CALL TNOUA('Free up some space using DELETE, then type ',
& 43)
CALL TNOU('START to continue.',19)
CALL TNOU(0,0)
C
C Now invoke a new command level, and hope we return.
C
CALL COMLV$
C
C User has typed START, repositioned, and retried the write.
C
CALL PRWF$$ (K$POSN+K$PREA, /* Position absolute.
& UNIT, /* Hopefully this is still open!
& LOC(0), /* Unused during a position.
& 0, /* Also unused.
& POSITN, /* Specify the desired position.
& RNW, /* Unused, but play it safe.
& CODE) /* The error code.
C
C If it works, retry, else return to the caller.
C
IF (CODE.EQ.0) GO TO 1
RETURN
C
END

```

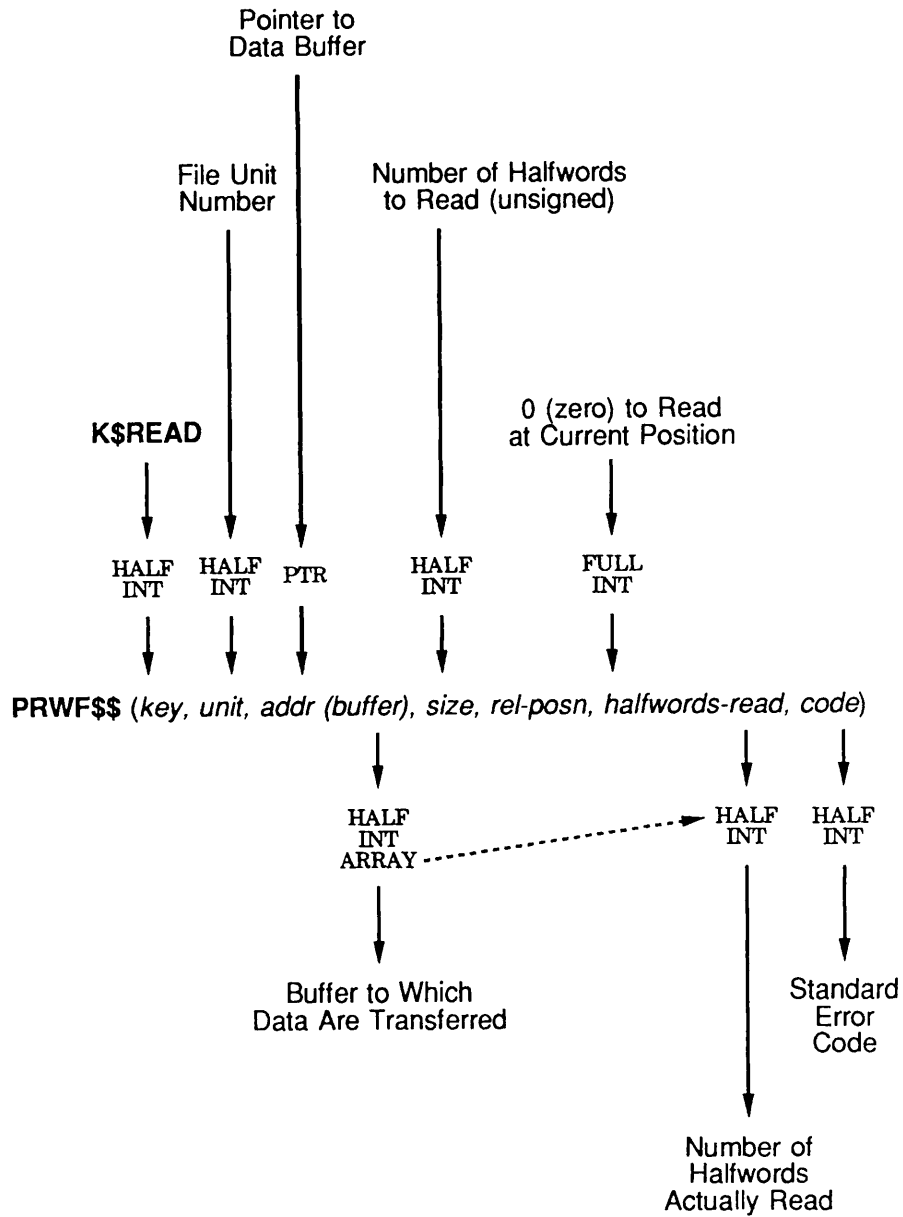
The following is a sample use of RDLIN\$. This PL/I subroutine serves as an interlude for RDLIN\$. It returns a CHARACTER(80) VARYING string containing the input line.

```

read: proc(unit,line,code); /* Similar to RDLIN$, but no buffer
                             length, and LINE is a varying
                             character string. */
dcl unit fixed bin(15), /* The file unit (input). */
line char(80) var, /* The line read (output). */
code fixed bin(15), /* The error code (output). */
buff char(80); /* This is what is passed to RDLIN$. */

```


Read a File

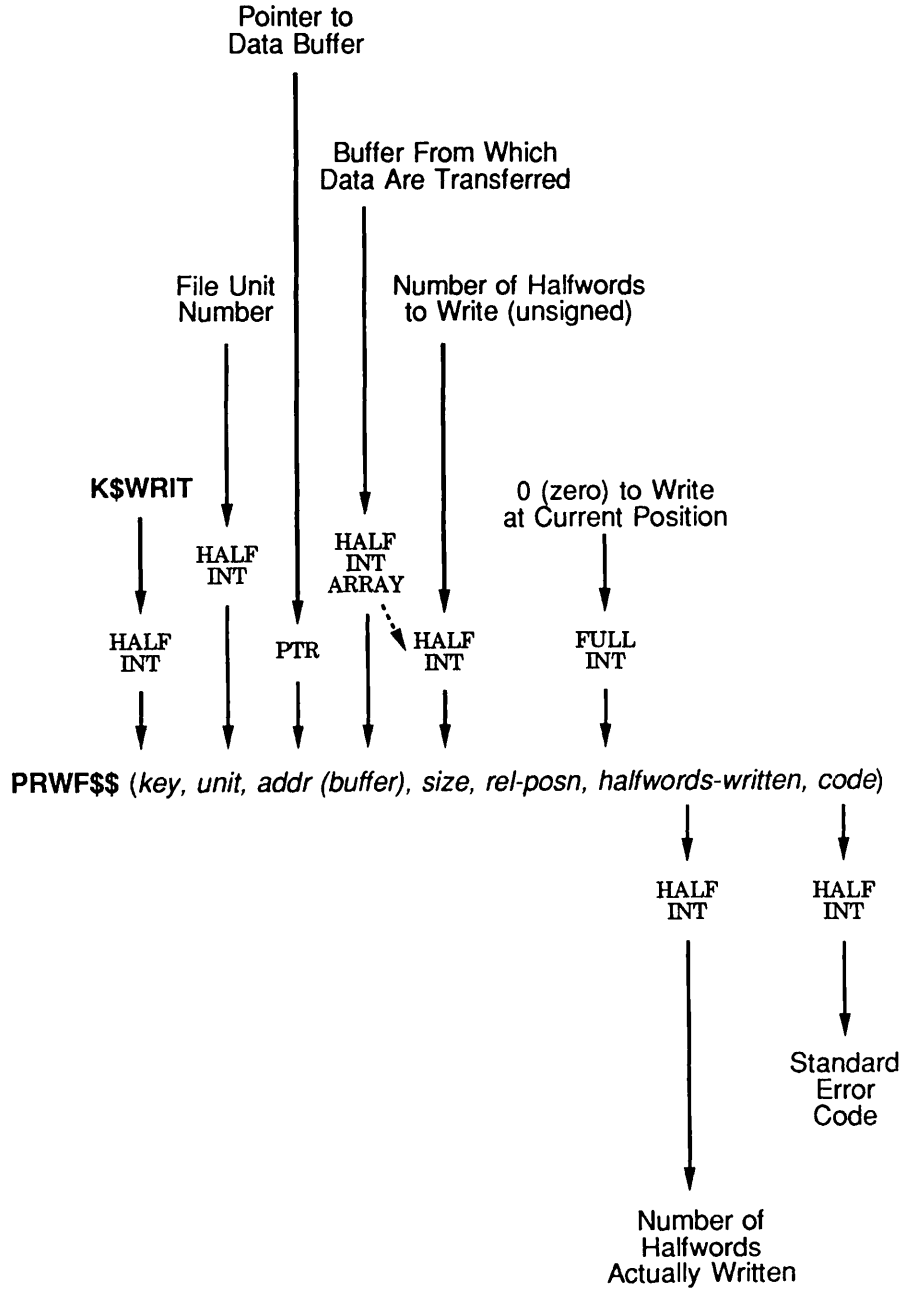


Side Effects: Contents of *buffer* elements *halfwords-read* + 1 through *size* are undefined after the operation if fewer halfwords than requested were read.

Q07.10.D10056.3LA

Figure 7-10. Calling Sequence of PRWF\$\$ to Read a File

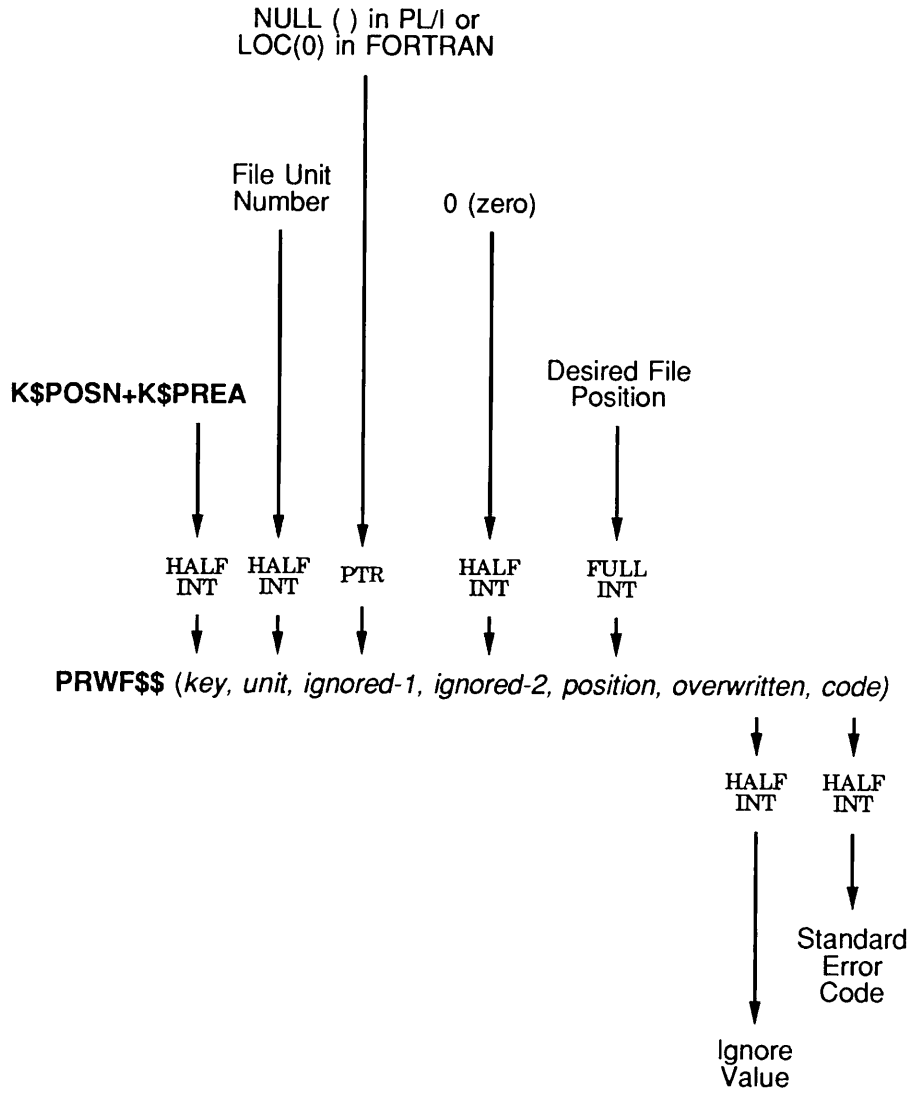
Write a File



Q07.11.D10056.3LA

Figure 7-11. Calling Sequence of PRWF\$\$ to Write a File

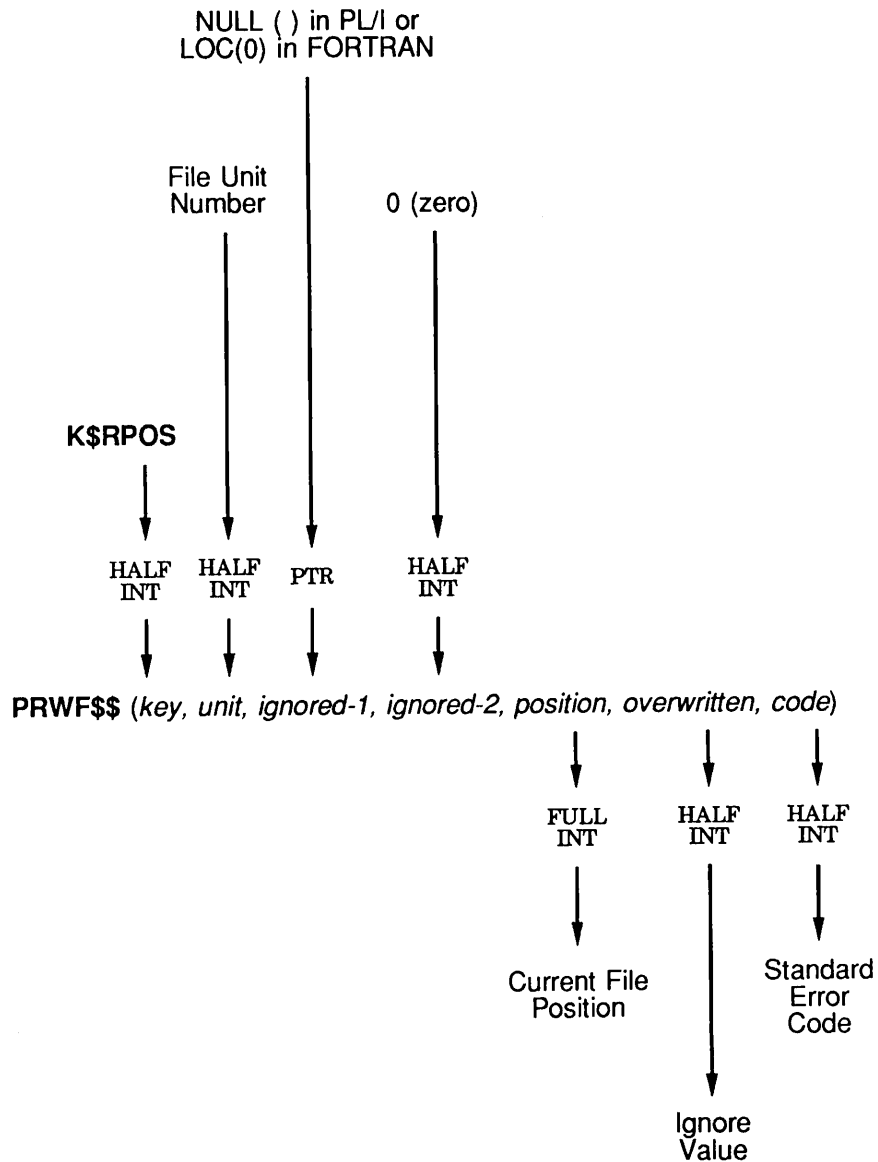
Position a File



Q07.12.D10056.3LA

Figure 7-12. Calling Sequence of PRWF\$\$ to Position a File

Read the Position of a File



Q07.13.D10056.3LA

Figure 7-13. Calling Sequence of PRWF\$\$ to Read the Position of a File

Key: The key argument tells PRWF\$\$ what operation is to be performed:

<i>Key</i>	<i>Meaning</i>
K\$READ	Read data from the file starting at the current position.
K\$WRIT	Write data to the file starting at the current position.
K\$POSN+K\$PREA	Position the file to the specified location.
K\$TRNC	Truncate the file at the current position.
K\$RPOS	Return the current position of the file.

The above functions represent all of the functions you need to make full use of PRWF\$\$ for text files. There are more key values you can pass, but aside from two special functions, these involve performing two or more of the above operations at one time. Generally, you should avoid the use of such combined operations, because ambiguity can result if you perform combined operations and a nonzero error code is returned. For example, if you attempt to pre-position the file and read data at the same time, it is unclear what has actually happened if the returned error code is E\$EOF.

File Unit: The file unit must be open for reading, writing, or both reading and writing. Further, the file that is open must be a SAM or DAM file. It cannot be a segment directory or file directory.

The data are read or written beginning at the current position in the file for that file unit. After the data are successfully read or written, the current position of the file unit is changed to immediately follow the data. Therefore, a subsequent call to PRWF\$\$ reads or writes subsequent data; you do not need to call PRWF\$\$ to position the file when reading or writing contiguous data in file.

Pointer to a Buffer: You supply a pointer to the buffer as an input argument to PRWF\$\$, whereas the buffer itself is used as input or output data by PRWF\$\$, or not used at all, depending on the function you are requesting. This argument is used only during calls to read or write data; in all other cases, LOC(0) or INTL(0) may be specified in FORTRAN 66 or FORTRAN 77, and NULL () may be specified in PL/I.

When your program performs a read or write operation, the buffer may have any appropriate declaration as long as it begins and ends on a halfword boundary and resides within a single segment. Unless you explicitly instruct them to do so, Prime linkers do not put appropriately declared buffers into memory in such a way that the buffers cross segment boundaries.

For example, a FORTRAN programmer may choose to use an INTEGER*2 array as the buffer. A PL/I programmer might find using a structure useful. Because PRWF\$\$ is the raw data mover for the PRIMOS file system, the data may be of any size and shape.

However, your program must also supply the length of the buffer in halfwords in the next argument.

Length of the Buffer: The length of the buffer is an unsigned number that represents the number of halfwords in the buffer. If it is 0, then no data is transferred to or from the buffer by PRWF\$\$.

Note The maximum value of the *size* argument is 65,535 because it is an unsigned HALF INT argument. If you wish to read or write an entire segment, you cannot do so in one PRWF\$\$ operation, because 65,536 halfwords are in an entire segment. Instead, use two separate calls to PRWF\$\$, specifying a size of 32,768 in each call.

File Positioning Information: Your program uses a FULL INT argument to communicate with the PRWF\$\$ subroutine concerning the file position. Its use depends on the function being performed.

When reading or writing data, this argument's value should always be 0 in PL/I, and 000000 or INTL(0) in FORTRAN. If it is nonzero, PRWF\$\$ first positions the file forward or backward relative to the current position based on the value of this argument (positive or negative). It is recommended that you avoid this functionality, as it is intended only for applications that perform many positioning operations.

When positioning the file, the value of this argument should be the desired position in the file. File position is measured in halfwords. The first halfword of the file is position 0, the second is position 1, the third is position 2, and so on.

Number of Halfwords Actually Read or Written: PRWF\$\$ returns the number of halfwords actually read or written during a read or write operation to your program. The value of this argument is not modified for other PRWF\$\$ operations, but it is recommended that a variable always be passed.

Normally, a read or write operation completes successfully, in which case PRWF\$\$ sets this argument to the same value supplied as the length of the buffer. However, if an error such as END-OF-FILE or DISK-FULL occurs, the number of halfwords actually transferred may range from 0 to the length of the buffer. Therefore, unless the returned error code is 0, this value should always be checked to see how many halfwords were actually transferred.

The Error Code: An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was entirely successful. Otherwise, *code* is always positive. After a call to PRWF\$\$ to read or write data, *code* may have one of many values. *Advanced Programmer's Guide: Appendixes and Master Index* contains a comprehensive list of all standard file system error codes. Codes specific to this operation follow.

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$EOF	1	<p>End of file. The end of the file was reached. If no data were to be transferred, this error code indicates that an attempt was made to position the file past the end-of-file position. The file is positioned at end-of-file when this error occurs.</p> <p>If data were to be read from a file, this error code indicates that the end of the file was reached during the process. However, some data may or may not have actually been read into the buffer. The number of halfwords actually read is returned in the argument described above. This value ranges from 0 to one less than the length of the buffer when the E\$EOF error code is returned. The contents of the buffer following the last halfword transferred (as indicated by the number of halfwords actually read) are undefined.</p> <p>Unless the file is being simultaneously written by another process or on another file unit, any further attempts to read the file without first repositioning it result in E\$EOF being returned with the number of halfwords transferred set to 0.</p> <p>If data were to be written to the file, this error code indicates that an attempt to position the file failed. This means that the value of the file position (a doubleword value) was not 0.</p>
E\$BOF	2	<p>Beginning of file. The beginning of the file was reached. This should occur only if an attempt is made to position the file backward, using relative positioning keys. If the key value is one of the values described above, then this error means that the value of the file position is negative instead of 0 as it should be.</p>
E\$UNOP	3	<p>Unit not open. If the key specified a read operation, then the file unit is open only for writing or is not open at all. If the key specified a write operation, then the file unit is open only for reading or is not open at all. If any other operation was specified, then the file unit is not open.</p>
E\$DKFL	9	<p>The disk is full. This occurs only during a write operation. The buffer could not be completely written to disk because the disk was full, so anywhere from 0 halfwords to one less than the length of the buffer halfwords were written.</p> <p>After the data are written by PRWF\$\$, the file is automatically returned to its pre-write position. Therefore, your program can easily retry the operation by simply calling PRWF\$\$ in the same manner after freeing up disk space in some fashion.</p>

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$MXQB	143	<p>Maximum quota exceeded. This occurs only during a write operation. The buffer could not be completely written to disk because the quota on the directory was exceeded; therefore, anywhere from 0 halfwords to one halfword less than the length of the buffer were written.</p> <p>After the data are written by PRWF\$\$, the file is automatically returned to its pre-write position. Therefore, your program can easily retry the operation by simply calling PRWF\$\$ in the same manner after freeing up some directory space or increasing the directory's quota.</p>

Sample Uses of PRWF\$\$

Here is a sample FORTRAN subroutine that uses PRWF\$\$ to write records to an open file unit. If a DISK-FULL or QUOTA-EXCEEDED error occurs, the user is given an opportunity to delete files and restart the program, and the subroutine retries the write in the correct fashion. This subroutine has the same calling sequence as PRWF\$\$.

```

SUBROUTINE PRWF (KEY, UNIT, LOCBUF, BUFLen, POSN, RNW, CODE)
  INTEGER*2 KEY, UNIT, BUFLen, RNW, CODE
  INTEGER*4 LOCBUF, POSN
C
C $INSERT SYSCOM>ERRD.INS.FTN
C $INSERT SYSCOM>KEYS.INS.FTN
C
C      INTEGER*2 RNW, PATHNM(40), PATHLN, CODE2
C
1     CALL PRWF$$ (KEY, UNIT, LOCBUF, BUFLen, POSN, RNW, CODE)
2     IF (CODE.NE.E$DKFL.AND.CODE.NE.E$MXQB) RETURN
C
C Disk-full or quota-exceeded. Tell the user to clean up the
C directory and type START. First, output the error message,
C along with the full pathname of the file being written (so
C the user knows where to delete files!).
C
C      CALL GPATH$(K$UNIT, UNIT, PATHNM, 80, PATHLN, CODE2) /* Get the
C & /* full pathname of the file open on the file unit.
C      IF (CODE2.EQ.0) GO TO 10 /* Error?
C
C If we can't get the treename, ignore the error, but make the
C error message useful.
C
C      CALL ERRPR$(K$IRTN, CODE, 'Unknown file name', 17,
C & 'PRWF', 4)

```

```

GO TO 20
C
C Otherwise, produce an error message showing the treename.
C
10 CALL ERRPR$(K$IRTN, CODE, PATHNM, PATHLN, 'PRWF', 4)
C
C Now explain to the user what must be done.
C
20 CALL TNOU(0,0) /* Blank line.
CALL TNOUA('Free up some space using DELETE, then type ',
& 43)
CALL TNOU('START to continue.', 19)
CALL TNOU(0,0)
C
C Now invoke a new command level, and hope we return.
C
CALL COMLV$
C
C User has typed START, so retry the write.
C
IF (AND(KEY, K$POSR).NE.0) GO TO 1 /* Retry exactly as
& /* specified if post-positioning desired.
C
CALL PRWF$(KEY, UNIT, LOCBUF, BUFLN, 000000, RNW, CODE)
GO TO 2 /* Otherwise, retry with no positioning because
& /* the positioning portion of the operation has already
& /* happened.
C
END

```

The next example shows a subroutine that simulates a RDLIN\$ interface for a fixed-length record file. The record length is determined by the size of the RDLIN\$ buffer being passed. The calling program calls this subroutine just as it would call RDLIN\$, except that the file is a fixed-length record file, in which each record has trailing spaces. A similar subroutine could be written that simulates a WTLIN\$ interface for fixed-length record files.

```

SUBROUTINE RDLIN(UNIT, BUFFER, BUFLN, CODE)
INTEGER*2 UNIT, BUFFER(1), BUFLN, CODE
C
C BUFFER can be dimensioned to just 1, even though it is probably
C larger, because this subroutine does not reference its contents;
C it simply passes the buffer on to WTLIN$.
C
$INSERT SYSCOM>ERRD.INS.FTN
$INSERT SYSCOM>KEYS.INS.FTN
C
INTEGER*2 RNW, CODE, I, START
C

```

```

        CALL PRWF$$ (K$READ, /* Read a record.
& UNIT, /* The file unit.
& LOC(BUFFER), /* The buffer.
& BUFLN, /* The length of the buffer.
& 000000, /* No positioning.
& RNW, /* The number of halfwords actually read/written.
& CODE) /* The error code.
        IF (CODE.EQ.0) RETURN /* Success.
C
C If an error occurred, we must first fill the buffer with spaces,
C starting with the word following the last word successfully
C read.
C
C If the error code was not end-of-file, however, pretend that no
C halfwords were read at all.
C
        IF (CODE.NE.E$EOF) RNW=0 /* No words read.
C
        IF (RNW.EQ.BUFLN) GO TO 20 /* Unlikely that the buffer was
& /* completely filled.
C
        START=RNW+1 /* Start with the next halfword.
        DO 10 I=START,BUFLN
            BUFFER(I)=' '
10     CONTINUE
C
C Now, if at least one halfword was read, then return an error
C code of zero. This can happen only on an end-of-file error.
C
20     IF (RNW.NE.0) CODE=0
        RETURN
C
        END

```

The next sample subroutine shows how to position a fixed-length record file to a particular record number. This subroutine is called with the file unit to be positioned, the desired position in terms of a record number, and the record length (in halfwords) for the file. It returns a standard file system error code.

```

SUBROUTINE POS (UNIT, RECNO, RECLN, CODE)
    INTEGER*2 UNIT, RECLN, CODE
    INTEGER*4 RECNO
C
$INSERT SYSCOM>ERRD.INS.FTN
$INSERT SYSCOM>KEYS.INS.FTN
C
    INTEGER*2 RNW
C
    CALL PRWF$$ (K$POSN+K$PREA, /* Position absolute.

```

```
& UNIT, /* The file unit.
& LOC(0), /* Unused during a position-only operation.
& 0, /* Also unused.
& RECNO*INTL(RECLEN), /* The file position.
& RNW, /* Unused, but play it safe.
& CODE) /* The error code.
```

C

```
RETURN
END
```

Format of a Variable-length Record File

Variable-length record text files have the following attributes:

- Each line of text can contain from 0 characters to as many as needed.
- Each line of text begins on a halfword boundary.
- Each character in a line of text occupies one 8-bit byte.
- A new-line character (ASCII line-feed, 212 octal) and optionally a pad character (000 octal or 240 octal) follows each line of text to cause each new line to begin on a halfword boundary.
- Space compression is performed by substituting two or more spaces with an ASCII DC1 character (221 octal) followed by a byte representing the number of spaces.
- In general, the last line of a file is terminated by a new-line character and an optional pad character, the same as all other lines in the file. When an unterminated last line is encountered, the behavior of Prime and user-written software is undefined. Most programs ignore all text following the last new-line code. Some programs, however, treat the end of the file as an implicit new-line character, and therefore recognize an unterminated last line as if it were terminated.

For example, suppose that a file named REBEL contains the following lines of text:

```
Rebel, Jean-Fery Position: Composer Born: 1661 Died: 1747
Interests: Swimming, Hiking, Dissonance
```

When stored using the standard PRIMOS variable-record organization, the disk copy of this file would be:

```
R e b e l , J e a n - F e r y D C 1 0 4 P o s i  
t i o n : C o m p o s e r D C 1 0 4 B o r n :  
1 6 6 1 D C 1 0 4 D i e d : 1 7 4 7 L F N U L I n t e  
r e s t s : S w i m m i n g , H i k i n  
g , D i s s o n a n c e L F
```

In the above example, DC1 represents the ASCII DC1 code (221 octal), and 04 represents the number of spaces. The new-line character is represented by LF (212 octal), and NUL indicates a null character (000 octal).

Notice that the final character in the file is a new-line code (LF). It is not followed by a NUL code because the LF code is in the low-order byte of the 16-bit halfword. A NUL code is inserted when it follows an LF code only if the LF code is in the high-order byte of a 16-bit halfword in a disk record. Some programs use the space character (240 octal) as the fill character instead of NUL.

Caution A program that searches forward or backward in a file for particular characters (such as new-line) must take into account the space compression character DC1 (221 octal). If a character is preceded by an odd number of DC1 characters in the file, the character is to be interpreted as a space count. Another concern is that some programs may generate multiple consecutive DC1 and space count characters, which must be treated collectively as the appropriate number of spaces. Finally, it is possible for some programs to output a DC1 followed by a byte containing 0 or 1. These values are to be taken literally, that is, no spaces (the DC1 code is a “no-op”), or one space. Do not interpret a space count of 0 as 256 decimal or 65536 decimal (byte-overflow and halfword-overflow values, respectively).

Format of a Fixed-length Record File

Fixed-length record text files have the following attributes:

- A record length must be defined for the file. This record length should be expressed in terms of the number of bytes per record. If this record length is odd, the blocking factor for the file (explained below) must be even, and the number of records in the file must also be even. However, PRIMOS utilities that support fixed-length records in files support only even-length records.
- A blocking factor should be defined for each program that reads or writes the file. The blocking factor is a number that specifies how many records are read and written during a single read/write operation.
- Each line of text is always r characters in length, where r is the record length for the file.

- If a line that is less than r characters long is to be written to the file, spaces (240 octal) are appended to the line by the user program before writing the record. This leaves the line left-justified in the record.
- If a line that is more than r characters long is to be written to the file, the line is truncated to r characters, and the remaining information on the line is discarded by the user program.
- Each line is implicitly terminated by the end of the record, that is, after the r th character in the record. The last character of record n of a file is immediately followed by the first character of record $n+1$. The new-line character, ASCII LF (212 octal), has no special meaning in fixed-length record files.
- No space compression is performed. The variable-record space compression character, ASCII DC1 (221 octal), has no special meaning in fixed-length record files.
- The last record is immediately followed by the end of the file. Therefore, no data follows the last record of the file. If an attempt is made to read past the last record in a file, PRIMOS returns an error code.
 - It is possible that a partial record (less than r characters long) follows the last complete record in a file. This often indicates that the file is corrupted. When such a partial record exists, the data should be ignored, and an error message should be generated to inform the user of the program that a possible data file corruption has occurred.
 - To ensure that no partial records are inadvertently created, programs that write a fixed-record length file should always truncate the file immediately after writing the last record.

PRIMOS does not maintain information on the record length for the file. All programs that use the file must know the record length. Similarly, PRIMOS does not maintain any history of blocking factors that were used when the file was written.

In fact, PRIMOS performs its own record blocking, optimized for the type of disk being used. A PRIMOS block is referred to as a *record* by other Prime documentation, or as a *PRIMOS disk* or *physical* record when clarification is needed. Usually, PRIMOS stores 2048 bytes per record, plus some housekeeping information. This housekeeping information is used by PRIMOS to provide optimized record blocking in a fashion that is transparent to user programs.

Determining the Blocking Factor

The choice of the appropriate blocking factor for your program depends upon the needs of the program. The only rule imposed by PRIMOS is that the blocking

factor must be even if the record length is odd; PRIMOS allows only reading, writing, positioning, and truncating of files at halfword boundaries.

Aside from that rule, the determination of a blocking factor is fairly straightforward. In general, the larger the blocking factor, the fewer read/write operations needed to peruse a large file. However, a larger blocking factor increases the amount of time needed to perform most individual read/write operations, and also requires more program memory to hold the records. Because PRIMOS provides large amounts of virtual memory, large blocking factors are preferred.

Although PRIMOS does not require a consistent blocking factor to be used when reading and writing the file, it is good practice to use the same blocking factor throughout any given program. Once the file is written, other programs using different blocking factors can read the same file. However, they must use the same record length, or the data is misinterpreted.

Calculating Record Position During Random-access Operations

The position of a record in a fixed-length record file is calculated as follows:

n	The record number (starting at record 0)
r	The record length in characters
p	The position of the record in characters (starting at character 0 in the file)
p	$n * r$

If r is odd, p will be odd whenever n is odd. In such a case, the file should be positioned to record number $n-1$, and the record copied starting at the appropriate low-order byte of the halfword in which record n begins.

Assuming p is even, the halfword position of the record is p divided by 2. The number of halfwords to read in is calculated as follows:

r	The record length
b	The blocking factor ($r * b$ is even)
h	The number of halfwords to read
h	$(r * b) / 2$

Questions and Answers About Text Files

This section answers some typical questions about text storage and retrieval.

- How can I open only an existing file for writing, or reading and writing; that is, without creating a new file if it doesn't exist?

There are several ways to do this. The most straightforward way is to test for the existence of the file before opening it. This is done by calling the SRSFX\$ or SRCH\$\$ subroutines with a *key* value of K\$EXST. For files within segment directories, use the SGD\$EX subroutine.

There is a drawback to this method, however, another user might delete the file between your call to test for its existence and the call to open the file.

A more reliable way to open an existing file is to first open it for reading. If the file does not exist, an error code E\$FNTE is returned. If the file does exist, it is opened for reading, preventing any other users from deleting it.

At this point, call the CH\$MOD subroutine with the file unit number and a *key* argument of K\$WRIT or K\$RDWR. This changes the state of the file unit from read to write or read/write. If an error code such as E\$FIUS (File in use) is returned, close the unit and return the error code.

A sample PL/I subroutine that opens a file for writing, without creating the file, follows.

```
open_existing_file: proc(filename) returns(fixed bin(15));

/* Declarations are not shown except for the input argument
and the returned value. */

dcl filename char(*) var,
    code fixed bin(15);

call srsfx$(k$read+k$getu, filename, unit, type, 0, '', basename,
    suffix_used, code);
if code ^= 0 then return(code);

call ch$mod(k$writ, unit, code);
if code ^= 0 then call clo$fu(unit, ignore_code);

return(code);
end; /* open_existing_file: proc */
```

- How do I choose between SRSFX\$, SGD\$OP, SRCH\$\$, and others?

If you want to open a file that may or may not have one or more suffixes appended to its name, use SRSFX\$. If you are programming in PL/I or Pascal, and you want to open a file by using a pathname (such as MYDIR>THIS_FILE), use SRSFX\$.

If you want to open a file that is not a pathname, and is in the current directory, use SRCH\$\$ for maximum performance.

If you want to open a file that is in a segment directory that you have already opened on another file unit, use SGD\$OP.

If you want to close a file by name, use CLO\$FN. If you want to close a file by unit number, use CLO\$FU.

If you want to test for the existence of a file within a segment directory, use SGD\$EX. If you want to test for the existence of a file within a file directory, make the decision as to which subroutine to use (SRSFX\$ or SRCH\$\$) as if you were opening the file, and use the K\$EXST key.

If you want to delete a file, use FIL\$DL if it is a member of a directory, or SGD\$DL if it is a member of a segment directory.

If you want to change the access of an open file unit, always use CH\$MOD.

If you want to open a member of an open segment directory, use SGD\$OP.

- What is the K\$GETU additive key for? What if I don't use it?

Add the K\$GETU key to K\$READ, K\$WRIT, and K\$RDWR when you open files using the SRSFX\$ or SRCH\$\$ subroutines. This causes an available file unit to be selected by PRIMOS and returned in the *unit* argument of the calling sequence.

If you do not specify K\$GETU when opening a file, PRIMOS treats the *unit* argument as an input-only argument, and uses the number in *unit* as the file unit number. This is not recommended practice.

Dynamic unit allocation has been preferred since PRIMOS Revision 16, when the K\$GETU functionality was first made available. Only programs that have to maintain past behavior for compatibility reasons should pass file units to PRIMOS when opening files. All other programs should use K\$GETU or specify -10000 as the unit number, whichever is needed by the procedure called.

If you decide not to use K\$GETU, be aware that there are two additional error codes that may be returned. The E\$UIUS (Unit in use) code is returned if the file unit you specified is already open. The E\$BUNT (Bad unit number) code is returned if the file unit you specified is not a legal file unit number. However, the error code E\$FUIU (All file units in use) is not returned, because you are not asking PRIMOS to allocate a new file unit.

Data Storage and Retrieval

8



Certain applications require the ability to maintain one or more databases on disk. Prime provides several facilities for building and manipulating databases from within programs and via interactive sessions. These include MIDASPLUS, DBMS, DISCOVER, Prime INFORMATION, and PRISAM. Information on these products can be found in the *50 Series Technical Summary*.

In addition, PRIMOS provides several facilities to allow applications to perform their own database management. These facilities are

- The PRIMOS file system, which allows the hierarchical organization of files, identified by name or by number.
- The semaphore mechanism, which provides a method of handling concurrency problems that occur when several users attempt to access the same database simultaneously.
- The 50 series architecture, which facilitates rapid access by permitting the sharing of memory-resident data among several users.

This chapter describes the PRIMOS file system as used by database management software. *Volumes II and III* of the *Subroutines Reference* series describe the use of semaphores and shared memory for database management purposes. In addition, the *50 Series Technical Summary* and the *System Architecture Reference Guide* describe semaphores and shared memory in detail.

File Organization

Two useful file organizations provided by the PRIMOS file system are

- Segment directories, for organizing data files by number
- File directories, for organizing data files by name

This chapter describes how to manipulate segment directories and file directories under PRIMOS. This is followed by a short discussion on the reading and writing of data files (whether in segment directories or file directories). This chapter ends with a question and answer section.

Segment Directories

Your program manipulates files within a segment directory by first opening the segment directory itself and then positioning the segment directory to the desired file by using SGDR\$\$\$. Your program then calls SGD\$OP, SGD\$EX, or SGD\$DL with the file unit number of the open segment directory to manipulate members of the segment directory. When your program is finished with the segment directory, it closes the segment directory unit by calling CLO\$FU.

Once a file within a segment directory is opened, you can treat it as a text file (described in Chapter 7, Text Storage and Retrieval) or a data file (described later in this chapter). The remainder of this section describes

- Subroutines used to access segment directories
- How to open a segment directory
- How to position a segment directory
- How to extend a segment directory
- How to open a file within a segment directory
- How to delete a file within a segment directory
- How to scan a segment directory

Subroutines Used to Access Segment Directories

The subroutines most often used when accessing segment directories follow.

<i>Subroutine</i>	<i>Use</i>
SRSFX\$	Accepts a pathname and calls SRCH\$\$\$ to manipulate the object according to the specified key. SRSFX\$ calls SGDR\$\$\$ to position to a file.
SGDR\$\$\$	Positions an open segment directory to a specified member. Positioning is necessary before calling SRCH\$\$\$ or SGD\$DL to operate on a member of a segment directory. In addition, SGDR\$\$\$ is used to expand and truncate segment directories, and to read the position of an open segment directory.
SGD\$OP	Opens a member file of an open segment directory, optionally creating the member if it does not already exist.
SGD\$DL	Deletes a member file of an open segment directory.
SGD\$EX	Tests for the existence of a member file of an open segment directory.

How to Open a Segment Directory

Your program must open a segment directory before it can access members of the segment directory. Use the SRSFX\$ or SRCH\$\$ subroutine to open the segment directory. Your program must open the segment directory for reading and writing if it is going to create or delete members. If your program is going to open and close only existing members, it need open the segment directory only for reading. Opening a segment directory only for writing (but not reading) is not recommended.

When your program calls the SRSFX\$ or SRCH\$\$ subroutine to open a segment directory, it provides

- The name of the segment directory to be opened
- A key that specifies how the segment directory is to be opened

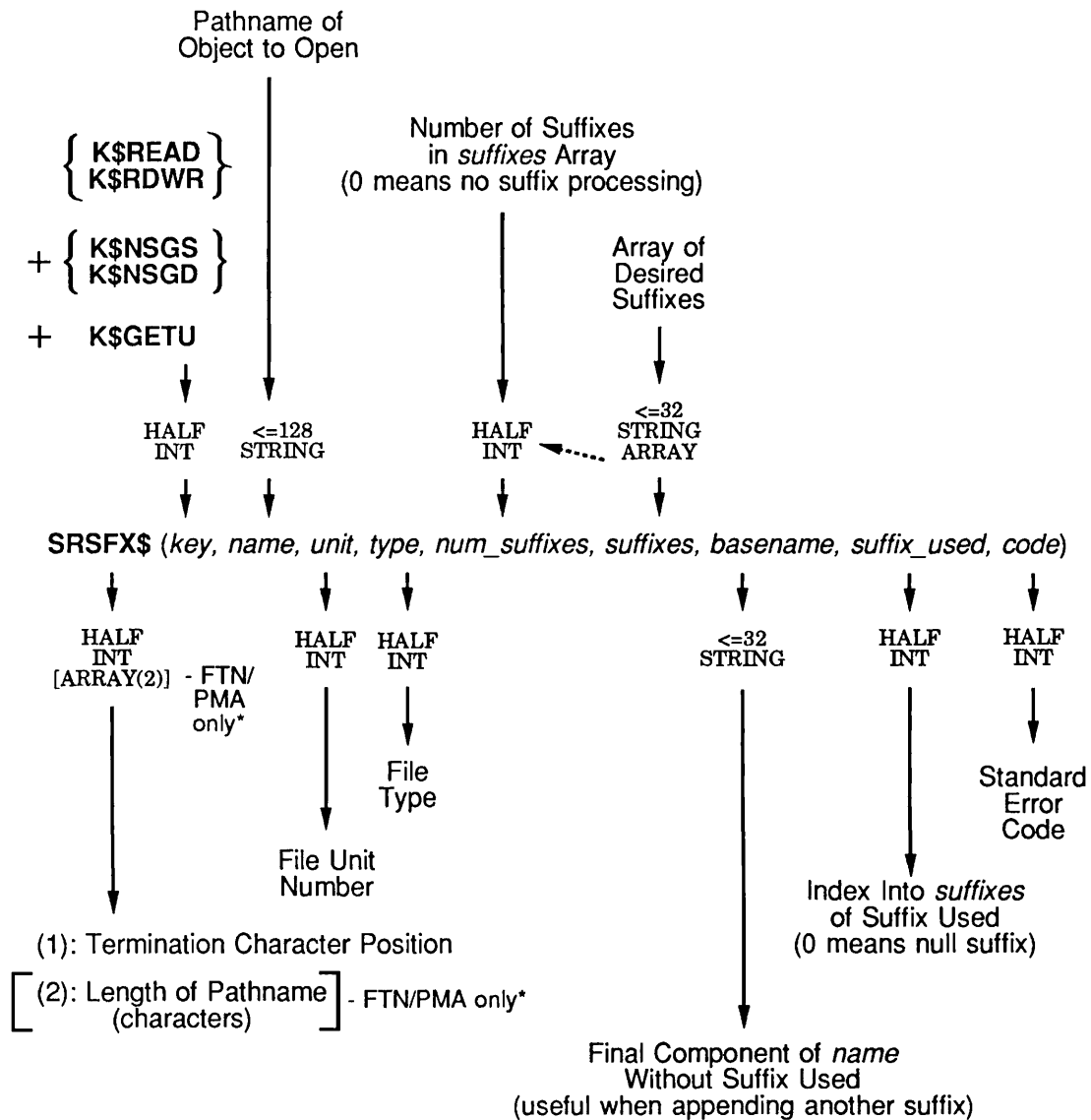
The SRSFX\$ or SRCH\$\$ subroutine attempts to open the specified segment directory, and returns to your program

- An error code that indicates whether the operation was successful
- A file unit number that identifies the open segment directory. Your program uses this number when performing operations (such as extend and truncate) on an open segment directory or when it manipulates members of a segment directory.
- The file type, indicating the type of segment directory just opened (including SEGSAM and SEGDM)

Your program should close a segment directory when finished with it. Do this with CLO\$FU to close the file unit, as described in Chapter 7, Text Storage and Retrieval.

This section describes the input and output parameters that apply when calling SRSFX\$ and SRCH\$\$, and then shows a sample call to SRCH\$. Figure 8–1 illustrates the calling sequence of SRSFX\$ to open a segment directory; Figure 8–2 illustrates the calling sequence of SRCH\$ to open a segment directory.

Open a Segment Directory, With Possible Suffix



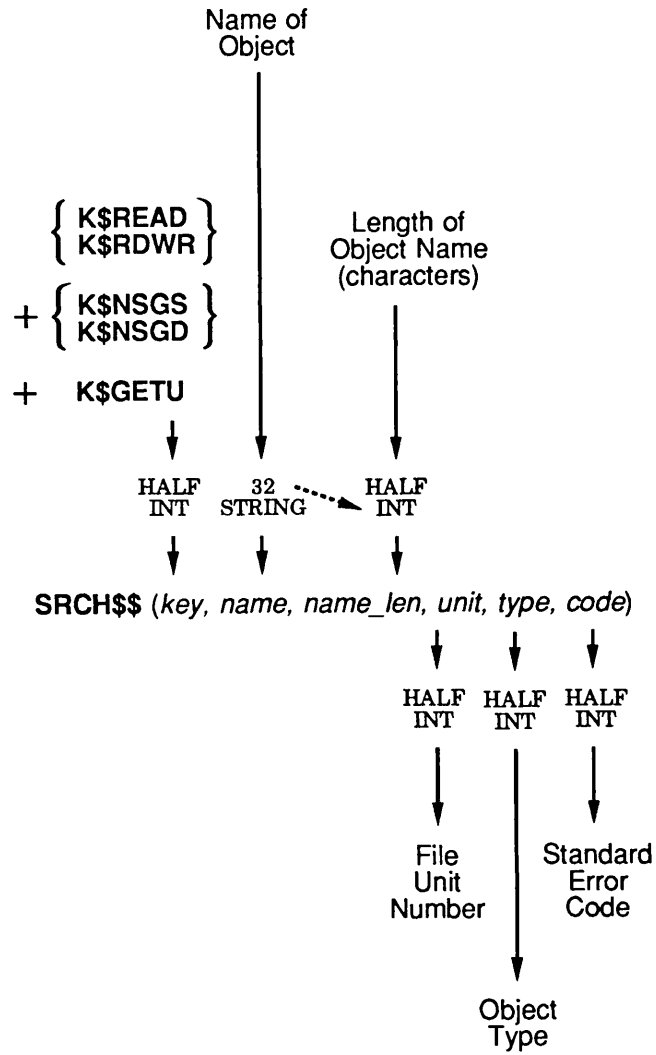
Side Effects: May reset current attach point.

* Function value is returned in L-register; typically, you need only to declare as HALF INT, because first datum is all you need and is in A-register. Otherwise, you must declare it as FULL INT to make it work.

Q08.01.D10056.3LA

Figure 8-1. Calling Sequence of SRSFX\$ to Open a Segment Directory

Open Segment Directory by Object Name



Q08.02.D10056.3LA

Figure 8-2. Calling Sequence of SRCH\$\$ to Open a Segment Directory

Name of the Segment Directory: The rules for specifying the segment directory name depend on the system subroutine being called. The filename may be a pathname if `SRSFX$` is being used. If `SRCH$$` is being used, the filename must be an entryname; that is, it cannot contain a `>` symbol.

Key: The *key* argument is calculated as follows:

$$key = action + newfile [+ K$GETU]$$

The values and meanings are

<i>Value</i>	<i>Meaning</i>
<i>action</i>	Specifies whether the segment directory is to be opened for reading or for both reading and writing. These states are often identified by the mnemonics R and RW (or WR), respectively. The keywords used when opening segment directories are

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
<code>K\$READ</code>	1	Open the segment directory for reading.
<code>K\$RDWR</code>	3	Open the segment directory for both reading and writing.

When a segment directory is open for reading, and an attempt is made to create or delete a member of it or to change its size, an error code of `E$UNOP` (Unit not open) is returned.

<i>newfile</i>	Specifies what type of segment directory should be created if the segment directory does not already exist. (The segment directory is created only if it is being opened for writing or for reading and writing.) The keywords used for segment directories are
----------------	---

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
<code>K\$NSGS</code>	2048	Create a new threaded (SAM) segment directory.
<code>K\$NSGD</code>	3072	Create a new directed (DAM) segment directory.

SAM and DAM segment directories differ only in performance and storage efficiency, as described in Chapter 3, Accessing the PRIMOS File System.

Note The type of a segment directory (SAM or DAM) is unrelated to the type of any of its members. For example, a SAM segment directory may contain DAM files.

K\$GETU Specifies that PRIMOS is to use an available file unit, and return the selected file unit number in the *unit* parameter of the calling sequence.

Error Code: An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was entirely successful. Otherwise, *code* is always positive. After a call to SRSFX\$ or SRCH\$\$ to open a segment directory, *code* may have one of many values. The *Advanced Programmer's Guide: Appendices and Master Index* contains a comprehensive list of all standard file system error codes. Error codes specific to this operation follow.

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$FIUS	5	The segment directory being opened is already open on another file unit, or by another user. Under normal lock settings, a segment directory that is open for reading can have many file units open to it, but a segment directory open for writing can have only one file unit open to it. Therefore, a segment directory that is open for reading cannot be opened for writing, nor can a segment directory that is open for writing be opened for writing or for reading. See Chapter 10, File Attributes, for more information on the read/write lock.
E\$NRIT	10	Insufficient access rights. If the segment directory being opened already exists, this means that the user does not have sufficient access to the segment directory. If the segment directory does not exist, then the user does not have Add access to the directory in which the segment directory is to be created. For calls to SRSFX\$, this error code may indicate a problem attaching to the directory specified by the pathname argument of the calling sequence. In this case, the user does not have Use access to at least one directory in the pathname.

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$FNTE	15	Not found. The segment directory being opened does not exist. The <i>action</i> portion of the <i>key</i> argument is probably K\$READ, otherwise the segment directory would be created. For calls to SRSFX\$, this error code may indicate a problem attaching to the directory specified by the pathname argument of the calling sequence. In this case, at least one directory in the pathname does not exist. Even if the <i>action</i> portion of the <i>key</i> argument is K\$RDWR, no directory is created via a call to SRSFX\$. Use the DIR\$CR subroutine to create a directory.
E\$BNAM	17	Illegal name. The filename or pathname as supplied by the calling program is not valid. See the <i>PRIMOS User's Guide</i> for a description of the valid syntax for a filename and pathname.
E\$WTPR	56	The disk is write-protected. A segment directory cannot be opened for writing, nor can it be created, on a write-protected disk. (Disk write-protection is enabled using the ADDISK command, described in the <i>Operator's Guide to System Commands</i> .)
E\$MXQB	143	Maximum quota exceeded. This error can occur only if a new segment directory is being created, and cannot occur if the action portion of the <i>key</i> argument is K\$READ.
E\$NFAS	189	Top-level directory not found or inaccessible (SRSFX\$ only). The first directory name supplied in the pathname could not be located on any of the system disks.

File Unit Number: The returned file unit number is valid only when the returned error code is 0. After opening a segment directory, your program passes the returned file unit number to other system subroutines (such as SGDR\$\$ and SGD\$OP) to manipulate the segment directory and its members.

Once your program closes the segment directory, the file unit number is returned to the free pool for reuse by PRIMOS when another file is opened.

File Type: The returned file type is valid only when the returned error code is 0, and the segment directory is actually opened. The file type is one of the following five values:

<i>Value</i>	<i>Meaning</i>
0	A SAM file has been opened. Use RDLIN\$, WTLIN\$, PRWF\$\$, and similar subroutines to read or write it. (See Chapter 7, Text Storage and Retrieval, for information on how to do this.)

- 1 A DAM file has been opened. Use RDLIN\$, WTLIN\$, PRWF\$\$, and similar subroutines to read or write it. (See Chapter 7, Text Storage and Retrieval, for information on how to do this.)
- 2 A SAM segment directory (SEGSAM) has been opened. Use SGDR\$\$ to operate on members of this segment directory.
- 3 A DAM segment directory (SEG DAM) has been opened. Use SGDR\$\$ to operate on members of this segment directory.
- 4 A root-entry directory has been opened. Use DIR\$LS, DIR\$RD, or ENT\$RD to read information on files in this directory. (See the section entitled File Directories, later in this chapter.)

Note It is important to understand that opening a segment directory is very different from opening a segment directory member. The above section describes how to open a segment directory. Information on opening members of a segment directory is in an ensuing section entitled How to Open a Member File Within a Segment Directory.

Example: The following example shows how a FORTRAN program would open the object MYSEGDIR in the current directory for reading and writing, creating a SAM segment directory if it does not already exist:

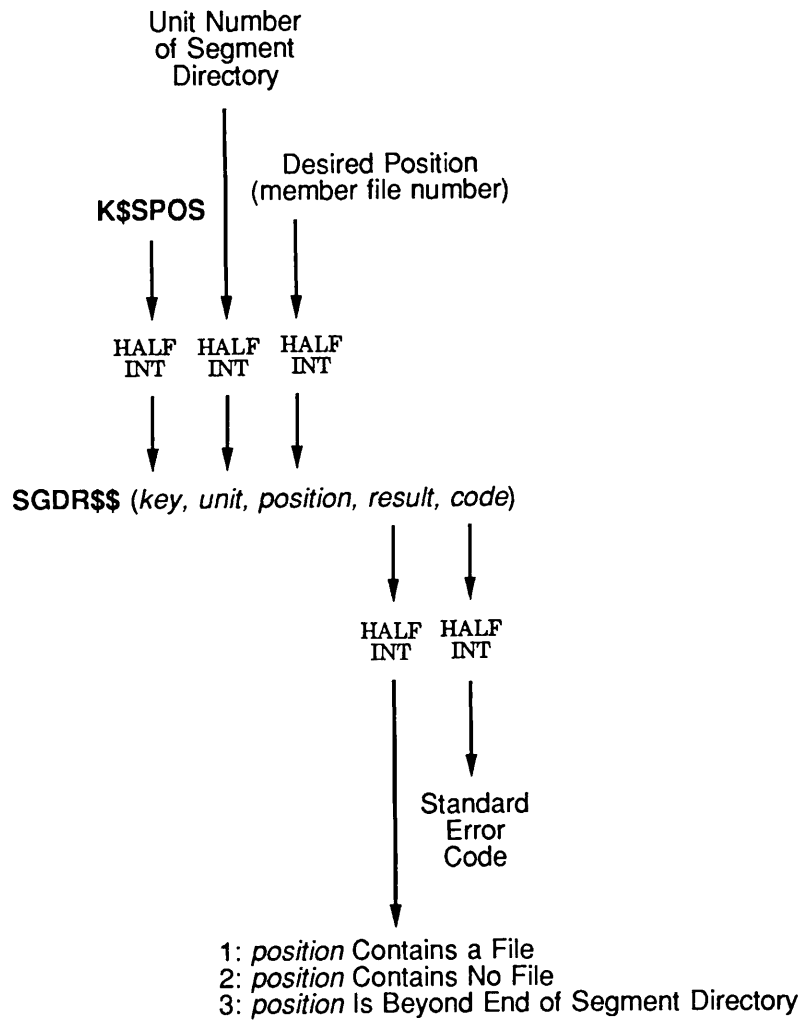
```
CALL SRCH$$ (K$RDWR+K$NSGS+K$GETU, 'MYSEGDIR', 8, UNIT, TYPE, CODE)
IF (CODE.NE.0) GO TO 1000
.
.
.
1000 CALL ERRPR$ (K$IRTN, CODE, 'MYSEGDIR', 8, 'MYPROGRAM', 9)
RETURN
```

How to Position a Segment Directory

Before opening a file within a segment directory, your program must position the segment directory to the appropriate member file. You position a segment directory by using the SGDR\$\$ subroutine. When your program calls SGDR\$\$ to position a segment directory, it provides

- The file unit of the open segment directory
- A key that specifies that a position operation is to be performed
- The desired position of the segment directory, also known as the **member file number**

Position Segment Directory to Entry Number



Q08.03.D10056.3LA

Figure 8-3. Calling Sequence of SGDR\$\$ to Position a Segment Directory

Desired Position of the Segment Directory: Your program passes the desired position of the segment directory, which ranges from 0 to 65535 (-1 signed), inclusive. The resulting position in the segment directory may or may not have a member file present.

Error Code: An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was entirely successful. Otherwise, *code* is always positive. After a call to SGDR\$\$ to position a segment directory, *code* may have one of many values. *Advanced Programmer's Guide: Appendices and Master Index* of this series contains a comprehensive list of all standard file system error codes. Error codes specific to this operation are:

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$EOF	1	End of file. The desired position is beyond the end of the segment directory. The segment directory is left positioned at the end of the directory. If a call to SRCH\$\$ with a <i>key</i> argument of K\$WRIT or K\$RDWR is performed within the segment directory at this point, a new file is created and the segment directory is automatically extended by one file entry.
E\$UNOP	3	Unit not open. The specified file unit is not open. This usually indicates a program error, although it can also be the result of the user exiting the program via CONTROL-P, typing CLOSE ALL, and then starting the program again by typing START.

File Existence Indicator: If the positioning operation is successfully performed, SGDR\$\$ returns a file existence indicator (result) to your program. This variable takes on one of the following values:

<i>Value</i>	<i>Meaning</i>
1	The specified position is within the bounds of the segment directory, and a member file exists at this position. In other words, a member file exists with this member file number.
0	The specified position is within the bounds of the segment directory, but no member file exists at this position. In other words, no file exists with this member file number.
-1	The specified position is at the end of the segment directory. Therefore, no member file exists at this position. If a new member file is created at this position, the segment directory is automatically extended to accommodate it; however, the file number of the newly created member file is not necessarily <i>position</i> , but is instead the end-of-file member number plus one (which may be less than or equal to <i>position</i>).

Example: The following sample use of SGDR\$\$ positions the segment directory open on file unit SGUNIT to member file number 5. If the file exists, it prints the word EXISTS. If the file does not exist, it prints the words NOT THERE. If the position is at the end of the segment directory, it prints the words AT END OF SEGDIR.

```

CALL SGDR$$ (K$$SPOS, SGUNIT, 5, INDIC8, CODE)
  IF (CODE.NE.0) GO TO 1000
  IF (INDIC8) 10, 20, 30
C
10  CALL TNOU('AT END OF SEGDIR', 16)
    GO TO 100
20  CALL TNOU('NOT THERE', 9)
    GO TO 100
30  CALL TNOU('EXISTS', 6)
    GO TO 100
C
100 .
    .
    .
1000 CALL ERRPR$(K$IRTN, CODE, 'SGDR$$ error', 12, 'MYPROGRAM', 9)
      RETURN

```

How to Extend a Segment Directory

Sometimes when you attempt to create a new member file within a segment directory, the member file number represents a position beyond the end of the segment directory. This situation is indicated by a returned error code of E\$EOF when the program attempts to position the segment directory. To circumvent this situation, your program must extend the segment directory.

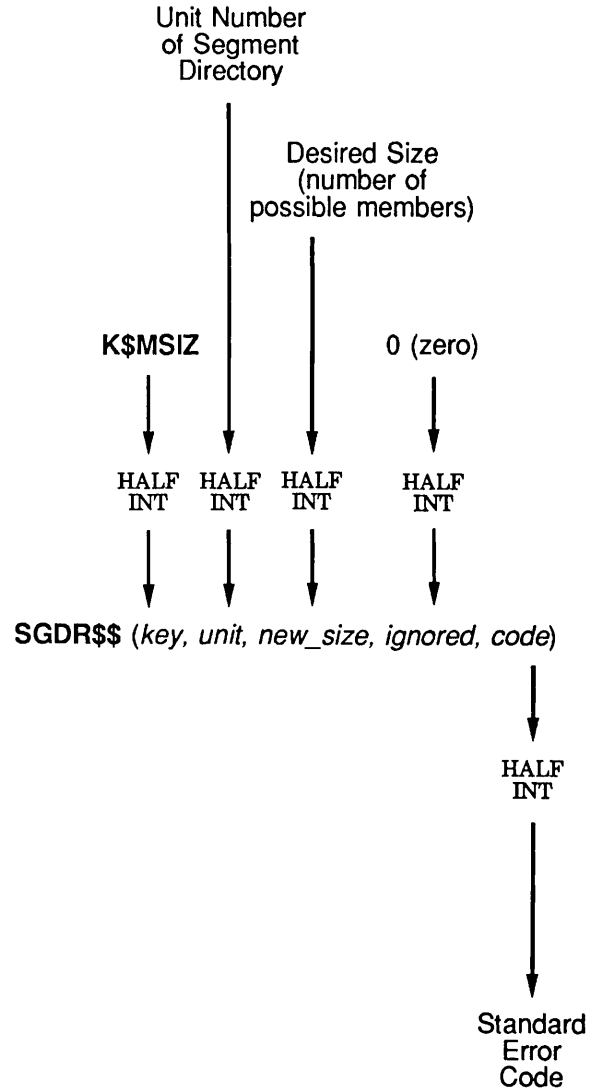
When a segment directory is extended, PRIMOS adds new placeholders for segment directory members. These placeholders represent nonexistent files: they may be used by your program to hold new files. Use the SGDR\$\$ subroutine to extend a segment directory. When your program calls SGDR\$\$ to extend a segment directory, it provides

- The file unit of the open segment directory.
- A key that specifies that an extend operation is to be performed
- The desired size of the segment directory, also known as the new end-of-segment-directory location

This section describes the input and output parameters that apply when calling SGDR\$\$ to extend a segment directory, and then shows a sample call to SGDR\$\$.

Figure 8-4 illustrates the calling sequence of SGDR\$\$ to extend a segment directory.

Extend (or Truncate) Segment Directory



Side Effects: Position of *unit* after operation is at end of segment directory if *code* is 0, undefined otherwise.

Q08.04.D10056.3LA

Figure 8-4. Calling Sequence of SGDR\$\$ to Extend a Segment Directory

Desired Size of the Segment Directory: Your program passes the desired size of the segment directory (*new_size*), which ranges from 0 to 65535 (–1 signed), inclusive.

Note You cannot use K\$MSIZ to extend a segment directory to a full length of 65,536 member entries, because the data type of the desired size cannot accommodate the number 65536. If it is necessary to extend a segment directory to 65,536 entries,

1. extend it to 65,535 entries using SGDR\$\$, which leaves the segment directory unit positioned at the end of the segment directory;
2. use SGDSOP to create a member file, which, when at the end of a segment directory, automatically extends the segment directory by one entry;
3. use CLOSFU to close the newly created (and empty) member file;
4. use SGDSDL to delete the member file, leaving an empty entry at member file number 65535.

Keep in mind that a segment directory of size *position* can have member file numbers ranging from 0 through position –1. For example, extending a segment directory to 65 entries for member file numbers ranging from 0 through 64, but not including member file number 65.

The Error Code: An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was entirely successful. Otherwise, *code* is always positive. After a call to SGDR\$\$ to extend a segment directory, *code* may have one of many values. *Advanced Programmer's Guide: Appendices and Master Index* of this series contains a comprehensive list of all standard file system error codes. Error codes specific to this operation are

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$UNOP	34	Unit not open. The specified file unit is open only for reading, or is not open. This usually indicates a program error, although it can also be the result of the user exiting the program via CONTROL–P, typing CLOSE ALL, and then typing START.
E\$DKFL	9	The disk is full. The segment directory may or may not have been extended, but it has not been extended to the desired size. The segment directory is left positioned at the end of the directory.
E\$MXQB	143	Maximum quota exceeded. The segment directory may or may not have been extended, but it has not been extended to the desired size. The segment directory is left positioned at the end of the directory.

Example: The following sample use of SGDR\$\$ extends the segment directory open on file unit SGUNIT to hold 205 entries.

```
CALL SGDR$$ (K$MSIZ, SGUNIT, 205, IGNORE, CODE)
  IF (CODE.NE.0) GO TO 1000
  .
  .
  .
C
1000 CALL ERRPR$ (K$IRTN, CODE, 'SGDR$$ error', 12, 'MYPROGRAM', 9)
      RETURN
```

How to Open a Member File Within a Segment Directory

Before data in a member file are accessed, the file must be opened. To open a file within a segment directory, have your program position the segment directory by using the SGDR\$\$ subroutine, and then open the member file by using the SGD\$OP subroutine. When your program calls the SGD\$OP subroutine, it provides the following items of information

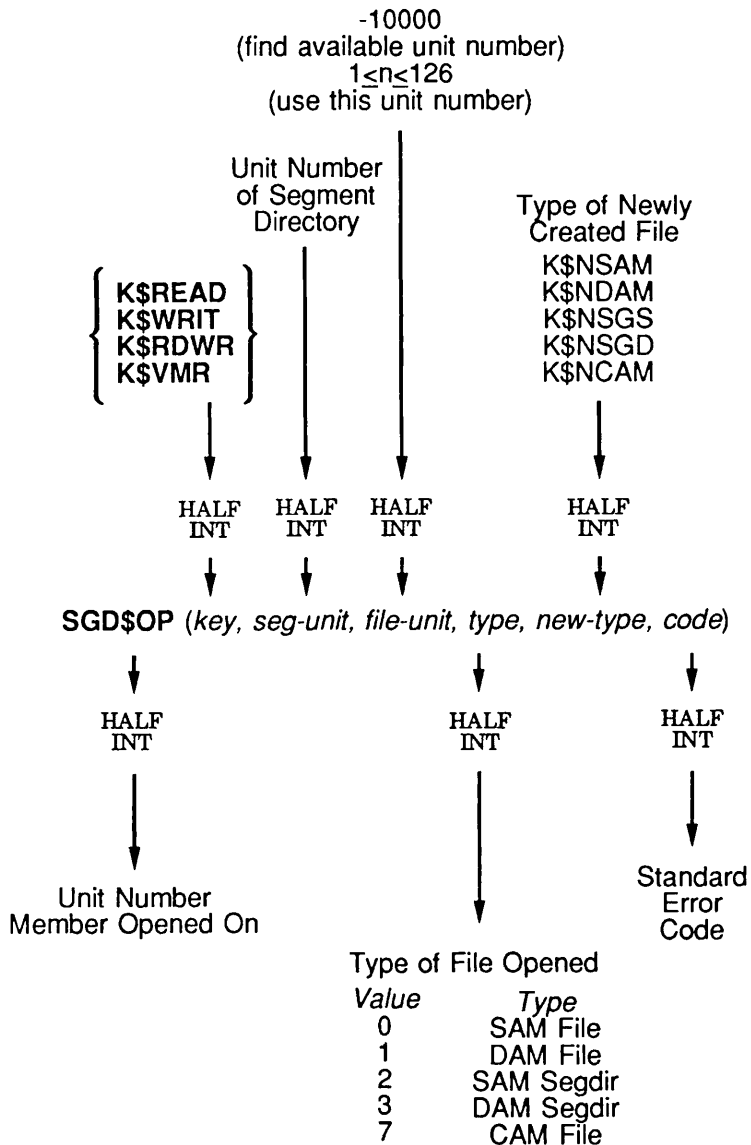
- The file unit number of the open segment directory that is positioned to the member file to be opened
- A key that specifies how the member file is to be opened

The SGD\$OP subroutine attempts to open the specified file and returns to your program

- An error code indicating whether the operation was successful
- A file unit number that identifies the open file. This number is used when performing operations (such as read and write) on an open file.
- The file type, indicating the type of file just opened (including SAM, DAM, CAM, SEGSAM, and SEG DAM).

This section describes the input and output parameters used when calling SGD\$OP, and then shows a sample call to SGD\$OP. Figure 8-5 illustrates the calling sequence of SGD\$OP.

Open Member of Segment Directory



Side Effects: If *seg-unit* is at end of segment directory and *key* is K\$WRIT or K\$RDWR, SGD\$OP attempts to automatically extend segment directory by one entry, which also repositions *seg-unit* to new *end-of-segdir* position; otherwise, size of segment directory and position of *seg-unit* remain unchanged.

Q08.05.D10056.3LA

Figure 8-5. Calling Sequence of SGD\$OP

Key: The *key* argument equals *action*. Its values and meanings are

<i>Value</i>	<i>Meaning</i>
<i>action</i>	Specifies how the file is to be opened. This distinguishes between a file being open for reading, for writing, and for both reading and writing. These states are often identified by the mnemonics R, W, and RW (or WR), respectively. The keywords used for opening files are

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
K\$READ	1	Open a file for reading only.
K\$WRIT	2	Open a file for writing only.
K\$RDWR	3	Open a file for reading and writing.
K\$VMR	16	Open a file for VMFA read, used only before calling one of the EPF subroutines to initialize or execute the file.

If your program attempts to write to a file that is open for reading, an error code of E\$UNOP (Unit not open) is returned to your program. This same error code is returned if your program attempts to read a file that is open for writing.

Desired Unit Number: Your program passes the value -10000 to indicate that PRIMOS is to choose an available file unit number. If you want your program to specify the unit number instead of letting PRIMOS select the number, your program supplies a unit number between 1 and 126 (or 1 and 15 for a program running under PRIMOS II). SGD\$OP returns the chosen file unit number used as the value of the SGD\$OP function.

Error Code: An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was entirely successful. Otherwise, *code* is always positive. After a call to SGD\$OP to open a segment directory member, *code* may have one of many values. *Advanced Programmer's Guide: Appendices and Master Index* of this series contains a comprehensive list of all standard file system error codes. Error codes specific to this operation are

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$FIUS	5	<p>File in use. The file being opened is already open on another file unit, or is being used by another user. Normally, a file that is open for reading cannot be opened for writing, nor can a file open for writing be opened for reading. A file that is open for writing can have only one file unit open to it, whereas a file open for reading can have many file units open to it.</p> <p>If you expect your program to open a file that may occasionally be in use by another process for a short time, consider having your program repeatedly attempt to open an in-use file for 30 seconds or a minute, sleeping one second in between each attempt by calling SLEEP\$.</p> <p>See Chapter 10, File Attributes, for more information on the read/write lock.</p>
E\$DKFL	9	The disk is full. This error can occur only if a new file is being created, and hence cannot occur if the action portion of the <i>key</i> argument is K\$READ.
E\$NRIT	10	Insufficient access rights. If the file being opened already exists, this means that the user does not have sufficient access to the parent segment directory. If the file does not exist, then the user does not have Write access to the parent segment directory in which the file is to be created.
E\$FNFS	16	Not found in segment directory. The file being opened does not exist in the segment directory. The action portion of the <i>key</i> argument is typically K\$READ; otherwise the file would have been created.
E\$MXQB	143	Maximum quota exceeded. This error can occur only if a new file is being created, and hence cannot occur if the action portion of the <i>key</i> argument is K\$READ.
E\$NINF	159	No information. This indicates that some error occurred, but the user does not have List access to the directory involving the error. In such a case, the E\$NINF error code is always returned to prevent the user or calling program from being able to determine any information on the directory. Therefore, this error code is to be interpreted as any possible error, in addition to a case of insufficient access.

File Unit Number: The returned file unit number is valid only when the returned error code is 0. After opening a file, your program passes the returned file unit number to other system subroutines (such as PRWF\$\$ and RDLIN\$) to read, write, and position the file.

Once your program closes the file, the corresponding file unit number can no longer be used. It may then be reused by PRIMOS when another file is opened.

File Type: The returned file type is valid only when the returned error code is 0. The file type is one of the following five values:

<i>Value</i>	<i>Meaning</i>
0	A SAM file has been opened. Use RDLIN\$, WTLIN\$, PRWF\$\$, and similar subroutines to read or write it. (See Chapter 7, Text Storage and Retrieval, for information on how to do this.)
1	A DAM file has been opened. Use RDLIN\$, WTLIN\$, PRWF\$\$, and similar subroutines to read or write it. (See Chapter 7, Text Storage and Retrieval, for information on how to do this.)
2	A SAM segment directory (SEGSAM) has been opened. Use SGDR\$\$ to operate on members of this segment directory.
3	A DAM segment directory (SEGDAM) has been opened. Use SGDR\$\$ to operate on members of this segment directory.
7	A CAM file has been opened. Use RDLIN\$, WTLIN\$, PRWF\$\$, and similar subroutines to read or write it. (See Chapter 7, Text Storage and Retrieval, for information on how to do this.)

New File Type: The new file type specifies what type of file should be created if the file does not already exist. (The file is created only if it is being opened for writing or for reading and writing.) The keywords used for text or data files are

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
K\$NSAM	0	Create a new threaded (SAM) file. (This is the default.)
K\$NDAM	1024	Create a new directed (DAM) file.
K\$NSGS	2048	Create a new threaded (SAM) segment directory.
K\$NSGD	3072	Create a new directed (DAM) segment directory.
K\$NCAM	4096	Create a new contiguous (CAM) file.

SAM and DAM files differ only in performance and storage efficiency, as described in Chapter 3, Accessing the PRIMOS File System.

Examples: The following example shows how a FORTRAN program would open the file at the current position for reading in the segment directory open on unit SGUNIT.


```

UNIT=SGD$OP (K$READ, SGUNIT, -10000, TYPE, CODE)
  IF (CODE.NE.0) GO TO 1000
  .
  .
1000 CALL ERRPR$(K$IRTN, CODE, 'Segdir file', 'MYPROGRAM', 9)
      RETURN

```

The next example illustrates the use of the *new-type* value in the calling sequence to SGD\$OP. The file at the current position is opened for reading and writing in the segment directory open on unit SGUNIT. If it does not exist, it is created as a DAM (directed) type file. Only the subroutine call itself is shown; the error code is examined in the same fashion as shown in the above example.

```

UNIT=SGD$OP (K$RDWR, SGUNIT, -10000, TYPE, K$NDAM, CODE)

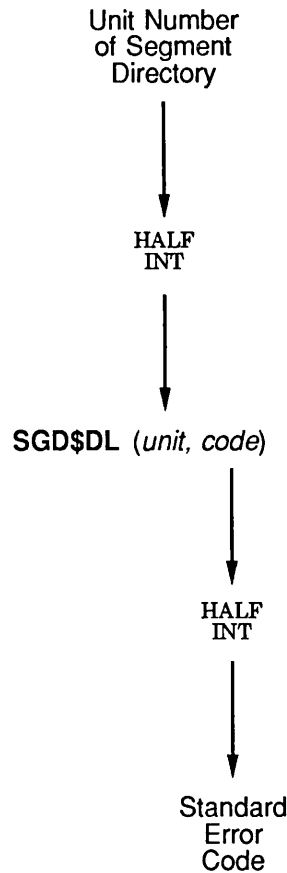
```

How to Delete a Member File Within a Segment Directory

Your program can delete a member file within a segment directory by positioning the segment directory with the SGDR\$\$ subroutine, and then using the SGD\$DL subroutine to actually delete the file. When calling the SGD\$DL subroutine, your program provides the file unit number of the open segment directory that is positioned to the file to be deleted. The SGD\$DL subroutine attempts to delete the specified file, and returns an error code indicating whether the operation was successful.

This section describes the input and output parameters used when calling SGD\$DL, and then shows a sample call to SGD\$DL. Figure 8-6 illustrates the calling sequence of SGD\$DL.

Delete Member of Segment Directory



Q08.06.D10056.3LA

Figure 8-6. Calling Sequence of SGD\$DL

Error Code: An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was entirely successful. Otherwise, *code* is always positive. After a call to SGD\$DL to delete a segment directory member, *code* may have one of many values. *Advanced Programmer's Guide: Appendices and Master Index* of this series contains a comprehensive list of all standard file system error codes. Error codes specific to this operation follow.

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$FDEL	11	File open on delete. The file to be deleted is already open on another file unit, or is being used by another user.
E\$FNLS	16	Not found in segment directory. The file being deleted does not exist in the segment directory, or the segment directory is positioned at the end of the directory.
E\$SUNO	31	Segdir unit not open. The segment directory unit is open only for reading, or is not open at all.

Example: The following example shows how a FORTRAN program would delete the file at the current position in the segment directory open on unit SGUNIT:

```

CALL SGD$DL(SGUNIT, CODE)
IF (CODE.NE.0) GO TO 1000
.
.
.
1000 CALL ERRPR$(K$IRTN, CODE, 'Segdir file', 'MYPROGRAM', 9)
RETURN

```

Scanning a Segment Directory

If you want your program to scan a segment directory for all of its members, you use the SGDR\$\$ subroutine. In addition to the functions described earlier, this subroutine can find the file numbers of all of the members of a segment directory. This is referred to as the **find full entry** function.

In addition, you can use the SGDR\$\$ subroutine to find all of the unused file numbers in a segment directory. This capability is useful when your program needs to create a new segment directory member. Your program can scan the segment directory for free member numbers, and use one of these numbers for the new member it is going to create. This capability is referred to as the **find free entry** function.

The find full entry and find free entry functions are very similar. Your program provides the starting position of the segment directory, and SGDR\$\$ searches the segment directory for the first full or free entry starting at that position and continuing toward the end of the segment directory. When SGDR\$\$ finds the appropriate entry, it leaves the segment directory at that position and returns the position to your program. (The returned position also serves as the file number of a new or existing segment directory member.)

The only difference between the two functions is that the find full entry function of SGDR\$\$ sets the position of the segment directory at the first position that corresponds to an existing member of the segment directory, whereas the find

free entry function of SGDR\$\$ sets the position of the segment directory at the first position that corresponds to an unused member number in the segment directory.

When your program calls SGDR\$\$ to search a segment directory for a full or free position, it provides

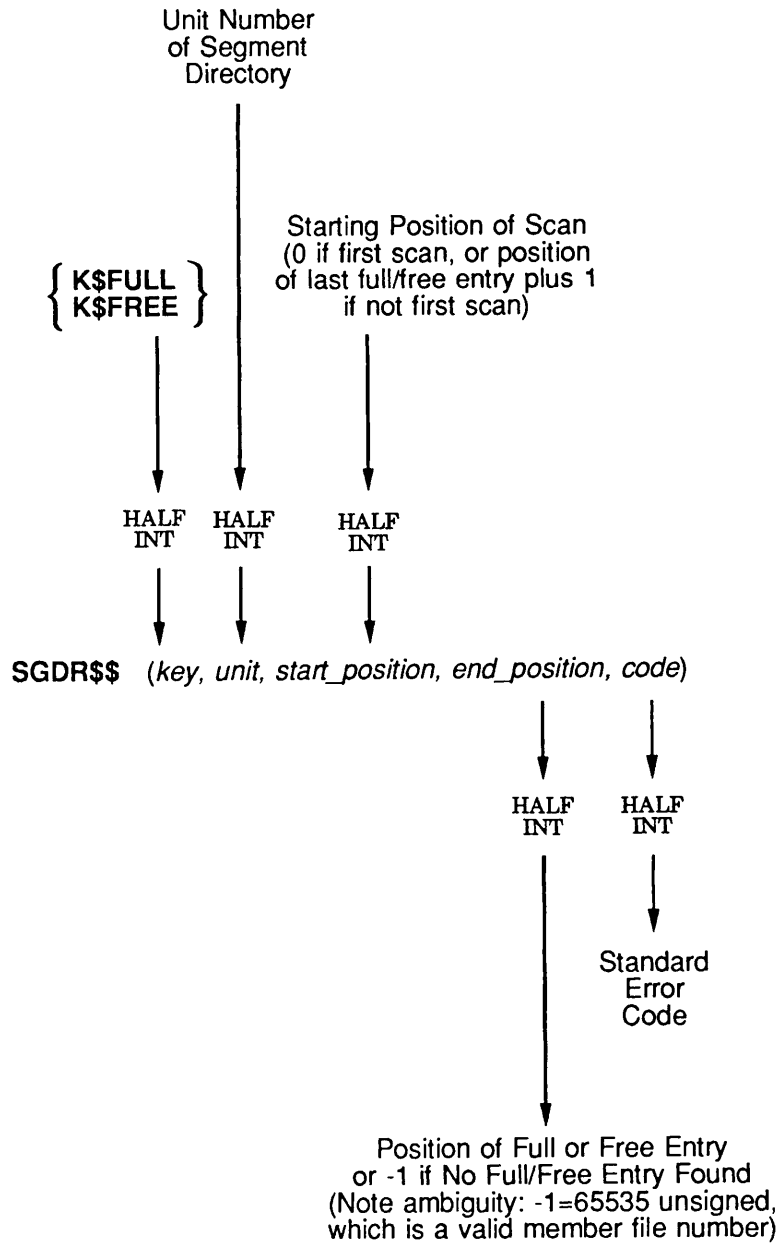
- The file unit of the open segment directory
- The starting position of the segment directory, also known as the first file number to be checked

The SGDR\$\$ subroutine searches the specified segment directory and returns to your program

- An error code indicating whether the operation was successful
- The ending position of the segment directory following the search, also known as the first full or free entry following the specified starting position

This section describes the input and output parameters that apply when calling SGDR\$\$ to search a segment directory, and then shows a sample call to SGDR\$\$. Figure 8–7 illustrates the calling sequence of SGDR\$\$ to scan a segment directory.

Scan Segment Directory for Next Full/Free Entry



Side Effects: The position of *unit* is left at *end-position* if desired entry is found; otherwise, *unit* is positioned to end of segment directory.

Q08.07.D10056.3LA

Figure 8-7. Calling Sequence of SGDR\$\$ to Scan a Segment Directory

Key: Set the *key* argument to one of the following two values:

<i>Value</i>	<i>Meaning</i>
K\$FULL	Find the first full entry
K\$FREE	Find the first free entry

Starting Position of the Segment Directory: Your program passes the starting position of the segment directory, which ranges from 0 to 65535 (–1 signed). The resulting position in the segment directory may or may not have a file present.

Error Code: An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was entirely successful. Otherwise, *code* is always positive. After a call to SGDR\$\$ to search a segment directory, *code* may have one of many values. *Advanced Programmer's Guide: Appendices and Master Index* of this series contains a comprehensive list of all standard file system error codes. Error codes specific to this operation are

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$EOF	1	End of file. The starting position is beyond the end of the segment directory. The segment directory is left positioned at the end of the directory. If a call to SRCH\$\$ with a key argument of K\$WRIT or K\$RDWR is performed within the segment directory at this point, a new member file is created, and the segment directory is automatically extended by one file entry.
E\$UNOP	3	Unit not open. The specified file unit is not open. This usually indicates a program error, although it can also be the result of the user exiting the program by means of a CONTROL–P sequence, typing CLOSE –ALL, and then typing START.

The Ending Position of the Segment Directory: The SGDR\$\$ subroutine returns the ending position of the segment directory resulting from the search operation. If the desired entry (full or free) was found, its position is returned in this variable. Otherwise, a value of –1 is returned.

Caution

A returned value of –1 in this variable corresponds to an unsigned value of 65535, and hence is not a reliable indicator of a search operation that failed to find the desired entry. When a value of –1 is returned in this field, have your program call SGDR\$\$ to position the segment directory to file number 65535. If SGDR\$\$ returns an end-of-file error code, or if it returns a file existence indicator of –1, then segment directory position 65535 does not exist. If SGDR\$\$ returns a file existence indicator of 0, then segment directory position 65535 exists, but there is no file with that number (the entry is free). If it returns a file existence indicator of 1, then file number 65535 exists (the entry is full).

Example: The following sample subroutine displays a list of all full entries in an open segment directory by using SGDR\$\$ to scan for existing entries.

```

SUBROUTINE LISTEM(SGUNIT, CODE)
  INTEGER*2 SGUNIT, CODE
C
$INSERT SYSCOM>ERRD.INS.FTN
$INSERT SYSCOM>KEYS.INS.FTN
C
  INTEGER*2 FILNBR, NXTNBR, MYTH
C
  FILNBR=0
C
10  CALL SGDR$$ (K$FULL, SGUNIT, FILNBR, NXTNBR, CODE)
    IF (CODE.NE.0) RETURN
C
    IF (NXTNBR.NE.-1) GO TO 20
C
C The returned file number is -1, or 65535 unsigned. Find out
C if file number 65535 is a myth.
C
    CALL SGDR$$ (K$SPOS, SGUNIT, -1, MYTH, CODE)
    IF (CODE.NE.E$EOF) GO TO 15
C
    CODE=0 /* Treat end-of-file as no more full entries.
    GO TO 100
C
15  IF (MYTH.LE.0) GO TO 100 /* No entry there.
C
C We have a file number in NXTNBR, print out the number with
C an optional header.
C
20  IF (FILNBR.NE.0) GO TO 30 /* First full entry?
C
    CALL TNOU('File numbers:',13) /* Yes, explain the list.
C
30  CALL TNOUA(' ',2) /* A little indentation.
    CALL TOVFD$(NXTNBR) /* Number may be negative, of course.
    CALL TNOU(0,0) /* End of line.
C
    IF (NXTNBR.EQ.-1) GO TO 100 /* Definitely last entry?
    FILNBR=NXTNBR+1 /* No, search for next entry.
    GO TO 10 /* Thanks, Debbie...
C
100 IF (FILNBR.NE.0) RETURN /* Finished with listing.
C
    CALL TNOU('No files.',9)
    RETURN
C
END

```

File Directories

Relating primarily to the manipulation of file directories themselves, this section describes

- How to create a file directory
- How to open a file directory
- How to scan a file directory
- How to determine a new filename

The subroutines most often used when accessing file directories are

<i>Subroutine</i>	<i>Use</i>
DIR\$CR	Creates a directory. Your program passes the pathname of the directory to DIR\$CR along with a structure defining the initial state of the directory. DIR\$CR creates the specified directory. Your program may then populate the directory with new file system objects.
SRSFX\$	Accepts a pathname and calls SRCH\$\$ to manipulate the directory according to the specified key.
SRCH\$\$	Accepts a filename, and searches for the directory in the current directory. The SRSFX\$ subroutine calls SRCH\$\$ after it attaches to the directory specified by the supplied pathname. SRCH\$\$ can open, close, change access on, or verify the existence of the directory.
DIR\$RD	Reads the next entry from an open directory, and returns a structure that describes the name of the entry and its attributes. Your program can use DIR\$RD to read successive entries in a directory.
ENT\$RD	Reads a particular entry from an open directory, and returns a structure that describes the name of the entry and its attributes. Your program can use ENT\$RD to read the attributes of a particular entry in a directory by specifying the name of the entry.

Creating a File Directory

Your program creates file directories by using the DIR\$CR subroutine. Your program supplies the pathname of the directory to be created along with control information on the type of directory to be created and on its attributes. Once created, the directory contains no file system objects; your program may then create new file system objects in the newly created directory.

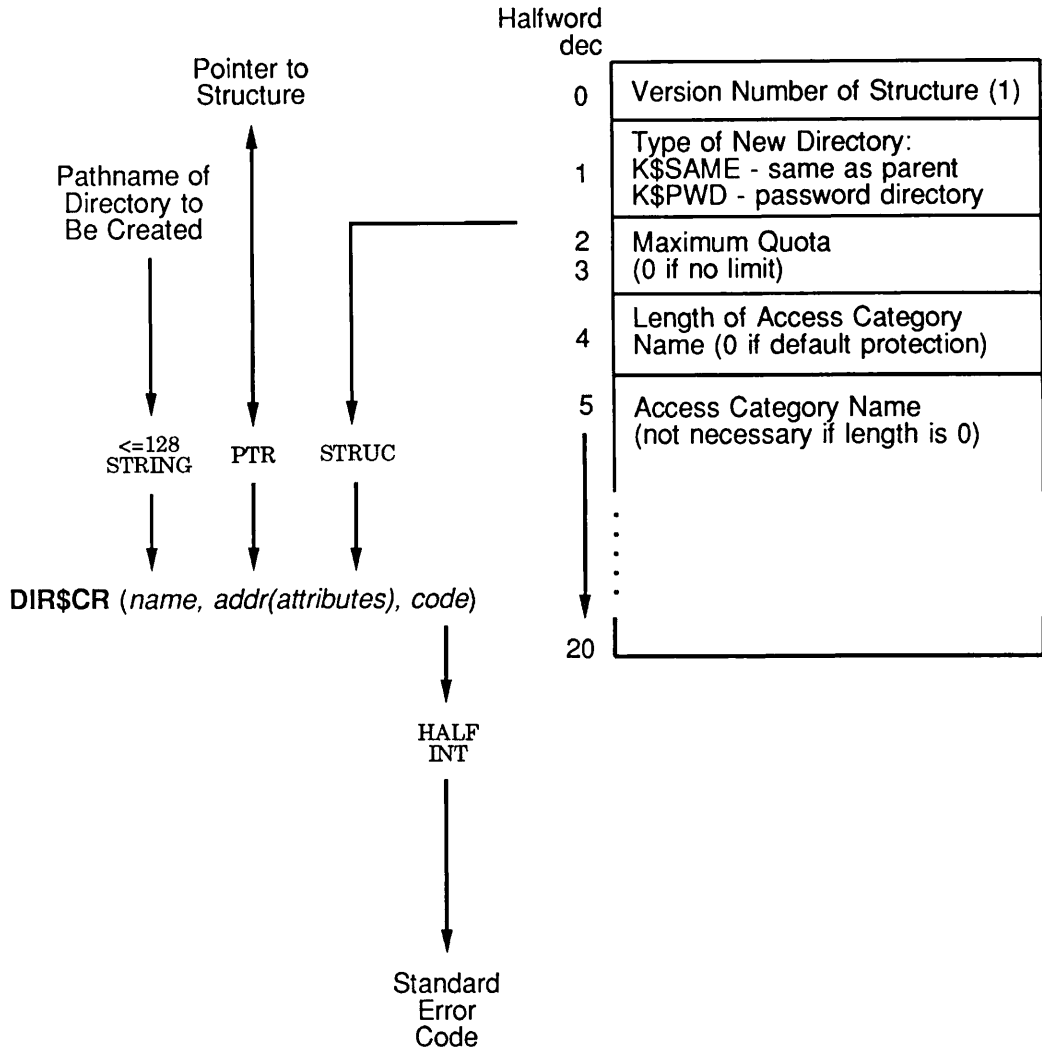
When your program calls the DIR\$CR subroutine, it provides

- The pathname of the directory to be created
- The attributes of the directory

The DIR\$CR subroutine attempts to create the directory and returns to your program an error code indicating whether the operation was successful.

This section describes the input and output parameters to use when calling DIR\$CR, and then shows a sample call to DIR\$CR. Figure 8–8 illustrates the calling sequence of DIR\$CR.

Create File Directory



Side Effects: Resets current directory if *name* contains a > symbol; otherwise, new directory created in current directory.

Q08.D8.D10056.3LA

Figure 8-8. Calling Sequence of DIR\$CR

Attributes of the Directory: Your program constructs a structure that contains the attributes of the directory to be created, and passes a pointer to this structure to DIR\$CR. This structure describes

- Whether the directory is to be the same type (ACL or password) as its parent, or is to be made a password directory.
- The maximum quota of the directory.
- The access category that is to protect the directory.

Normally, you set the directory type to the same type as its parent, the maximum quota to 0 (meaning no quota), and the access category to null (meaning default protection).

Error Code: An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was entirely successful. Otherwise, *code* is always positive. After a call to DIR\$CR to create a file directory, *code* may have one of many values. *Advanced Programmer's Guide: Appendices and Master Index* of this series contains a comprehensive list of all standard file system error codes. Error codes specific to this operation follow.

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
ESBPAR	6	Bad parameter. The maximum quota for the directory is a negative number.
ESDKFL	9	The disk is full. The new directory cannot be created.
ESNRIT	10	Insufficient access rights. The user does not have sufficient access to create the specified directory. This error code may also indicate a problem in attaching to the directory specified by the pathname argument of the calling sequence. In this case, the user does not have Use access to at least one directory in the pathname.
ESFNTF	15	Not found. There is a problem in attaching to the directory specified by the pathname argument of the calling sequence. In this case, at least one directory in the pathname does not exist.
ESEXST	18	Already exists. Another file system object already exists with the name of the new directory. The existing object may be a file directory, or some other file system object.
ESMXQB	143	Maximum quota exceeded. The new directory cannot be created.
ESNOQD	144	Not a quota disk. The disk on which the directory is to be created is not a quota disk, but the supplied structure indicates that a maximum quota is to be imposed.

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$PNAC	148	Parent not an ACL directory. The parent directory of the new directory is not an ACL directory, but the supplied structure indicates that the new directory is to be protected by an access category.
E\$ACNF	155	Access category not found. The access category to be used to protect the newly created directory does not exist in the parent directory.
E\$BVER	158	Bad version. The version number of the supplied structure containing the attributes of the directory is not 1. The calling program must initialize this number to 1 before calling DIR\$CR.
E\$DTNS	173	Date and time not set yet. The supplied structure indicates that a quota is to be placed on the new directory, but quotas cannot be imposed unless the system date and time are set.
E\$NFAS	189	Top-level directory not found or inaccessible. The first directory name supplied in the pathname could not be located on any of the system disks.

Examples: The following PL/I code illustrates a sample call to DIR\$CR to create a new directory named FRODO in the HOBBIT directory. The newly created directory is of the same type as the HOBBIT directory, is a non-quota directory, and has no access category protecting it (it is protected by the access on HOBBIT).

```

struc.version=1;
struc.dir_type=k$same;
struc.max_quota=0;
struc.acc_cat='';

call dir$cr('HOBBIT>FRODO',addr(struc),code);
if code^=0 then call
errpr$(k$irtn,code,'HOBBIT>FRODO',12,
'MYPROGRAM',9);

```

Opening a File Directory

Your program must open a directory before it may read entries in the directory. To open the directory, your program uses the SRSFX\$ or SRCH\$\$ subroutine. When calling these subroutines, your program provides

- The name of the directory to be opened
- A key that specifies how the directory is to be opened

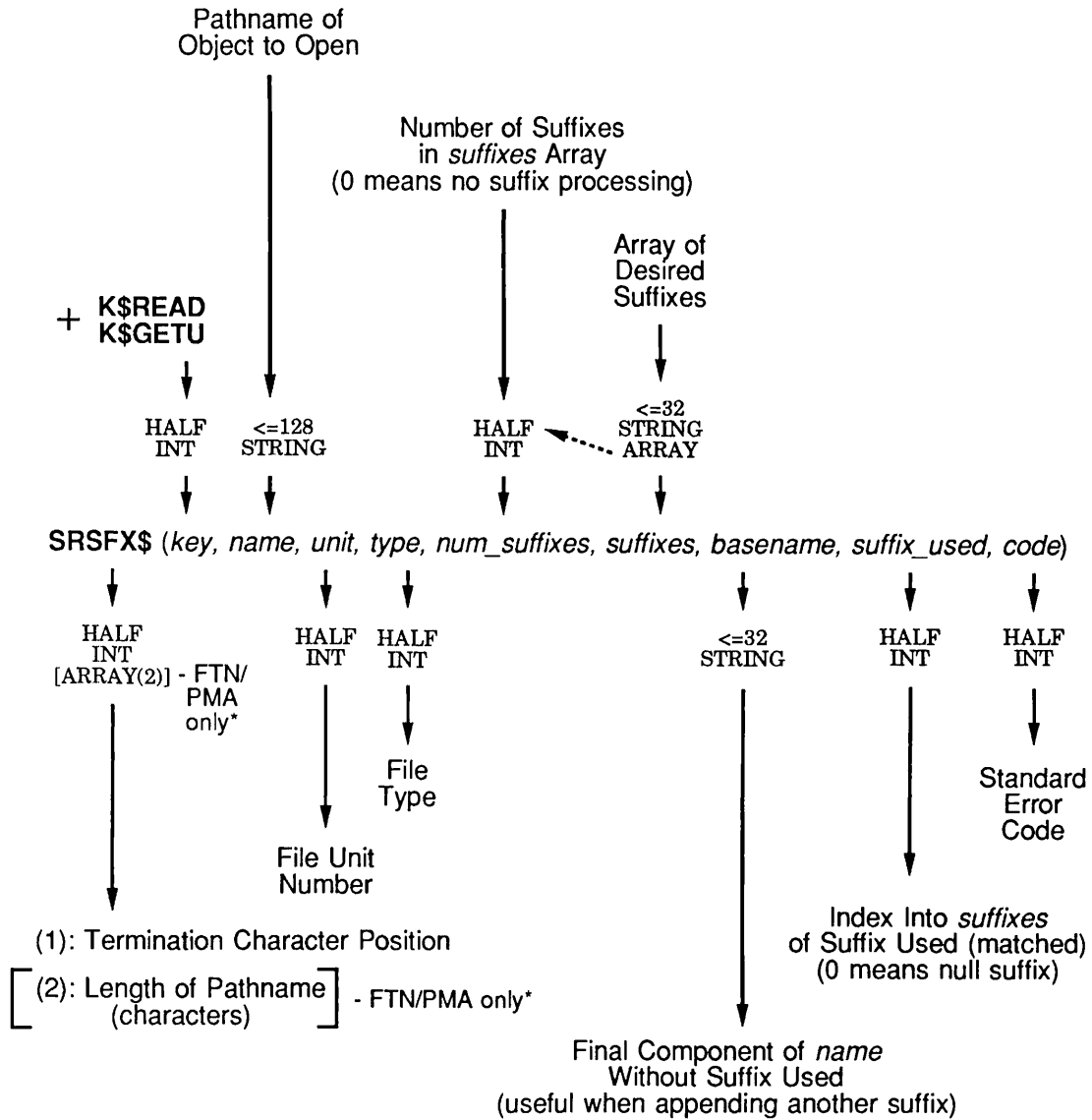
The SRSFX\$ or SRCH\$\$ subroutine attempts to open the specified directory and return to your program

- An error code indicating whether the operation was successful
- A file unit number that identifies the open directory; your program uses this number when reading directory entries in an open directory
- The file type, indicating the type of file just opened (including SAM, DAM, SEGSAM, SEGDAM, and Directory)

Additional information returned by SRSFX\$ is not relevant to this discussion.

This section describes the input and output parameters that apply when calling SRSFX\$ and SRCH\$\$, and then shows a sample call to SRCH\$\$. Figure 8–9 illustrates the calling sequence of SRSFX\$ to open a directory; Figure 8–10 illustrates the SRCH\$\$ calling sequence.

Open a File Directory, With Possible Suffix



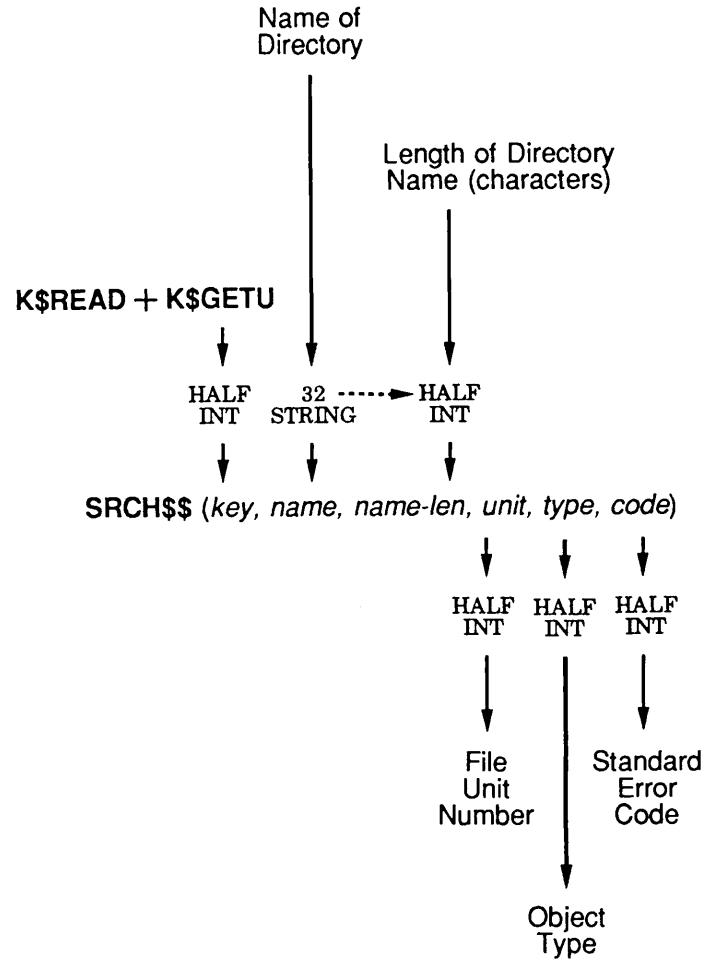
Side Effects: May reset current attach point.

* Function value is returned in L-register; typically, you need only to declare as HALF INT, because first datum is all you need and is in A-register. Otherwise, you must declare it as FULL INT to make it work.

Q08.09.D10056.3LA

Figure 8-9. Calling Sequence of SRSFX\$ to Open a File Directory

Open File Directory by Object Name



Q08.10.10056.3LA

Figure 8-10. Calling Sequence of SRCH\$ to Open a File Directory

The Error Code: An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was entirely successful. Otherwise, *code* is always positive. After a call to SRSFX\$ or SRCH\$\$ to open a directory, *code* may have one of many values. *Advanced Programmer's Guide: Appendices and Master Index* of this series contains a comprehensive list of all standard file system error codes. The error code specific to this operation is

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$NRIT	10	Insufficient access rights. This means that the user running your program does not have List access to the directory. For calls to SRSFX\$, this error code may indicate a problem attaching to the directory specified by the pathname argument of the calling sequence. In this case, the user running your program does not have Use access to at least one directory in the pathname.

File Type: The returned file type is valid only when the returned error code is 0 and the directory is actually opened. The file type is one of the following five values:

<i>Value</i>	<i>Meaning</i>
0	A SAM file has been opened. Use RDLIN\$, WTLIN\$, PRWF\$\$, and similar subroutines to read or write it. (See Chapter 7, Text Storage and Retrieval, for information on how to do this.)
1	A DAM file has been opened. Use RDLIN\$, WTLIN\$, PRWF\$\$, and similar subroutines to read or write it. (See Chapter 7, Text Storage and Retrieval, for information on how to do this.)
2	A SAM segment directory (SEGSAM) has been opened. Use SGDR\$\$ to operate on members of this segment directory. (See the section Segment Directories, above, for information on how to do this.)
3	A DAM segment directory (SEGDAM) has been opened. Use SGDR\$\$ to operate on members of this segment directory. (See the section Segment Directories, above, for information on how to do this.)
4	A directory has been opened. Use DIR\$LS, DIR\$RD, DIR\$SE, ENT\$RD, and RDEN\$\$ to read information on files in this directory.

Example: The following example shows how a FORTRAN program would open the directory MYDIR in the current directory for reading:


```

CALL SRCH$$ (K$READ+K$GETU, 'MYDIR', 5, UNIT, TYPE, CODE)
IF (CODE.NE.0) GO TO 1000
.
.
.
1000 CALL ERRPR$ (K$IRTN, CODE, 'MYDIR', 5, 'MYPROGRAM', 9)
RETURN

```

How to Scan a File Directory

Once your program opens a directory, it may scan that directory. Scanning a directory consists of reading file system object entries. Your program may read entries sequentially, that is, in the order in which they appear in the directory, or may read particular entries by name. This section describes how to read a directory sequentially, one entry at a time, using the DIR\$RD subroutine. To read sequential entries several entries at a time, see the description of the DIR\$SE subroutine in *Subroutines Reference II: File System*. To read directory entries by name, see Chapter 10, File Attributes.

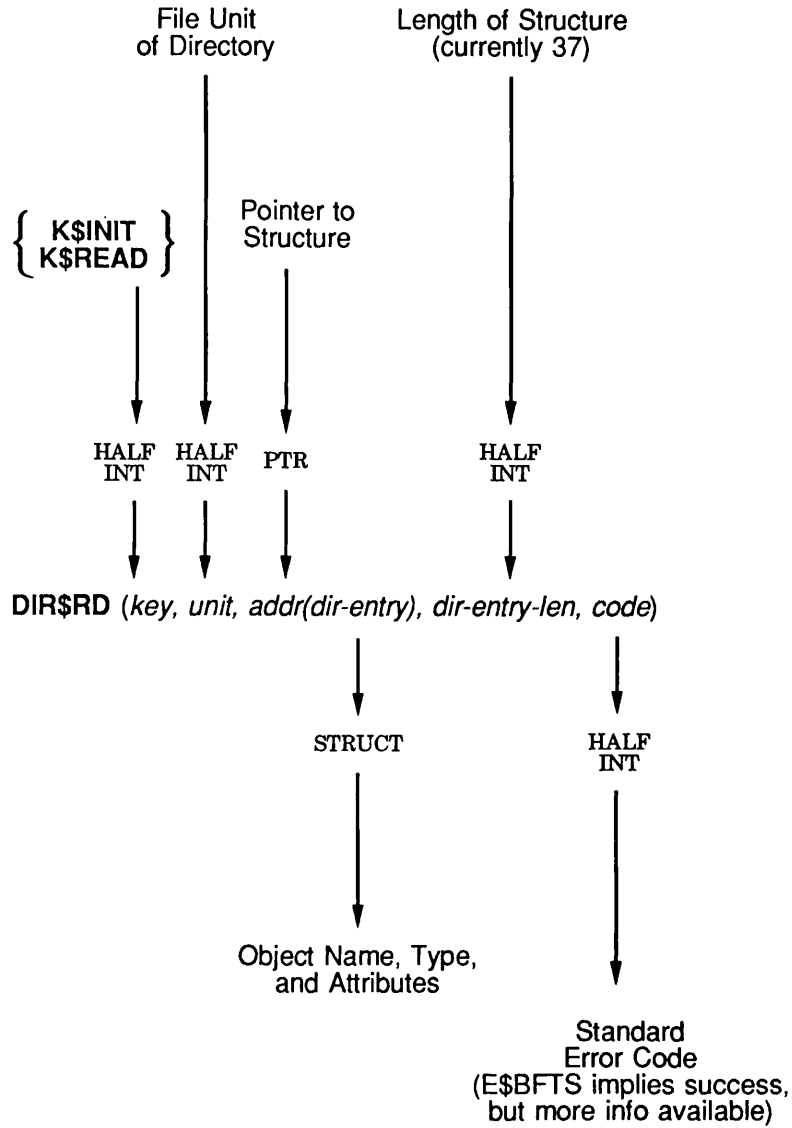
After opening the directory, your program calls DIR\$RD providing

- The file unit of the open directory
- A key that specifies the operation to be performed
- A pointer to a structure into which the entry information is to be stored
- The size of the storage structure

The DIR\$RD subroutine finds the next sequential entry in the specified directory and returns to your program an error code indicating whether the operation was successful.

This section describes the input and output parameters to specify when you call DIR\$RD to scan a directory, and then shows a sample call to DIR\$RD. Figure 8–11 illustrates the calling sequence of DIR\$RD.

Read Next Entry in File Directory



Side Effects: Repositions *unit*.

Q08.11.D10056.3LA

Figure 8-11. Calling Sequence of DIR\$RD

Key: Your program sets the *key* argument to one of the following values:

<i>Value</i>	<i>Meaning</i>
K\$READ	Read the next entry
K\$INIT	Reset to the beginning of the directory

Normally, your program passes the K\$READ value for *key*. Your program uses the K\$INIT value only if the open directory is to be read again from the beginning, as in a two-pass directory scanning program.

Pointer to a Structure: Your program provides a structure that DIR\$RD fills in with information on the next entry in the directory. Your program passes a pointer to this structure to DIR\$RD. Assuming DIR\$RD finds an entry, it fills the structure with information such as the filename, the file type, and other information on the file. See the description of the *dir_entry* structure in Chapter 10, File Attributes, for details.

Error Code: An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was entirely successful. Otherwise, *code* is always positive. After a call to DIR\$RD to scan a directory, *code* may have one of many values. *Advanced Programmer's Guide: Appendices and Master Index* of this series contains a comprehensive list of all standard file system error codes. Error codes specific to this operation are

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$EOF	1	End of file. No more directory entries are present.
E\$UNOP	3	Unit not open. The specified file unit is not open. This usually indicates a program error, although it can also be the result of the user exiting the program by means of a CONTROL-P sequence, typing CLOSE -ALL, and then typing START.
E\$BFTS	35	Buffer too small. The supplied structure is too small to hold the information. Unlike the other error codes, this error code indicates that the operation succeeded, but that only some of the available information (as much as the calling program has asked for) has been returned in the structure.

Example: The following subroutine displays the names of all the files in an open file directory:

```

list_file_names: proc(unit,code);

dcl unit fixed bin(15),/* File unit directory is open on. */
     code fixed bin(15); /* Standard f/s error code. */

/* Other declarations omitted. */

/* This subroutine assumes that the specified file unit is
   already positioned at the beginning of the directory. It
   therefore does not call DIR$RD with the K$INIT key. */

first='1'b;

do until(code^=0);
  call dir$rd(k$read,unit,addr(dir_entry),31,code);
  if code=0
    then do;
      if first then call tnou('File names:',11)
      first='0'b;
      call tnoua(' ',2);
      call tnou(dir_entry.name,length(trim(dir_entry.name,
        '01'b))); /* Don't output trailing blanks. */
      end; /* if code=0 */
    end; /* do until(code^=0) */
  if code=e$eof
    then do;
      if first then call tnou('No files.',9);
      code=0;
      end; /* if code=e$eof */
    end; /* list_file_names: proc */

```

Reading and Writing Data Files

A **data file** is a file containing data that does not logically break down into single 8-bit bytes. For example, a file that contains a list of employee records that contain some single-bit data is a data file, rather than a text file.

In general, PRIMOS does not distinguish between text and data files. PRIMOS does provide a simple interface for variable-length record text files (the RDLIN\$ and WTLIN\$ subroutines); this interface is described in Chapter 7, Text Storage and Retrieval. The interface for data files is precisely the same interface used for fixed-length record files, described in Chapter 7, Text Storage and Retrieval.

Many application programs store data files in segment directories. The manipulation of data files in segment directories is described in the section entitled Segment Directories, found earlier in this chapter. Whether a data file is a member of a file directory or segment directory, however, does not affect how

it is read, written, extended, and truncated. These operations are very similar to the operations performed on fixed-length record text files.

There are several important things to remember when you are designing a program that reads and writes data files:

- There is no record length or blocking factor that PRIMOS is aware of. If your program writes more or less data than originally specified in the design specification for your program, PRIMOS does not know to truncate or extend the data.
- Because there is no implicit record length, your program must satisfy its own random-access position calculation requirements. PRIMOS provides the ability to position a file only to a specified halfword location.
- PRIMOS allows data files to be read and written in any order. PRIMOS imposes no sequential ordering, although such ordering is typically the default.
- The only way your program may extend the length of a data file is by writing new data starting at the end-of-file location; PRIMOS automatically extends the end-of-file location as your program writes the file.
- PRIMOS allows your program to use more than one file unit at a time to access a single file, assuming the read/write lock restrictions are satisfied. You can use this capability to improve the performance of your program in certain cases.

For example, suppose your program needs to read data record indexes at the beginning of a large file, whereas the data records themselves are scattered throughout the file. If your program uses two file units to access the file simultaneously, your program can position one file unit at the beginning of the file to access the indexing information rapidly, and use the other file unit to retrieve and store the data records themselves.

- In most cases, data files should be created as DAM files.

Questions and Answers About Data Files

This section answers some typical questions about data storage and retrieval. See Chapter 7, *Text Storage and Retrieval*, for questions and answers about text storage and retrieval, including opening files.

- Explain the relationship between SRSFX\$ and segment directories.



SRSFX\$ is the only file system subroutine that allows references to files within segment directories. A detailed description of accessing files within segment directories is provided earlier in this chapter.

SRSFX\$ is a high-level interface for opening a segment directory member. You typically use SRSFX\$ for opening one or two particular members of a segment directory, because it provides a simpler interface than the more complex method recommended in this chapter.

- Aren't there more subroutines I can use to do things like change filenames and numbers, and determine pathnames?

Yes, there are. Most application programs do not need to use these subroutines during most of their development process. However, functions such as changing the number (position) of a segment directory member are sometimes useful when you construct administrative tools for the application. Determining the full pathname of a file system object is also useful.

For information on changing filenames and numbers, see the descriptions of the SGDR\$\$ and CNAM\$\$ subroutines in *Subroutines Reference II: File System*. Similarly, the GPATH\$ subroutine is useful for determining the pathname of an open file system object or an attach point. In fact, one of the examples in Chapter 7, Text Storage and Retrieval, used GPATH\$ for a typical situation in which the full pathname of an open file unit is quite useful.

Access Control Lists (ACLs)

9



This chapter discusses

- Subroutines used to manipulate ACLs
- How programs should parse an ACL
- Typical questions and answers about ACLs

The reader should be familiar with Access Control Lists (ACLs), as described in the *PRIMOS User's Guide*.

Subroutines That Manipulate ACLs

Subroutines that manipulate ACLs are fully described in *Subroutines Reference II: File System*. They are summarized briefly here.

When using these subroutines, you may wish to think of access categories as file system objects that have specific ACLs set on them. An access category centralizes access for several files and directories in one ACL represented by that access category. In a sense, the access category itself is a placeholder file system object with a specific ACL set on it. Envisioning access categories in this fashion is particularly useful when using subroutines such as AC\$SET, AC\$CHG, and AC\$LST.

Setting Access on Files and Directories

You can set any of the following three accesses:

- Default
- Specific
- Category

Setting Default Access: To set access for a file or directory to the default access, use the AC\$DFT subroutine. The default access for a file system object



comes from the ACL for the parent directory of that object. You cannot set the MFD for a disk partition to default access. Figure 9-1 illustrates the calling sequence for the AC\$DFT subroutine.

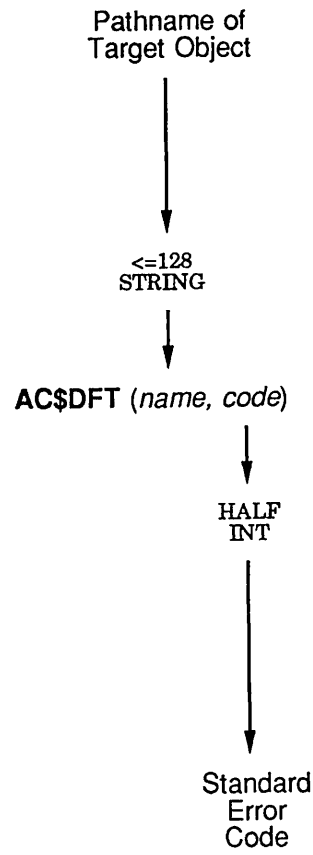
Setting Specific Access: To set a specific ACL on a file or directory, use the AC\$SET subroutine. Your program provides a structure describing the desired access. Figure 9-2 illustrates the calling sequence of the AC\$SET subroutine.

Setting Category Access: To set a category ACL on a file or directory, use the AC\$CAT subroutine. Your program passes the name of the access category that is to protect the object. The access category must already exist in the same directory as the object being protected. Figure 9-3 illustrates the calling sequence of the AC\$CAT subroutine.

Creating Access Categories

To create an access category, use the AC\$SET subroutine. Your program passes the name of the access category to be created and provides a structure describing the desired access. Figure 9-2 illustrates the calling sequence of the AC\$SET subroutine.

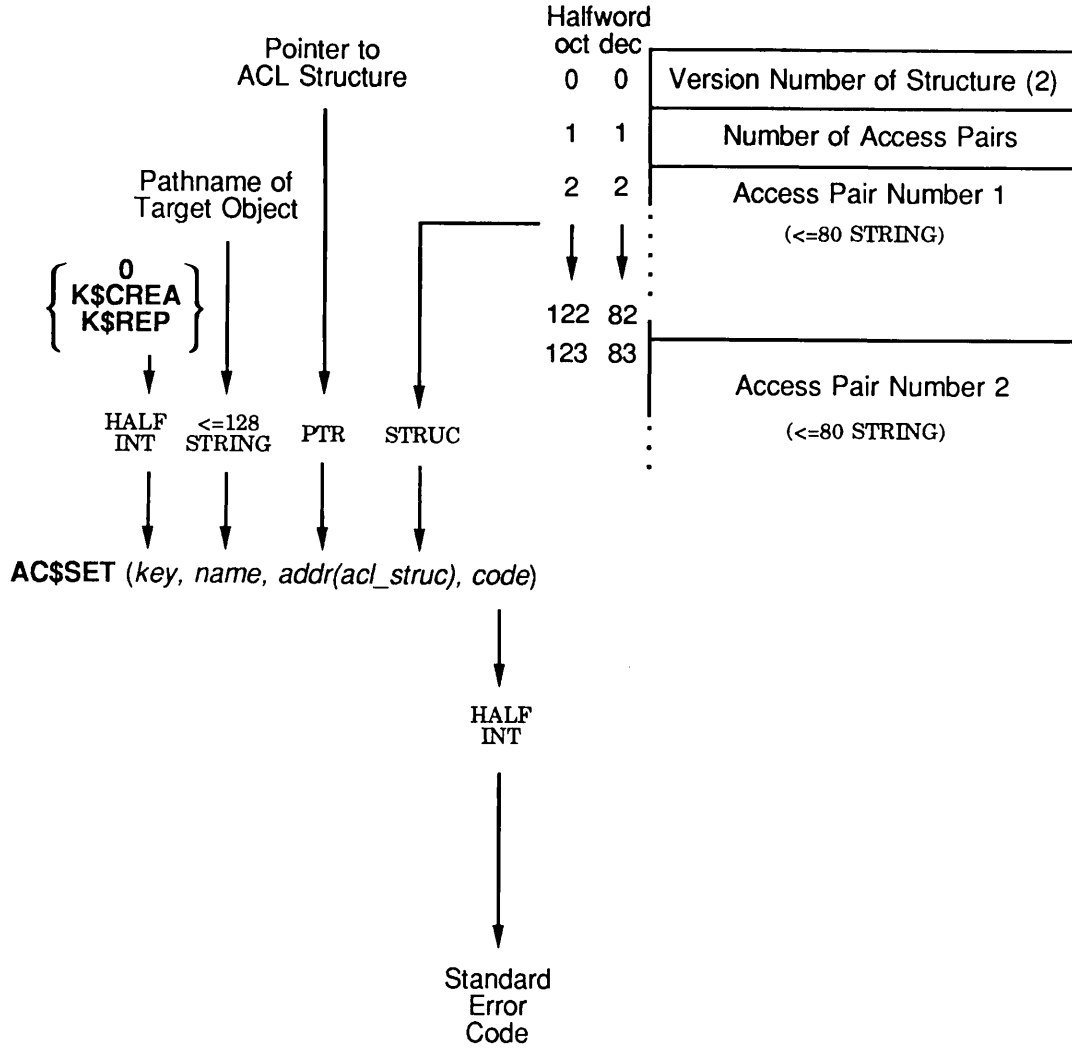
Protect Object With Default ACL



Q09.01.D10056.3LA

Figure 9-1. Calling Sequence of AC\$DFT

Create or Replace Specific ACL of Object, or Replace ACL of Access Category

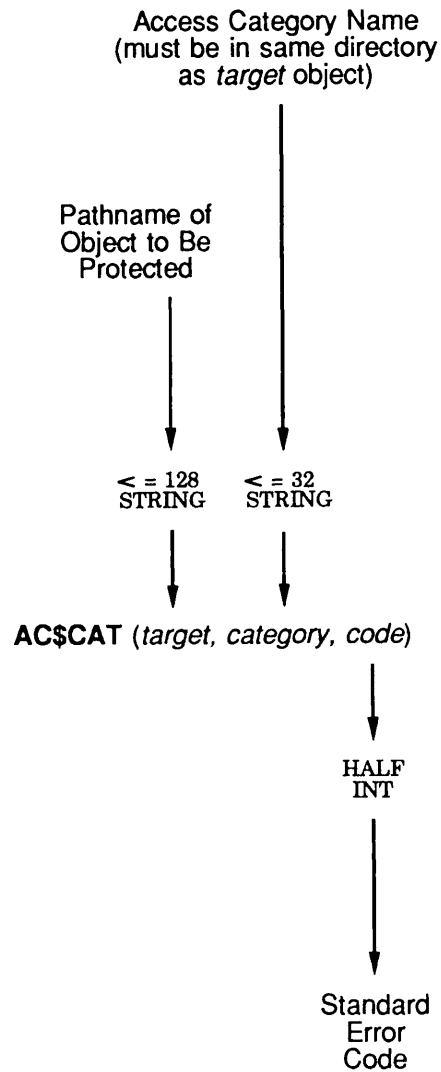


Side Effects: May reset current attach point.

Q09.02.D10056.3LA

Figure 9-2. Calling Sequence of AC\$SET

Protect Object With Access Category



Side Effects: May reset current attach point.

Q09.D3.D10056.3LA

Figure 9-3. Calling Sequence of AC\$CAT

Changing Access to a File System Object

When changing access to a file system object, the object must already be protected by a specific ACL or a category ACL. To change the contents of an access category, you treat the access category as a file system object protected by a specific ACL.

Changing Access for a Specific-protected Object: If you wish to change the access of a file or directory that is protected by an access category, you have two choices:

- Change the ACL of the access category
- Set a new specific ACL on the file or directory

To change the ACL of an existing file system object protected by a specific ACL, including an access category, use either the AC\$CHG or AC\$SET subroutine, depending on the nature of the change. To modify the contents of an existing ACL, use AC\$CHG. To replace the existing ACL with an entirely new ACL, use AC\$SET. In both cases, your program provides a structure describing the desired access.

Figure 9-4 illustrates the calling sequence of the AC\$CHG subroutine. The calling sequence of the AC\$SET subroutine is illustrated in Figure 9-2.

When changing the ACL of an access category, keep in mind that the access for all files and directories protected by the access category also changes. To change the ACL of an access category, treat the access category as a file system object protected by a specific ACL, as described earlier in this section.

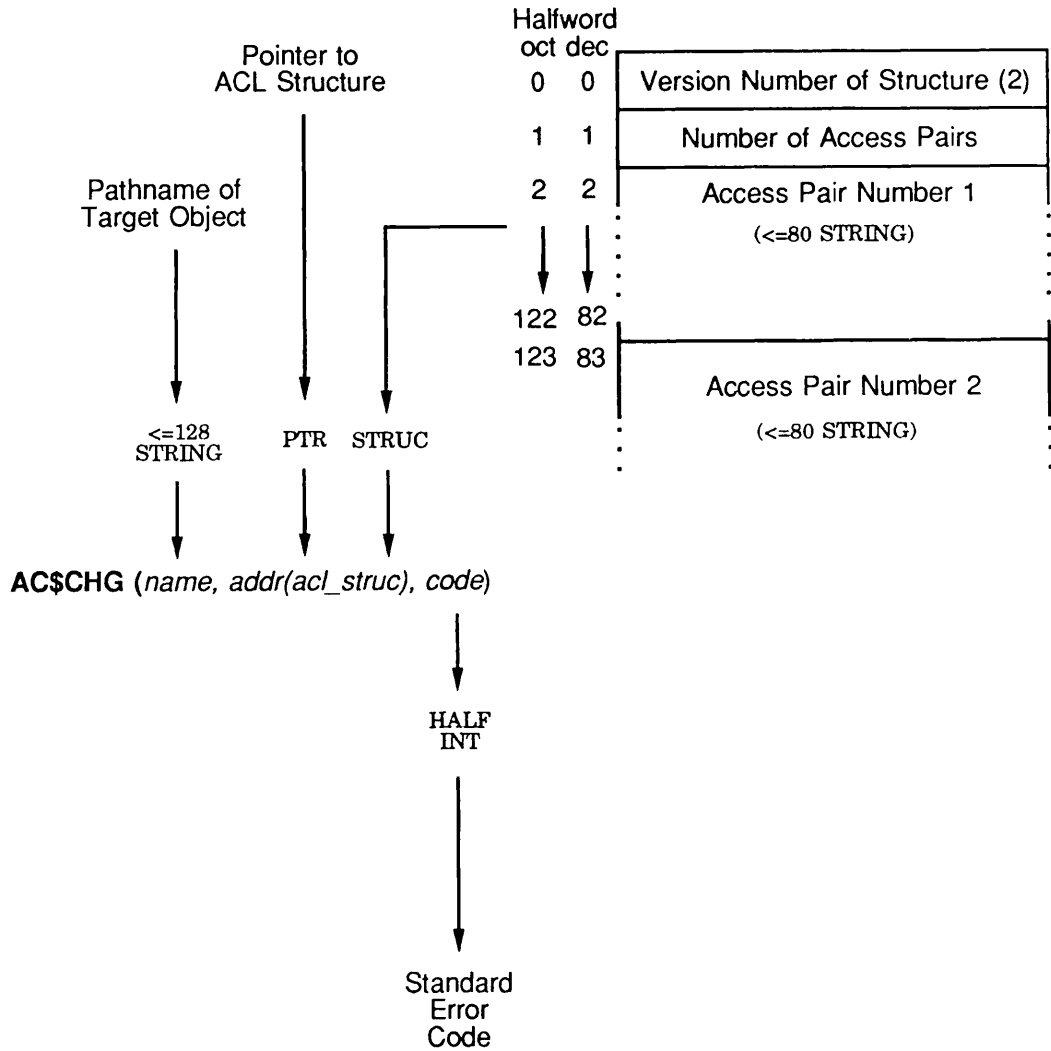
To set a new specific ACL on a file or directory that is currently protected by a category ACL, use the AC\$LIK subroutine. This creates a specific ACL that protects the file or directory in the same way as the category ACL. Now, use AC\$SET or AC\$CHG on the file or directory to change the specific ACL for the object.

Figure 9-5 illustrates the calling sequence of the AC\$LIK subroutine.

Setting the Access for an Object to That of Another Object

To set the access for an object to that of another object, use the AC\$LIK subroutine. The access for the target object is copied from the ACL protecting the model object, whether via default, specific, or category access. A specific ACL with this access is then set on the target object by AC\$LIK. The target and model objects need not reside in the same lower-level directory, as the ACL is copied by value, rather than by reference.

Change Protection of Object (Specifically Protected),
or Change Access Control List of Access Category

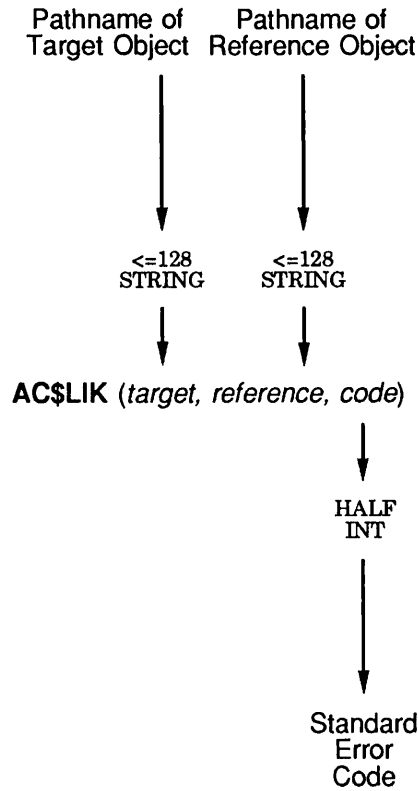


Side Effects: May reset current attach point.

Q09.D4.D10056.3LA

Figure 9-4. Calling Sequence of AC\$CHG

Protect Object With Specific ACL According to ACL That Protects Reference Object



Side Effects: May reset current attach point.

Q09.05.D10056.3LA

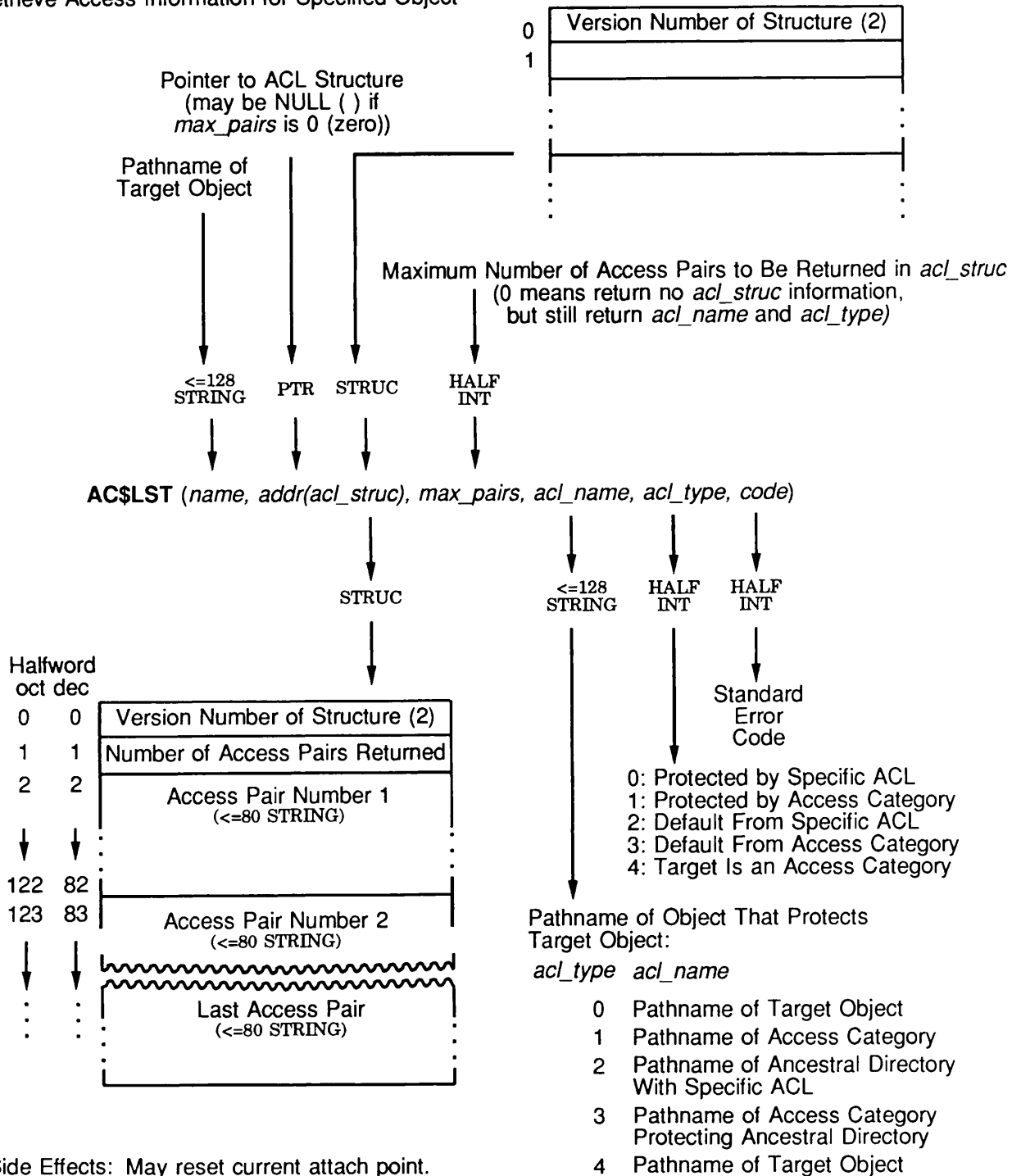
Figure 9-5. Calling Sequence of AC\$LIK

Figure 9-5 illustrates the calling sequence of the AC\$LIK subroutine.

Reading the Access for an Object

To read the ACL protecting a file system object, use the AC\$LST subroutine. Your program provides a structure describing the ACL that is to be filled in by AC\$LST. Your program then analyzes the returned structure to determine the access. Figure 9-6 illustrates the calling sequence of the AC\$LST subroutine.

Retrieve Access Information for Specified Object



Side Effects: May reset current attach point.

Q09.06.D10056.31A

Figure 9-6. Calling Sequence of AC\$LST

How Programs Should Parse an ACL

This section describes how to parse an ACL on an existing file system object. The access string information also applies to constructing an ACL to be placed on a file system object.

When the AC\$*LST* subroutine is used to read an ACL, a list of access pairs is returned. (The calling sequence of AC\$*LST* is shown in Figure 9–6.) Each access pair has the following format:

id:access

Both *id* and *access* are at least one character long, separated by a colon (:). If you are constructing an ACL to be passed to AC\$*CHG*, *access* may be the null string to indicate deletion of the access pair for the specified *id*.

The *id* portion of the access pair is either a user ID, a group name, or the character string \$*REST*. A group name begins with a period (.), whereas a user ID does not.

The *access* portion of the access pair can be the character string *NONE* (indicating no access rights), a character string listing individual access rights, or the string *ALL* (indicating all access rights— OPDALURWX at Rev. 21.0 and after). Note that AC\$*LST* never returns a string representing all of the supported access rights; it is translated to *ALL*. Because *ALL* may represent a different set of rights at different revs, it is recommended that access rights be checked individually using the CALAC\$ subroutine. CALAC\$ takes a list of accesses you supply as input and checks against them. CALAC\$ is described in the *Subroutines Reference II: File System*.

You may design your program so that it ignores unrecognized characters.

Questions and Answers About ACLs

- Can ACL operations result in disk–full or quota–exceeded errors?

Yes. Even though specific ACLs do not appear as separate files, they do take up room on the disk when they are created or modified. Therefore, it is possible to exceed the capacity of the directory or the disk when

- Placing a specific ACL on an object that does not already have one
- Creating a new access category
- Updating the specific ACL of an object
- Updating the ACL of an existing access category

Changing the protection of an object from one category ACL to another (existing) category ACL never results in a disk-full or quota-exceeded error.

- Is there a limit to how many access pairs can be put in a specific ACL or access category?

Yes, there are two distinct limitations on access control lists:

- Limit on the number of access pairs passed in *acl_struct*
- Limit on the maximum size of a physical ACL on the disk

The first limit is the maximum number of access pairs accepted by PRIMOS AC\$ subroutines. This limit, named *max_acl_entries*, is currently 32. If your program attempts to pass more than 32 access pairs to a subroutine such as AC\$SET and AC\$CHG, the subroutine returns the error code E\$PBAR (Bad parameter) to your program. (The AC\$LST subroutine places no limit on the *max_pairs* argument, because it never returns more than 32 access pairs.)

The second limit is more complex. The limit on the number of halfwords that a physical representation of an ACL may take up on the disk is a PRIMOS parameter named *max_ent_len*, which is currently 255. An ACL with no access pairs (not counting the \$REST access pair, which is present in every ACL, even when not specified) takes up a minimum number of halfwords; named *base_entry_len*, this value is currently 11. Finally, each access pair in an ACL (excluding the \$REST access pair) takes up 5 halfwords plus the number of halfwords needed to contain the *id* portion of the access pair (not counting trailing blanks).

Therefore, the second limit can be defined as follows:

$$11 + [\text{for each access pair } (5 + (\text{length}(\text{id}(n))+1)/2)] \leq 255$$

If the second limit is exceeded, the AC\$ subroutine called by your program returns the error code E\$ACBG (ACL too big) to your program, and does not perform the requested operation.

File Attributes

10



This chapter first describes how to read the attributes of a file system object; then it describes how to set each attribute. Finally, a question-and-answer section is provided.

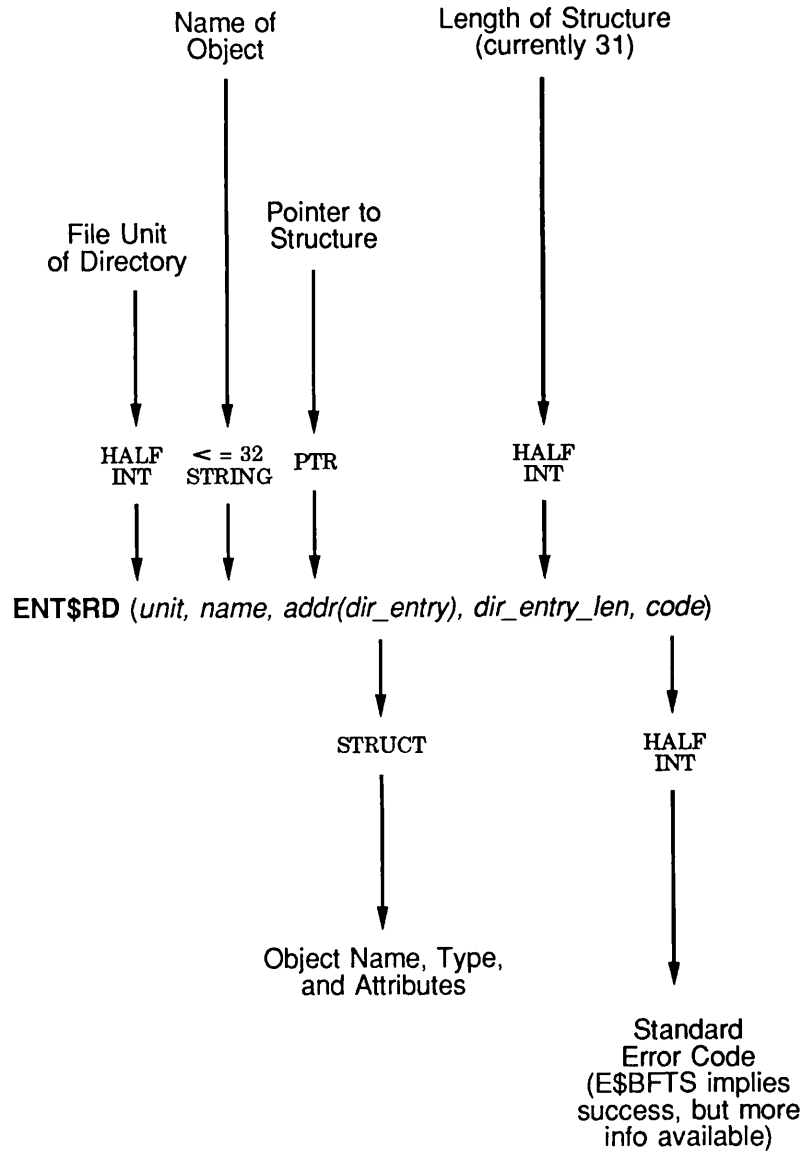
How to Read the File Attributes of an Object

To read the file attributes of a specific file system object, your program first opens the parent directory of that object for reading. See Chapter 8, Data Storage and Retrieval, for a description of how to open a directory for reading.

Then, your program calls the ENT\$RD subroutine to read the attributes. Remember that your program should close the parent directory when finished with it. Figure 10-1 illustrates the calling sequence of ENT\$RD. Chapter 8, Data Storage and Retrieval, describes DIR\$RD, a similar subroutine that is used to scan a directory sequentially for entries and read their attributes.

The structure returned by the ENT\$RD subroutine (*dir_entry*) contains the objectname, the file type of the object, and all other attributes of the object. The format of this structure is shown in Figure 10-2.

Read Particular Named Entry in File Directory



Side Effects: Repositions *unit*.

Q10.01D10056.3LA

Figure 10-1. Calling Sequence of ENT\$RD

Halfword	Offset	Bit #																
oct	dec	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
0	0	directory entry type: 3 for access category; 2 for other objects						length of structure (halfwords) -currently 31										
1	1	name of object, 32 characters, blank-padded																
20	16	reserved bits						(^acl) owner rights delete/truncate write read			reserved bits			(^acl) non-owner rights delete/truncate write read				
21	17	reserved bits						reserved bits						reserved bits				
22	18	has own acl	reserved bits						reserved bits									
23	19	long rat header	dumped bit	PRIMOS II modified	special file	read/write lock	reserved bits			file type								
24	20	year last modified minus 1900						month last modified (January is month 1)			date last modified (1-31)							
25	21	time last modified (seconds since midnight divided by 4)																
26	22	logical_type																
27	23	hash_thread																
30	24	truncated	reserved bits															
31	25	year last backed-up minus 1900						month last backed-up (January is month 1)			date last backed-up (1-31)							
32	26	time last backed-up (seconds since midnight divided by 4)																
33	27	year created minus 1900						month created (January is month 1)			date created (1-31)							
34	28	time created (seconds since midnight divided by 4)																
35	29	year last accessed minus 1900						month last accessed (January is month 1)			date last accessed (1-31)							
36	30	time last accessed (seconds since midnight divided by 4)																

Q10.02.D10056.3LA

Figure 10-2. Format of Directory Entry Returned by DIR\$RD or ENT\$RD

In PL/I, the declaration of *dir_entry* is

```
dcl 1 dir_entry based, /* Logical object entry. */
  2 ecw, /* Entry control word. */
    3 type bit(8), /* 3 for ACAT, 2 otherwise. */
    3 len bit(8), /* Length of structure (currently 31). */
  2 name char(32), /* Name of object. */
  2 pwprot_or_delprot, /* Password protection bits (for non-
                       ACL dirs) or delete-protect bit
                       (for ACL dirs). */
  3 owner, /* Owner protection bits. */
    4 reserved bit(5),
    4 delete bit(1), /* Can delete or truncate object. */
    4 write bit(1), /* Can write object. */
    4 read bit(1), /* Can read object. */
  3 delete_protect bit(1), /* Delete-protected bit. */
  3 nonowner, /* Nonowner protection bits. */
    4 reserved bit(4),
    4 delete bit(1), /* Can delete or truncate object. */
    4 write bit(1), /* Can write object. */
    4 read bit(1), /* Can read object. */
  2 non_default_acl bit(1), /* True if not protected by
                           default ACL. */

  2 reserved_1 bit(15),
  2 object_info, /* Information on object. */
    3 long_rat_hdr bit(1), /* BOOT or DSKRAT file on non-
                           floppy disk. */
    3 dumped bit(1), /* True if file has been backed up. */
    3 dos_mod bit(1), /* True if file modified under
                       PRIMOS II. */
    3 special bit(1), /* True if special file in MFD. */
    3 rwlock bit(2), /* Read/write lock. */
    3 reserved bit(2),
    3 type bit(8), /* Object type. */
  2 dtm, /* Date/time last modified. */
    3 date,
      4 year bit(7), /* 1900 is year 0. */
      4 month bit(4), /* January is month 1. */
      4 day bit(5), /* The first day of the month is day */
    3 time fixed bin(15), /* Seconds since midnight divided
                           by four. */

  2 reserved_2 fixed bin,
  2 reserved_3 fixed bin,
  2 truncated bit(1), /* True if truncated by FIX_DISK. */
  2 reserved_4 bit(15),
  2 dtbu, /* Date/time last backed-up. */
```

```

3 date,
  4 year bit(7), /* 1900 is year 0. */
  4 month bit(4), /* January is month 1. */
  4 day bit(5), /* The first day of the month is day */
3 time fixed bin(15), /* Seconds since midnight divided
                      by four. */
2 dtc, /* Date/time created. */
  3 date,
    4 year bit(7), /* 1900 is year 0. */
    4 month bit(4), /* January is month 1. */
    4 day bit(5), /* The first day of the month is day */
  3 time fixed bin(15), /* Seconds since midnight divided
                      by four. */
2 dta, /* Date/time accessed. */
  3 date,
    4 year bit(7), /* 1900 is year 0. */
    4 month bit(4), /* January is month 1. */
    4 day bit(5), /* The first day of the month is day */
  3 time fixed bin(15); /* Seconds since midnight divided
                      by four. */

```

The following changes to *dir_entry* at Rev. 23.0 should be noted:

- Special bits are set for both the root and each mount point because a reference to a mount point is directed to either an MFD (in the case of a disk partition) or is directed to the root (in the case of a root portal).
- When reading the entries in the root directory, be aware that there is a disparity between those entries and the attributes of the actual “grafted” disk partition MFD. For more information, see the section Entries in the Root Directory in Chapter 2 of this guide.

Example

Here is a sample PL/I subroutine that retrieves attributes for a file identified by a pathname, and displays some of the attributes:

```

display_attributes: proc(name,code);
dcl name char(128) var, /* Pathname of file. */
     code fixed bin(15); /* Standard error code. */
                      /* Other declarations omitted. */

```

```

if index(name,'>')=0 & substr(name,1,1) ^= '<'
  then do; /* Not a pathname, just read current directory. */
    call srch$$ (k$read+k$getu,k$curr,0,unit,type,code);
    if code^=0 then return;
    filename=name;
    end;
  else do; /* A pathname, open parent directory. *//*

First, call subroutine to split pathname (name) into parent
  directory name (pathname) and final objectname (filename). */

call get_parent_directory(name,pathname,filename);

/* Now, open parent directory for reading. */

chrpos(1)=0;
  chrpos(2)=length(pathname);
  call tsrc$$ (k$read+k$getu,(pathname),unit,chrpos,type,
    code);
  if code^=0 then return;
  end; /* if index(name,'>')^=0 */

/* Now read the desired entry. */

call ent$rd(unit,filename,addr(dir_entry),31,code);
if code^=0
  then do; /* If error, close directory and return. */
    call clo$fu(unit,i);
    return;

/* Now close the directory and return if error. */

call clo$fu(unit,code);
if code^=0 then return;

/* Display some info on the file. */

select (dir_entry.object_info.type);
  when('00'b4) call tnoua('SAM file',8);
  when('01'b4) call tnoua('DAM file',8);
  when('02'b4) call tnoua('SAM segdir',10);
  when('03'b4) call tnoua('DAM segdir',10);
  when('04'b4) call tnoua('DIRECTORY',3);
  when('06'b4) call tnoua('ACAT',4);
  when('07'b4) call tnoua('CAM file',8);
  otherwise call tnoua('Unrecognized type',17);
end; /* select (dir_entry.object_info.type) */

call tnoua('; ',2);

if non_default_acl then call tnoua('not default protected; ',23);

```

```

select (dir_entry.object_info.rwlock);
  when ('00'b) call tnou('sys', 3);
  when ('01'b) call tnou('EXCL', 4);
  when ('10'b) call tnou('UPDT', 4);
  when ('11'b) call tnou('NONE', 4);
  otherwise call tnou('????', 4); /* Theoretically impossible. */
end; /* select (dir_entry.object_info.rwlock) */

end; /* display_attributes: proc */

```

Setting File Attributes

You use the `SATR$$` subroutine to set most file attributes. The `SATR$$` subroutine can set attributes only on an object in the current directory. Therefore, you may have to include calls to `AT$` and `AT$HOM` to set a file attribute for an arbitrary object. `SATR$$` cannot work if the object is on a write-protected disk.

Usually, `SATR$$` also updates the date/time last modified (and date/time last accessed, if possible) of the parent directory. Exceptions are setting the dump bit and writing date/time last modified, date/time backed up, and date/time last accessed.

When calling `SATR$$`, your program provides

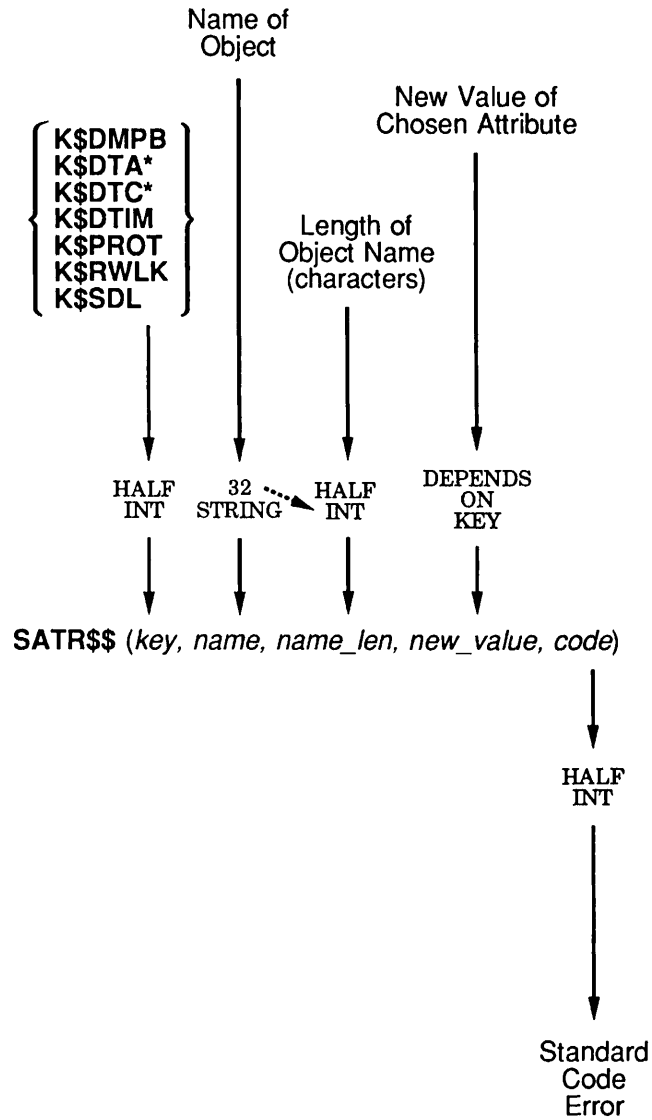
- The name of the file whose attributes are to be changed
- The length of the name
- A key that specifies the file attribute to be changed
- The new value of the file attribute

The `SATR$$` subroutine attempts to change the specified attribute to the new value, and returns an error code indicating whether or not the operation was successful. The caller must have protect rights on the object's parent directory in order to write any of the attributes except the dumped bit. The error `E$NRIT` indicates that the caller tried to set `dta` or `dtc` without belonging to the group, `.backup$`. The error `E$ATNS` indicates that the object is not an entry in a hashed directory; the `dtc` and `dta` attributes are not supported.

Figure 10-3 illustrates the calling sequence of `SATR$$`. This section describes the input and output parameters used when calling `SATR$$`, and then shows a sample call to `SATR$$`.

.....

See Attribute of Object



*For use only by .backup\$ group.

Side Effects: Updates dtm and dta attributes or parent directory if *key* is not K\$DMPB, K\$DTA, or K\$DTIM.

Q10.03D10056.31A

Figure 10-3. Calling Sequence of SATR\$\$

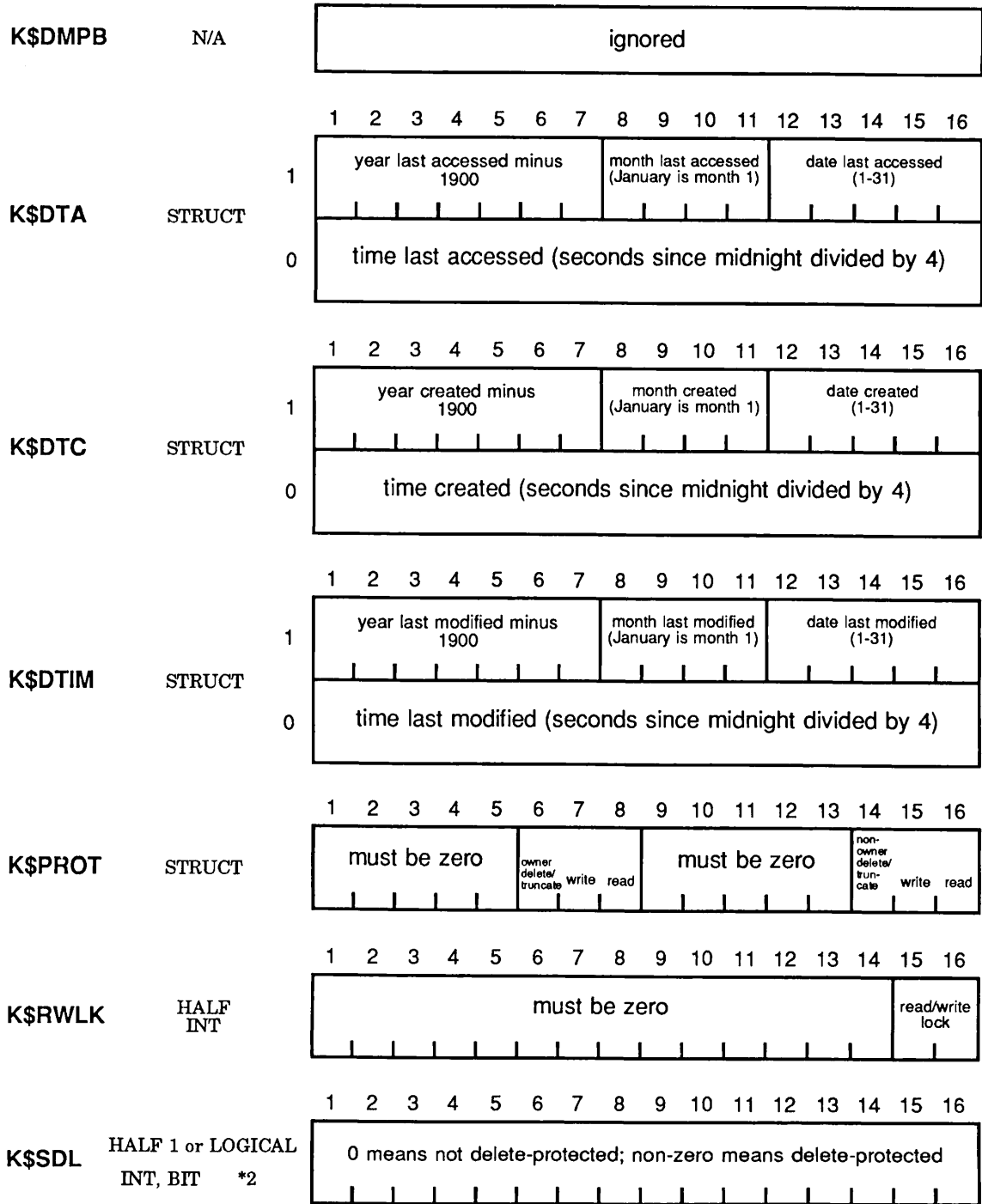
Key: Your program passes a *key* argument that specifies the file attribute to be changed. The values for *key* and their corresponding meanings can be one of

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
K\$DMPB	3	Set the dumped bit to 1. The only way to reset the dumped bit to 0 is by modifying the file or directory.
K\$DTA	10	Set date/time last accessed
K\$DTC	11	Set date/time created
K\$DTIM	2	Set date/time modified
K\$PROT	1	Set password protection keys (described in Subroutines Reference II: File System)
K\$RWLK	4	Set the read/write lock. Does not close file unit currently open to the file. Any such file units remain open, and no error indication is returned.
K\$SDL	6	Set delete-protected switch.

Caution Do not use the dumped bit to implement an incremental backup program unless the dumped bits are set only after the backup copy is verified to be readable.

New Value of the File Attribute: Your program supplies the new value of the file attribute for all keys, except for K\$DMPB which assumes a value of 1 (meaning Object dumped).

The formats of the new value for each key shown above are illustrated in Figure 10-4.



Q10.D4.D10056.3LA

Figure 10-4. Formats of SATR\$\$ Attributes for Each Key

Note Prior to Rev. 19.4, the second halfword of the new attribute value field had to be 0 when *key* was K\$PROT. This second halfword was thereby reserved for future use. As of Rev. 19.4, no second halfword is required, as future modifications are no longer planned for the K\$PROT key.

Four mnemonic keys are provided for use with the K\$RWLK key of SATR\$\$:

<i>Key</i>	<i>Meaning</i>
K\$DFLT	Default (system-wide) read/write lock. Depends on RWLOCK directive setting in system configuration file.
K\$EXCL	Exclusive. The file or segment directory may be open to several readers or to one writer, but not to both a reader and a writer, at the same time.
K\$UPDT	Update. The file or segment directory may be open to several readers and one writer at the same time.
K\$NONE	None. The file may be open to several readers and writers at the same time.

An output argument, *code*, informs your program of the success or failure of the operation. If *code* is 0, the operation was entirely successful. Otherwise, *code* is always positive. After a call to SATR\$\$ to set a file system attribute, contains a comprehensive list of all standard file system error codes. Error codes specific to this operation are

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$ATNS	238	Specified attribute is not supported in the directory. The target object does not have dta or dtc fields because it is not an entry in a hashed directory.
E\$BPAR	6	Bad parameter. The length of the objectname as passed by the calling program is less than 1 or greater than 32.
E\$NRIT	10	Insufficient access rights. The user must have Protect access to the parent directory of the object whose file attributes are being changed for keys other than K\$SDL; for K\$SDL, the user must have Delete access to the parent directory.

E\$NRIT may also indicate that the user tried to set dta or dtc without being a member of the group, .backup\$.

<i>Keyword</i>	<i>Value</i>	<i>Meaning</i>
E\$DIRE	14	Operation illegal on a directory. An attempt has been made to set the read/write lock for a file directory. File directories do not have read/write locks.
E\$IACL	15	Entry is an access category. An attempt has been made to set a file attribute other than the date/time last modified for an access category. See Chapter 9, Access Control Lists (ACLs), for information on access categories.

Example: The following FORTRAN statement changes the read/write lock of the file MY_DATABASE to UPDT (2):

```
CALL SATR$$ (K$RWLK, 'MY_DATABASE', 11, K$UPDT, CODE)
```

Disk Quotas

11



This chapter describes

- Retrieving information on disk space in use by a directory
- Improving quota system performance

Retrieving Information on Disk Space in Use

The Q\$READ subroutine is useful for finding out how much disk space is used in a given directory. It reports both the amount of space in use by the directory itself (including files and segment directories within that directory) and the total amount of space in use by the directory and all of its subdirectories. It also reports the maximum quota placed on that directory, but does not report information on quotas placed on parent directories of that directory, even though such quotas may restrict activity within the directory.

You can use Q\$READ to retrieve quota information on any directory residing on a Rev. 19.0 (or later) disk except for the MFD (Master File Directory), which requires the AVAIL command to determine how much disk space is being used.

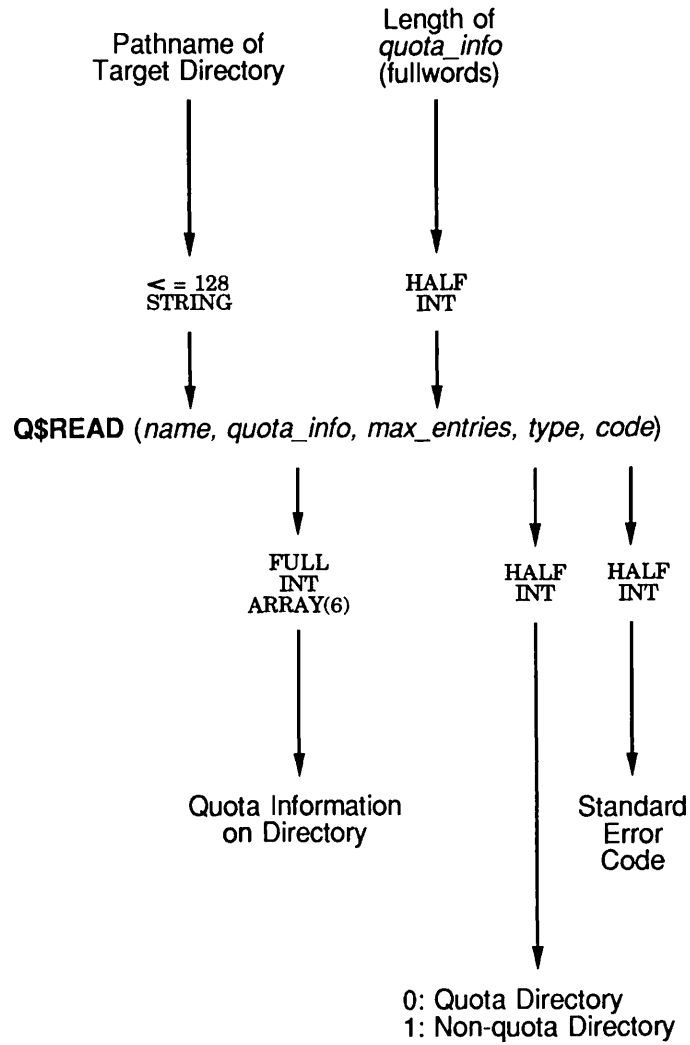
Retrieving Quota Information for a Directory

To retrieve quota information for a specific directory, simply call Q\$READ with the pathname of the directory. To retrieve quota information on the current directory, specify a null pathname.

An important piece of information returned is the quota/non-quota directory field. If the directory is not a quota directory, then the date/time last updated information for the directory is not maintained. Instead, this information is set to 0 by Q\$READ.

Figure 11–1 illustrates the calling sequence of Q\$READ. Figure 11–2 illustrates the returned array of directory quota information (*quota_info*).

Read Quota Information on File Directory



Side Effects: May reset cache attach point.

Q11.01.D10056.3LA

Figure 11-1. Calling Sequence for Q\$READ

Array Element	Halfword Offset								
	oct	dec							
1	0	0	Size of Disk Record in Halfwords (1024 for SMDs, CMDs, FMDs; 440 for Floppies)						
2	2	2	Number of Records Used by Directory						
3	4	4	Maximum Number of Records (Quota) 0 for Non-quota Directory						
4	6	6	Total Number of Records Used by Directory (# Records Used + Total # Records Used for All Subordinate Directories)						
5	10	8	Reserved						
(6	12	10)	Date/Time Last Updated (0 if Non-quota) in Format:						
6	12	10	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 33%; text-align: center;">year minus 1900</td> <td style="width: 33%; text-align: center;">mo(1=January)</td> <td style="width: 33%; text-align: center;">date (1-31)</td> </tr> <tr> <td style="text-align: center;"> _ _ _ _ _ _ _ _ _ _ _ _ _ _ </td> <td style="text-align: center;"> _ _ _ _ _ _ _ _ _ _ _ _ _ _ </td> <td style="text-align: center;"> _ _ _ _ _ _ _ _ _ _ _ _ _ _ </td> </tr> </table>	year minus 1900	mo(1=January)	date (1-31)	_ _ _ _ _ _ _ _ _ _ _ _ _ _	_ _ _ _ _ _ _ _ _ _ _ _ _ _	_ _ _ _ _ _ _ _ _ _ _ _ _ _
year minus 1900	mo(1=January)	date (1-31)							
_ _ _ _ _ _ _ _ _ _ _ _ _ _	_ _ _ _ _ _ _ _ _ _ _ _ _ _	_ _ _ _ _ _ _ _ _ _ _ _ _ _							
13	11		seconds since midnight divided by 4						

Q11.02.D10056.3LA

Figure 11-2. Structure of Directory Quota Information

Retrieving Quota Information for the MFD

To retrieve quota information for the MFD, your program must accumulate quota information for all top-level directories in the MFD and analyze the information. Useful information might include

- The total number of records in use. (Remember to count files and segment directories in the MFD itself, as they are not accounted for in any top-level directory by Q\$READ.)
- Whether the partition is open (has at least one top-level directory with no quota restriction) or closed (all top-level directories have quotas placed on them).
- If the partition is closed, the total of all top-level directory quotas (the total quota for the partition).
- If the partition is closed, whether it is overcommitted (total quota greater than the partition size) or undercommitted (total quota less than the partition size), and by how many records.

Improving Quota System Performance

If your system does not use disk quotas, an attempt to read quota information for a directory may take some time, as the quota system must size all files and directories within a directory to produce the directory-used and total-used values.

To speed this up, set a very large maximum quota on all top-level directories on all disk partitions on your system. A maximum quota of 1000000 (one million) records suffices. This forces PRIMOS to maintain up-to-the-minute quota information on all directories on your system. As a result, using Q\$READ (or the LIST_QUOTA command) potentially takes much less time. However, minor overhead cost is incurred when this is done.

You can write a program that does this using the Q\$SET subroutine, although the user who runs the program must have Protect access to the MFD of the disk partition on which the program is being run. Your program would set high quotas on all directories that do not already have quotas.

Bear in mind that a quota cannot be set on a non-quota directory that is in use by any user. This includes situations where a user is attached to a subdirectory of the non-quota directory. Therefore, it is best to run such programs immediately after system coldstart, or just before the system is shut down (but after all users are logged out).

12

Interprocess Communication via the File System

The PRIMOS file system may be used to communicate between processes. For example, an electronic mail subsystem can use a directory as the mail database, and use specific files within the directory to communicate between different processes of the subsystem.

This chapter describes the general concepts involved when using the file system for interprocess communication. Some specific direction is then given for solving typical interprocess communication problems using the file system.

General Concepts

For applications that require multiple processes to run simultaneously, you need some form of interprocess communication. If your application does not require high transaction processing rates, such as more than one transaction per second, you might find that relying upon the PRIMOS file system for all of your interprocess communication saves you development and maintenance cost.

If your application requires more than one transaction per second, you can still use the file system for primary storage, but the interprocess communication mechanism might be more efficiently handled by a combination of shared data and semaphores, at the expense of increased development and maintenance cost. See the *Subroutines Reference* series for further information.

File and System Read/Write Locks

For all files in your database, you must determine the appropriate per-file read/write locks. All files are created with a read/write lock of SYS, meaning use the system-wide read/write lock. The system-wide read/write lock is set by the RWLOCK configuration directive during system cold start. See the *System Administrator's Guide 1: System Configuration* for further information on the RWLOCK directive.

Typically, the system-wide read/write lock is 1, corresponding to a per-file read/write lock of EXCL (multiple readers or 1 writer). This is typically the most appropriate setting for database files. However, you should consider the effects on your application should the system-wide read/write lock be 3,

corresponding to a per-file read/write lock of UPDT (multiple readers and 1 writer), or should it be 0, meaning 1 reader or 1 writer.

If your application does not operate correctly with a non-standard system-wide read/write lock, then you should take one of the following actions:

- Document the restriction and have your application perform a safety check the first time it is started up after each system cold start.
- Avoid the restriction by placing per-file read/write locks on all files in your database.

Documenting the Restriction and Performing a Safety Check: To document the restriction your application places on the system-wide read/write lock, include a sentence in the System Requirements portion of your documentation that reads as follows:

This product requires the system-wide read/write lock to be set to 1 for proper operation.

The system-wide read/write lock is controlled by the RWLOCK configuration directive in your system configuration file (usually named CONFIG). If the RWLOCK directive is not present, or has an argument of 1, the requirement is satisfied. However, if it has an argument of 0 or 3, this product will not operate properly.

In addition, it is wise to have your product perform a safety check to make sure the system-wide read/write lock is set to the correct value. This safety check can be performed in CPL, using the following CPL program. This program returns the system-wide read/write lock as its function value.

```
&if [exists t$temp_file.t] &then delete t$temp_file.t -no_query
&severity &error &ignore
open t$temp_file.t 1 40002 /* Open for write, creating it too.
&s sev := %severity%
&if %sev% ^= 0 &then &result UNKNOWN
&else &do
    open t$temp_file.t 1 40001 /* Open it again for read.
    &s sev := %severity%
    &if %sev% = 0 &then &result 3 /* UPDT lock.
    &else &do /* Could be EXCL (1) or SNGL (0).
        close t$temp_file.t /* Close the file.
        open t$temp_file.t 1 40001 /* Open for read.
        &s sev := %severity%
        &if %sev% ^= 0 &then &result UNKNOWN
        &else &do
            open t$temp_file.t 1 40001 /*Again.
            &if %sev% = 0 &then &result 1
            &else &result 0
        &end
    &end
&end
```

```

                                &end
                                &end
close t$temp_file.t
delete t$temp_file.t
&return

```

If the above CPL program is entitled RWLOCK.CPL, then the following sequence of CPL statements verifies that the system-wide read/write lock is 1 or displays an error message:

```

&if [r rwlock] ^= 1 &then &do
  type System-wide read/write lock <RWLOCK> not set to 1. XYZ
  type product cannot operate under these conditions. Please
  type delete the RWLOCK directive from the system configuration
  type file, then start up the XYZ application again using the
  type START_XYZ command. For more information, see the XYZ
  type Guide, and the PRIMOS System Administrator's Guide.
  &data message /* Message to supervisor terminal.
XYZ product shutting down: unsupported RWLOCK configuration.
                                &end
                                &return 1 &message RWLOCK not set to 1
                                &end

```

Placing a Per-file Read/Write Lock on Each File: A method of insulating your product from the system-wide read/write lock value is to have your application place a per-file read/write lock on each file it uses. This means that each time your application creates a file, your application must call the SATR\$\$ subroutine to set the read/write lock of the file to the appropriate value. (See Chapter 10, File Attributes, for information on calling SATR\$\$.)

With this method, two problems exist:

- To set the read/write lock of a file, the user running your application must have Protect access to the parent directory of the file.
- Although creating a new file implies that the file is open for writing, your application must close the file and then reopen the file after setting the read/write lock for it, so that the new read/write lock setting may take immediate effect. If your application does not perform these steps in the order indicated, a window of time may still exist during which one process may open a file for reading while another process has it open for writing.

Caveats on Using the File System for Interprocess Communication

Under no circumstances should your application depend on the timing characteristics of the PRIMOS file system or of any other part of PRIMOS. If such a dependency is built in, then your application may be traumatized when

run on different models of Prime computers or on different revisions of PRIMOS. Additionally, the timing characteristics of the file system may vary with the system load at any given moment.

It is assumed that a database used for interprocess communication between processes in a given subsystem is accessed only by processes belonging to (or operating under the auspices of) that subsystem. PRIMOS makes no direct attempt to distinguish processes relating to a subsystem from other processes. If the database is accessed by a process that is outside the domain of the subsystem, the following may result:

- The contents of the database may be rendered invalid.
- Processes within the domain of the subsystem may encounter file system errors, such as E\$FIUS (File in use).
- Portions of the database that are protected only by the subsystem itself, not by the PRIMOS ACL mechanism, may be read or written by any users when outside the domain of the subsystem.

In summary, from the point of view of a subsystem database, all processes fall into two categories:

- Cooperating processes
- Noncooperating processes

If cooperative processes can be identified by user ID or group name, then the database can be protected by the PRIMOS ACL mechanism against unauthorized access.

However, certain applications require the ability for any process to become a cooperating process when running a program that is part of the subsystem. For example, an electronic mail system might require that all users be able to send and receive mail using a command such as MAIL, and yet these same users must not be allowed to access crucial portions of the database when not using the MAIL command. If this is a requirement of your subsystem, you have two choices:

- Accept the potential consequences described earlier in this section.
- Use a different interprocess communication mechanism, such as the PRIMENET X.25 interface.

Sample Models of Communication via File System

This section discusses several sample subsystem models, all of which can be implemented using the file system for interprocess communication. The models discussed are

- Multiple processes creating file-based transactions
- Multiple competing servers accessing file-based transactions
- Two-process transaction management
- Multiple processes accessing a database

Multiple Processes Creating File-based Transactions

Certain applications, such as electronic mail subsystems, require the ability for multiple processes to create new transactions, such as pieces of electronic mail, to be processed by one or more server processes.

It is convenient for such a subsystem to store transactions in a lower-level directory within the subsystem database, where each transaction is stored in its own file.

There are two requirements:

- While a transaction file is being created, it is incomplete and must not be read by one of the server processes.
- Once a transaction file is created, it must be overwritten or deleted by only one of the server processes. A transaction file must not be reused for another transaction until the original transaction has been serviced.

Preventing Premature Servicing of a Transaction: To prevent a server process from servicing a file-based transaction before the transaction has been completely written, one of two approaches can be used:

- A central database file, containing information on all outstanding transactions, can be used to indicate the status of each outstanding transaction.
- A field within the transaction file can be used to indicate the status of the transaction.

The status of a transaction is used to distinguish between a transaction being written, waiting for servicing, and being serviced. In both of the above situations, the process that creates a transaction would update the transaction status after it finished writing the transaction.

This ordering of events would prevent the transaction from being considered complete if the process creating the transaction was aborted (such as by a force-logout) before it finished writing the transaction. A server process might be unable to open a transaction file if it is still in use by the process creating the transaction file. This is true if the system-wide read/write lock (RWLOCK) is set to 0 or 1. This inability to open a transaction file can be used by a server process to recognize a transaction creation in progress.

Preventing Reuse of a Transaction File: To prevent inadvertent reuse of a transaction file that has not yet been serviced, a unique name can be assigned to each transaction file. When this method is used, Add and Use access to the lower-level directory containing the transaction files is the only access required for processes creating transaction files.

An obvious solution to the problem of preventing inadvertent reuse of a transaction file is for a process to pick a filename using some algorithm and then check for the previous existence of a file with that name. This approach has two problems:

- It is possible for process A to create a file between the point in time that process B tests for the existence of the file and the point in time that process B subsequently uses the file. If this happens, both processes use the same transaction file.
- If the algorithm used to pick a filename is limited to a sufficiently small set of possible filenames, a process could spend an unreasonable amount of time testing filenames representing existing files if enough transactions were pending. This would reduce system performance at a time when performance needs to be at its best to process the pending transactions.

Multiple Competing Servers Accessing File-based Transactions

When files in a directory represent transactions, or units of work, it is often desirable for one of several transaction server processes to read a transaction file, perform the transaction described within, and delete the transaction file. The sequence of events is

1. A server process, S, is directed to process a transaction represented by (and described within) a file, F.
2. Server process S opens file F.
3. Server process S reads the contents of file F and performs the corresponding transaction.
4. Having performed the transaction, server process S now closes file F.
5. Server process S deletes file F to signal the completion of the transaction to other server processes within the subsystem.

This sequence of events may not result in a sufficiently robust interprocess communication mechanism. The sequence shown above implies one crucial assumption involving interprocess communication:

Once Step 1 is in progress, no other server process attempts to perform the transaction described in file F.

If transactions are being assigned to server processes by a central process, this assumption can be satisfied by having the central process refuse to assign file F to another server process unless server process S is unable to complete the transaction.

If there is no central process, then, in Step 1, server process S must choose transaction file F for itself, based on some search algorithm. The following methods of preventing other competing server processes from making the same choice are in common use and are discussed below.

1. A central database file is used to maintain information on the outstanding transactions. In this case, each transaction is represented by a record within the central database file, and records for transactions being serviced also identify the server process servicing the transaction.
2. The beginning of each transaction file contains a field that describes whether and by which process the transaction is being serviced.
3. The transaction file is kept open while the transaction is being serviced, preventing other server processes from opening the same transaction file.

Method 1 — Central Database: Using a central database file to maintain information on transaction status has advantages and disadvantages. The primary advantage is that status on all transactions can be retrieved by reading only one file. Disadvantages are

- Access to the central file must be single-threaded, possibly reducing overall throughput.
- Additional overhead is incurred whenever a transaction is serviced, as its entry in the central file must be updated or deleted to reflect this fact.
- A premature server abort may cause the transaction status to be left in the “being serviced” state too long.

Method 2 — Transaction File Status Field: Maintaining a status field at the beginning of each transaction file has several advantages:

- Status is easily updated by the server process servicing the transaction represented by the file, simply by rewriting the transaction status field at the beginning of the file.

- When the transaction is completed, the status need not be updated if the transaction file is deleted.

However, this method has the following disadvantages:

- The status of all transactions must be obtained by examining each transaction file.
- A premature server abort may cause the transaction status to be left in the “being serviced” state too long.

Method 3 — Transaction File Kept Open: The status of a transaction can be inferred by the state of the transaction file. If it is in use, that is, open for reading and writing, then it is either being created or being serviced. This approach has its advantages:

- If the server process is aborted, then the act of its logging out closes the transaction file. This effectively implies a change to a “waiting for service” status, allowing other server processes to open and service the file.
- The status of the file is automatically updated when the server process opens the file. No separate operation need be performed to update the status.

This method also has its disadvantages:

- To determine the status of all transactions, each transaction file must be tested to see if it is in use.
- Constraints on the effective read/write lock of transaction files exist. Multiple writers must never be allowed to open the file, and if the file is open for writing, no other processes should be able to open it for reading. This implies that the per-file read/write lock must be either EXCL or must be SYS. If it is SYS, then the system-wide read/write lock (RWLOCK) must be 0 or 1.
 - Between Steps 4 and 5 above, that is, after closing a completed transaction file and deleting it, another server process may find that it is not in use and open the file. To prevent this, the file access can be set so that it does not include Read or Write access for server processes. (Use an existing category ACL, with a name like TO_BE_DELETED.ACAT, for best results.)
 - After the access is changed, then Step 4 can be performed followed by Step 5. Any attempt by another server process to open the transaction file between Steps 4 and 5 results in an insufficient access rights error.

Two-process Transaction Management

A subsystem that consists of two processes usually conforms to one of two models:

- One process creates transaction files, the other process services and deletes them.
- Both processes create and service transaction files.

The first model might be a distributed transaction processing service. One process receives transactions from other nodes on a network and deposits these transactions in the database. The other process reads these transactions, services them, and then deletes the transaction files.

The second model might be an electronic mail gateway service. Here, one process services the electronic mail traffic for the local network, while the other process services the incoming and outgoing electronic mail traffic for other networks (such as a Public Data Network, or PDN).

The second model can be considered a bidirectional version of the first model. To implement one direction of transaction communication, dedicate a subdirectory of your database to this single direction. The process that creates transactions can use the UID\$BT and UID\$CH subroutines to determine unique filenames, and then writes files with these names in the lower-level directory. The process that services the transactions can use the DIR\$RD subroutine to continually scan the lower-level directory for new transaction file arrivals.

When using this approach, the file creation process is the only process creating files in the lower-level directory, and the file servicing process is the only process servicing files in the lower-level directory. There is only one concern over read/write locks in this case: while a transaction file is being written by the transaction file creation process, the transaction file servicing process must not attempt to service the transaction. This implies that the system-wide read/write lock is set to 0 or 1, or that the per-file read/write lock is set to EXCL (using the open/set-lock/close/open sequence described earlier).

One way to avoid the read/write lock concern entirely is to use the per-file dumped bit to signal the readiness of a transaction file. When a file is created, the dumped bit is reset. After the process finishes creating the file, it can use SATR\$\$ to set the dumped bit. Meanwhile, the other process is using DIR\$RD to scan for new transaction files. Because DIR\$RD also returns the dumped bit for a file, it can avoid opening a file that has the dumped bit reset.

Multiple Processes Accessing a Database

For concurrent access to a database, Prime offers the MIDASPLUS system. If you do not need the full potential of MIDASPLUS, you can design your own

database system that uses only the PRIMOS file system for concurrency management.

This is particularly appropriate if your subsystem uses a small number of central database files to manage a larger number of transaction files. This possibility has been discussed earlier in this chapter. If this is the case, you must ensure that two processes do not attempt to update a central file simultaneously, and that one process does not attempt to read a central file while it is being written.

This implies that the system-wide read/write lock is restricted to 1, or that all central files in the database have their read/write locks set to EXCL (multiple readers or 1 writer).

Performing record locking within a file is not an alternative, since there is no reliable method of updating a field within the file from one value to another while preventing another process from updating the same field. Moreover, such an occurrence cannot be detected by either process.

For example, if process P wishes to lock a record within the file, it might read a field in the file that indicates the record is not in use. It would then update this field to indicate that it is using the record. In the meantime, however, process Q could perform the same sequence of operations, and both processes would then operate as if they had locked the record, although the field would record only one process as owning the record lock.

Therefore, it is recommended that central database files all have effective read/write locks of EXCL. If, for example, you need one central database file to manage pending transactions in your database, and you believe that single-threading access to the central database file will result in insufficient throughput, you might consider using several central database files. Here, the appropriate central database file would be selected using a hash function on the transaction key. This approach might increase throughput.

Appendix

.....

File System Glossary



This appendix contains a glossary of terms that refer to concepts and objects peculiar to operating systems/file systems in general, and the PRIMOS file system in particular. Terms that are italicized in these definitions are defined elsewhere in the glossary.

common file system name space

The object *name space* in the PRIMOS file system, specifically the set of names that the *Name Server* manages. The boundaries of the common file system name space are defined by a *DSM config group*.

data

Information that takes the form of letters, digits, symbols, and special characters.

data record

A number of fields that can be combined into a structured element.

directory tree

A directory structure in which logically-related objects are stored in a hierarchical fashion. Files reside within directories, which in turn may reside within other directories. The layout of the tree can be thought of as mimicking that of a physical file cabinet.

disk-directed portal

A file system object which redirects references to the MFD of a specified disk. This type of portal is defined primarily for compatibility with earlier revisions of PRIMOS which do not have a root directory.

disk partition

A logical disk unit; there may be one or more disk partitions on one physical disk pack.

disk name

Each disk partition is named by a six-character identification string, which is used as part of a *pathname* syntax form.

disk tree or naming tree

A *directory tree* structure that is restricted to one *disk partition*; the *root* of a disk tree is the MFD.

DSM config group

A subset of a network in which a group of machines is administered by one authority. A config group may not intersect with another config group. DSM stands for Distributed Systems Management.

field

Information in the form of letters, digits, and symbols arranged into useful groups of words and numbers. A field is usually designated as either alphanumeric (consisting of a mixture of letters, digits, and symbols) or numeric (consisting mostly of digits, but possibly including a plus or a minus sign, a decimal point, one or more commas, and perhaps a currency symbol). Other kinds of fields, such as pure alphabetic or binary, are recognized by some programming languages.

fully-qualified pathname

A pathname whose name is qualified by the root directory symbol (<).

logical mount

A partition that has been added using the `-MOUNT_PATH` option. The logical mount is either a partition that is grafted to a directory at a point lower in the tree hierarchy than the root, or a partition mounted in the root directory with a name between 7 and 32 characters long.

mount point directory

Any directory in the tree hierarchy (except the MFD) over which a partition has been grafted. The contents of the mount-point directory are inaccessible until the grafted partition is shut down.

multi-rooted name space

A *directory tree* hierarchy in which the name of every object in is ultimately defined by the name of its disk partition.

name or objectname

The name that references a particular file system object. The name must be unique within the *common file system name space*, and must adhere to the PRIMOS rules for length and permitted characters.

name space

The set of objectnames within the specified boundaries of a file system.

naming sphere

The name given to the boundary of the *common file system name space*; within the naming sphere, security and access controls are controlled by CONFIG_NET and DSM.

Name Server

The PRIMOS server that controls the set of functions that manipulate the *common file system name space* across a set of machines previously defined by DSM.

network

A collection of machines that have an embedded communication path among themselves. There may be one or more *common file system name spaces* within a given network.

partition ID

ID for a disk partition that allows the *Name Server* to distinguish multiple disk partitions with the same disk name.

pathname

A sequence of nodes that refers to a specific traversal path in a *directory tree*, where intermediate nodes in a path are usually directories.

portal

A file system object that provides a naming gateway to another *common file system name space*. A portal makes it possible to have transparent references to file system objects that are outside of the local common file system name space. The two types of portals are *root-directed* and *disk-directed*.

private partition

A *disk partition* that is not automatically available to the *common file system name space*. The private partition is visible to users on the local system, dependent upon ACLs.

record

A collection of fields; it is the basic unit upon which most file systems operate.

root or root directory

The topmost directory of the directory tree. The root directory has special characteristics, and contains directory entries of every *disk partition* in the *common file system name space* except a *logical mount*.

root-directed portal

A file system object which redirects references to the root directory of another machine. The remote machine can be any machine outside your file system name space as long as it resides on the network.

singly-rooted name space

A file system hierarchy in which the name of every object in is ultimately defined by the root directory.

text record

An unstructured record that consists of strings of alphanumeric information of varying lengths.

Index

.....

Index

Symbols

\$, 5–10

/* (comment indicator), 5–10

A

A access right, definition, 4–5

AC\$CAT subroutine

calling sequence, 9–5

introduction, 9–2

AC\$CHG subroutine

calling sequence, 9–7

using, 9–6

AC\$DFT subroutine

calling sequence, 9–3

introduction, 9–1

AC\$LIK subroutine

calling sequence, 9–8

using, 9–6

AC\$LST subroutine

calling sequence, 9–9

using, 9–8, 9–10

AC\$SET subroutine

calling sequence, 9–4

introduction, 9–2

using, 9–6

ACATs. *See* Access categories

Access categories (ACATs)

creating, 4–19

creating with AC\$SET, 9–2

definition, 2–16

Access control, discussion, 3–6

Access Control Lists. *See* ACLs

Access methods

file system, 2–10

files, 3–5

Access rights

attaching to directories, 3–11

calculating when opening files, 3–11

category access, 4–18

changing, 4–20

deleting, 4–21

setting default, 4–16

setting identical, 4–18

specific, 4–17

ACLs

access categories, 2–16

calculating, 3–9

changing, 4–20, 9–6

creating identical, 4–18

default, 4–6

definition, 2–10

deleting, 4–21

description, 4–5 to 4–6

discussion, 9–10

functions, 4–15

parsing with programs, 9–10

programmer access, 3–7

reading, 9–8

root directory, 2–12

setting category access, 9–2

setting default access, 9–1

setting identical, 9–6

setting specific access, 9–2

subroutine manipulation, 9–1

–added_disks keyword, 5–15, 5–16

Administrator search rules, 5–3

ALL access right, definition, 4–6

Arguments, introduction, 4–3

AT\$ subroutine, 6–7 to 6–8

calling sequence, 6–9

AT\$–type subroutines, list, 6–4

AT\$ABS subroutine, 6–10

calling sequence, 6–11

AT\$ANY subroutine, 6–13

calling sequence, 6–14

AT\$HOM subroutine

calling sequence, 6–3

discussion, 6–3

AT\$OR subroutine

calling sequence, 6–2

discussion, 6–1 to 6–2

AT\$REL subroutine, 6–16

calling sequence, 6–17

AT\$ROOT subroutine, 6–19

calling sequence, 6–19

Attach points

changing, 4–12

current, 3–7, 4–5, 6–4

discussion, 6–25

home, 6–3

initial, 4–4, 6–1

introduction, 4–4

manipulating, 6–7

resetting current, 6–5

ATTACH\$, discussion, 5–5

ATTACH\$ search list

discussion, 5–6, 5–7

introduction, 5–2

unqualified pathnames, 3–4

using added_disks keyword, 5–15

using with other search lists, 5–21

Attaching

access rights, 3–11

directory, 3–6

discussion, 4–12

subroutine, 4–14

Attributes

DTA, 3–21

DTB, 3–24

DTC, 3–22

DTM, 3–23

dumped/not–dumped, 3–26

file system objects, 10–1

Attributes (Continued)

- file type, 3-25
- general discussion, 4-7
- object, 3-20
- read/write locks, 3-24 to 3-25
- setting file, 10-7
- special/not-special, 3-27

B

- Badspot file, definition, 2-14
- .BIN files. *See* Binary files
- Binary files, 5-8
- BINARY\$ search list
 - discussion, 5-8
 - introduction, 5-2
- Blocking factor
 - definition, 7-42
 - determining, 7-43
- Bootstrap file, definition, 2-13

C

- CALAC\$ subroutine, 9-10
- CAM files, 3-5
- Category access rights
 - creating, 4-19 to 4-20
 - setting, 4-18
- Changing ACLs, 9-6
- CLOSEFN subroutine, calling sequence, 7-22
- CLOSEFU subroutine, calling sequence, 7-20
- Closing
 - files, 3-19, 7-19
 - objects, 4-37
 - text files, 7-5
- Command functions, introduction, 4-2
- Command Procedure Language. *See* CPL
- COMMAND\$ search list
 - discussion, 5-7
 - introduction, 5-2
- Commands, 4-1
- Comment indicator. *See* /*
- Common file system name space
 - characteristics, 2-4, 2-7
 - definition, A-1
- Compressed files, 7-3
- Config groups, definition, A-2

CPL

- checking system read/write locks, 12-2
- introduction, 4-2
- Creating
 - directories, 8-28
 - file directories, 4-24
 - files, 4-26
 - objects, 4-10
 - portals, 4-22
 - search rules, 5-9 to 5-10
- Current attach point
 - definition, 3-7
 - discussion, 4-5, 6-4
 - resetting, 6-5
- Current directory, opening, 6-23
- Current object position, discussion, 3-13

D

- D access right, definition, 4-5
- DAM files, 3-5
- Data, definition, 1-1, A-1
- Data files
 - discussion, 8-41
 - numbered, 2-15
 - reading and writing, 8-40
- Data records, definition, 1-1, A-1
- Date and Time Created attribute. *See* DTC attribute
- Date and Time Last Accessed attribute. *See* DTA
- Date and Time Last Backed Up attribute. *See* DTB attribute
- Date and Time Last Modified attribute. *See* DTM attribute
- Default access rights, 4-16
- Default search lists, introduction, 5-2
- Deleting
 - ACLs, 4-21
 - member files, 8-21
 - objects, 4-11, 4-38
- DIR\$CR subroutine
 - calling sequence, 8-30
 - using, 8-28
- DIR\$RD subroutine
 - calling sequence, 8-38
 - directory entry returned, 10-3
 - introduction, 8-28
 - using, 8-37

dir_entry declaration

- example, 10-4, 10-5
- Rev. 23.0 changes, 10-5

Directories

- access calculation when attaching, 3-11
- attaching, 6-13, 6-16
- attaching to file, 3-6
- attaching via subroutines, 4-14
- creating, 4-24, 8-28
- current, 3-7
- defined, 1-2
- dumped bit, 3-26
- home, 3-3
- lower-level, 2-15
- manipulating, 8-28
- mount-point, 2-8
- opening, 4-27, 8-32
- origin, 2-15
- password, 3-8
- reading, 4-31
- root, 2-5
- scanning, 8-37
- segment, 2-15, 8-2
- setting ACLs, 9-1
- writing, 4-35

Directory trees, definition, A-1

Disk partitions, definition, 2-14, A-1

Disk quotas, introduction, 11-1

Disk Record Availability Table (DSKRAT), 2-13

Disk trees, definition, A-2

Disk-directed portals, definition, 2-9, A-1

Disk-shut-down flag, 3-15

Disknames, definition, A-1

Disks

- physical, 2-13
- space in use, 11-1

Distributed System Management (DSM), Name Server usage, 2-7

DSM. *See* Distributed System Management

DSM config group, definition, A-2

- DTA attribute
 - characteristics, 3-21
 - format, 3-22

DTB attribute, definition, 3-24

DTC attribute, definition, 3-22

DTM attribute, definition, 3-23

Dumped bit, 3-26
Dynamic allocation, definition, 3-17

E

ENT\$RD subroutine
 calling sequence, 10-2
 directory entry returned, 10-3
 introduction, 8-28
 using, 10-1
ENTRY\$ search list
 discussion, 5-9
 introduction, 5-2
EPFs, 5-9
Error codes, object type, 3-14
ESR command. *See*
 EXPAND_SEARCH_RULES (ESR)
 command
Executable Program Format. *See* EPFs
EXPAND_SEARCH_RULES (ESR)
 command, 5-5, 5-7, 5-8
 CPL programs, 5-20
 using, 5-19
Extending
 segment directories, 8-13
 text files, 7-5
Extents, 3-6

F

Fields, definition, 1-1, A-2
File attributes
 discussion, 3-20, 4-7
 setting, 10-7
File pointers, discussion, 3-18
File system
 closing objects, 4-37
 creating objects, 4-22
 definition, 2-1
 deleting objects, 4-38
 glossary of terms, A-1
 opening objects, 4-27
 overview, 4-9
 pre-Rev. 23.0, 2-2
 primitives, 6-6
 PRIMOS, 2-1
 programmer interfaces, 4-1
 reading objects, 4-30
 Rev. 23.0, 2-4

 singly-rooted, 2-5
 starting point, 2-10
 writing, 4-35
File system name space, definition, 2-3
File system objects, discussion, 2-9
File types, attribute, 3-25
File units
 characteristics, 3-13
 definition, 3-13
 discussion, 4-7
 numbers, 3-17, 3-18
 opening, 3-11
 using RDLIN\$, 7-23
 using WTLIN\$, 7-23
Files
 access methods, 3-5
 accessing with subroutines, 7-1
 CAM, 3-6
 closing, 3-19, 7-19
 compressed, 7-3
 creating, 4-26
 DAM, 3-5
 defined, 1-2
 definition, 2-16
 fixed-length, 7-30
 manipulating with PRWF\$\$, 7-30
 maximum length, 7-5
 open, 3-5
 opening, 3-16, 4-29, 7-6
 organizing, 8-1
 positioning, 3-19
 positioning to end-of-file, 7-13
 read/write locks, 12-1, 12-3
 reading, 4-33
 SAM, 3-5
 search rules, 5-1
 setting ACLs, 9-1
 text, 7-1, 7-45
 transaction, 12-5
 truncating, 3-19, 7-16
 uncompressed, 7-3
 variable-length record, 7-23
 writing, 4-36
Find free entry function, introduction,
 8-23
Find full entry function, introduction,
 8-23
Fixed-length files
 discussion, 7-30
 format, 7-42

Fixed-length records
 differing from variable length, 7-2
 discussion, 7-3, 7-42
 introduction, 7-1
 using with variable length, 7-4
Flags, 3-15
Fully-qualified pathnames, 2-10
 definition, A-2
 determining, 6-20
 discussion, 3-2
Functions
 ACLs, 4-15
 find free entry, 8-23
 find full entry, 8-23

G

Global Mount Table (GMT)
 discussion, 2-7, 5-16
 reading, 4-34
GMT. *See* Global Mount Table
GPATH\$ subroutine, 6-20
 calling sequence, 6-21
 preserving attach point, 6-26
 returned pathnames, 6-22

H

Home attach point, discussion, 4-5, 6-3
Home directories, attaching, 4-13
[home_dir] keyword, 5-18

I

INCLUDE\$ search list
 discussion, 5-8
 introduction, 5-2
Initial attach point
 discussion, 6-1
 introduction, 4-4
 using the [origin_dir] keyword, 5-18
-insert keyword, 5-12
Interprocess communication
 caveats, 12-3
 introduction, 12-1
 sample models, 12-5

K

Keywords

- [home_dir], 5-18
- [origin_dir], 5-18
- [referencing_dir], 5-19
- added_disks, 5-15, 5-16
- insert, 5-12
- optional, 5-15
- primos_direct_entries, 5-17
- public, 5-17
- static_mode_libraries, 5-17
- system, 5-13
- search rules, 5-12

L

- L access right, definition, 4-5
- LIST_SEARCH_RULES command, 5-11
- Locks, read/write, 3-16, 3-24 to 3-25, 12-1, 12-3
- Logical mounts
 - definition, A-2
 - discussion, 2-8
- Lower-level directories, definition, 2-15
- LSR command. *See* LIST_SEARCH_RULES command

M

- Master File Directory. *See* MFD
- Member files
 - deleting within a SEGDIR, 8-21
 - opening within a SEGDIR, 8-16
- MFDs
 - definition, 2-14
 - quota information, 11-3
- Modes, open, 3-13
- Mount-point directories, definition, 2-8, A-2
- Multi-rooted name space, definition, A-2
- Multi-rooted tree hierarchies
 - discussion, 2-2
 - limitations, 2-4
- Multiple processes
 - accessing, 12-10
 - file-based transactions, 12-5
 - managing, 12-9

- Multiple servers, file-based transactions, 12-6

N

- Name. *See* Objectname
- Name Server
 - definition, A-3
 - discussion, 2-7
 - using -added_disks keyword, 5-16
- Name space
 - definition, A-2
 - file system, 2-3
 - multi-rooted, A-2
 - singly-rooted, A-4
- Naming sphere, definition, A-3
- Networks, definition, A-3
- NONE access right, definition, 4-6
- Numbered data files, 2-15

O

- O access right, definition, 4-5
- Object types, 3-14
- Object-modified flag, 3-15
- Objectnames
 - character components, 3-2
 - conventions, 3-2
 - definition, A-2
 - discussion, 4-6
- Objects
 - access control, 4-12
 - accessing, 3-9
 - calculated access, 3-15
 - changing ACLs, 9-6
 - closing, 4-37
 - creating, 4-22
 - definition, 1-2
 - deleting, 4-11, 4-38
 - determining current position, 3-13
 - file system, 2-9
 - manipulating, 4-10 to 4-11
 - naming, 3-5
 - naming and accessing, 2-10
 - opening, 4-10, 4-27
 - reading, 4-10, 4-30
 - Reading file attributes, 10-1

- writing, 4-11, 4-35

- Open mode, valid operations, 3-13

Opening

- current directory, 6-23
- directories, 4-27, 8-32
- files, 3-11, 3-16, 4-29
- member files, 8-16
- objects, 4-10
- segment directories, 8-3
- text files, 7-5
- optional keyword, 5-15
- Origin directories
 - attaching, 4-12
 - definition, 2-15
 - using with [origin_dir] keyword, 5-18
- [origin_dir] keyword, 5-18

P

- P access right, definition, 4-5
- Parsing, ACLs with programs, 9-10
- Partition IDs, definition, A-3
- Partitions
 - See also* Disk partitions
 - grafted, 2-8
- Password directories, discussion, 3-8
- Pathnames
 - definition, 3-1, A-3
 - discussion, 3-2
 - fully-qualified, 2-10, 3-2, A-2
 - relative, 3-3
 - simple, 3-4
 - unqualified, 3-4
- Performance, improving with quotas, 11-4
- Physical disks, discussion, 2-13
- Physical records, discussion, 2-13
- Portals
 - creating, 4-22
 - definition, 2-8, A-3
 - disk-directed, 2-9
 - removing, 4-40
 - root-directed, 2-9
- Positioning, segment directories, 8-9
- Positioning files, 3-19
- Positioning to end-of-file, 7-13
- PRIMENET, RFA access, 2-7

Primitives, 6–6
 introduction, 4–2
 PRIMOS, return codes, 4–8
 PRIMOS file system objects, discussion,
 2–9
 –primos_direct_entries keyword, 5–17
 Private partitions, definition, A–3
 Processes
 multiple, 12–1, 12–5
 transaction management, 12–9
 PRWF\$\$ subroutine
 calling sequence, 7–15, 7–18, 7–31,
 7–32, 7–33, 7–34
 discussion, 7–30
 introduction, 7–2
 using, 7–38
 –public keyword, 5–17

Q

Q\$READ subroutine
 calling sequence, 11–2
 discussion, 11–1
 quota structure returned, 11–3
 Q\$SET subroutine, using, 11–3
 Quotas
 directory, 3–27
 disk, 11–1
 improving system performance, 11–4
 MFD, 11–3

R

R access right, definition, 4–6
 Random-access operations, calculating
 record position, 7–44
 RDLIN\$ subroutine
 calling sequence, 7–25
 discussion, 7–23
 introduction, 7–2
 sample routine, 7–27 to 7–29
 Read/write locks
 attributes, 3–24 to 3–25
 discussion, 3–16
 file, 12–1, 12–3
 system, 12–1
 Reading
 directories, 4–31
 files, 4–33

Global Mount Table, 4–34
 objects, 4–10
 Records
 calculating position, 7–44
 definition, 1–1, A–3
 fixed-length, 7–1, 7–2, 7–3, 7–42
 text, A–4
 variable-length, 7–1, 7–2, 7–3, 7–41
 [referencing_dir] keyword, 5–19
 Registered EPFs, 5–17
 Remote File Access, 2–7
 Removing, portals, 4–40
 Return codes, introduction, 4–8
 Root directory
 attaching, 6–19
 characteristics, 2–12
 defined, 2–5
 definition, A–3
 discussion, 2–10
 entries, 2–12
 Root entry, definition, 2–14
 Root-directed portals, definition, 2–9,
 A–4
 RWLOCK directive, 12–1, 12–2

S

SAM files, 3–5
 SATR\$\$ subroutine
 calling sequence, 10–8
 formats, 10–10
 using, 10–7, 10–9, 10–11, 10–12
 Scanning
 directories, 8–37
 segment directories, 8–23
 Search lists
 accessing, 5–19
 default, 5–2
 introduction, 5–1
 setting, 5–10
 types, 5–4
 user-defined, 5–2, 5–4
 user-specified, 5–2
 Search rule keywords. *See* Keywords
 Search rules
 administrator, 5–3
 ATTACH\$, 5–5
 benefits, 5–2
 BINARY\$, 5–8
 COMMAND\$, 5–7
 creating, 5–9 to 5–10
 ENTRY\$, 5–9
 file, 5–1
 INCLUDE\$, 5–8
 introduction, 5–1
 keywords, 5–12
 system, 5–3
 types, 5–3
 user-specified, 5–4, 5–9
 using subroutines, 5–20
 Segment directories
 accessing with subroutines, 8–2
 definition, 2–15
 deleting member files, 8–21
 discussion, 8–2
 dumped bit, 3–26
 extending, 8–13
 naming, 8–6
 opening, 8–3
 opening member files, 8–16
 positioning, 8–9
 scanning, 8–23
 size, 8–15
 SET_SEARCH_RULES command, 5–11,
 5–13
 Setting, search lists, 5–10
 SGD\$DL subroutine
 calling sequence, 8–22
 introduction, 8–2
 using, 8–21
 SGD\$EX subroutine, introduction, 8–2
 SGD\$OP subroutine
 calling sequence, 7–8, 8–17
 introduction, 7–2, 8–2
 using, 8–16
 SGDR\$\$ subroutine
 calling sequence, 8–11, 8–14, 8–25
 introduction, 8–2
 using, 8–9, 8–13, 8–16, 8–23
 Simple pathnames, discussion, 3–4
 Singly-rooted file system
 –added_disks keyword, 5–16
 discussion, 2–5
 Singly-rooted name space, definition,
 A–4
 Special/not-special attribute, 3–27
 Specific access rights, 4–17
 .SR suffix, 5–10

SRCH\$\$ subroutine, 6-23
 calling sequence, 6-24, 7-9, 8-5, 8-35
 example, 4-3
 introduction, 7-2, 8-28
 using, 7-45, 8-3, 8-32

SRSFX\$ subroutine
 calling sequence, 7-7, 8-4, 8-34
 discussion, 8-41
 introduction, 7-2, 8-2, 8-28
 using, 7-45, 8-3, 8-32

SSR command. *See*
 SET_SEARCH_RULES command

Static allocation, definition, 3-17
 -static_mode_libraries keyword, 5-17

Subroutines
 accessing files, 7-1
 accessing segment directories, 8-2
 AT\$-type, 6-4
 attaching, 6-7 to 6-8
 introduction, 4-2
 positioning segment directories, 8-9
 return codes, 4-8
 top-level directory attaching, 6-10
 using search rules, 5-20

System, search rules, 5-3
 -system keyword, 5-13

System read/write locks
 checking with CPL, 12-2
 setting, 12-1

T

Text files
 discussion, 7-45
 introduction, 7-1
 manipulating, 7-5
 variable-length, 7-23

Text records, definition, 1-1, A-4

Transaction files
 creating, 12-5
 multiple servers accessing, 12-6
 preventing reuse, 12-6
 status, 12-8

Tree hierarchy, multi-rooted, 2-2

Truncating, text files, 7-5

Truncating files, 3-19, 7-16

U

U access right, definition, 4-6

Uncompressed files, 7-3

Unqualified pathnames, discussion, 3-4

User-defined search lists, 5-4

User-specified search rules, 5-4

V

Variable-length files, format, 7-41

Variable-length records
 differing from fixed-length, 7-2
 discussion, 7-3, 7-41
 introduction, 7-1
 using with fixed-length, 7-4

Virtual memory file access. *See* VMFA

VMFA, definition, 3-14

W

W access right, definition, 4-6

Writing
 directories, 4-35
 files, 4-36
 objects, 4-11

WTLIN\$ subroutine
 calling sequence, 7-26
 discussion, 7-23
 introduction, 7-2
 sample routine, 7-27 to 7-29

X

X access right, definition, 4-6

Surveys

.....

Reader Response Form
Advanced Programmer's Guide II: File System
DOC10056-3LA

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate this document for overall usefulness?

excellent *very good* *good* *fair* *poor*

2. What features of this manual did you find most useful?

3. What faults or errors in this manual gave you problems?

4. How does this manual compare to equivalent manuals produced by other computer companies?

Much better *Slightly better* *About the same*
 Much worse *Slightly worse* *Can't judge*

5. Which other companies' manuals have you read?

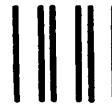
Name: _____

Position: _____

Company: _____

Address: _____

Postal Code: _____



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:



Attention: Technical Publications
Bldg 10
Prime Park, Natick, Ma. 01760

