HPSS

Programmer's Reference Guide, Volume 1

High Performance Storage System
Release 4.5
October 2002 (Revision 2)

© 1992-2002 International Business Machines Corporation, The Regents of the University of California, Sandia Corporation, Lockheed Martin Energy Research Corporation, and NASA Langeley Research Center.

All rights reserved.

Portions of this work were produced by the University of California,Lawrence Livermore National Laboratory (LLNL) under Contract No. W-7405-ENG-48 with the U.S. Department of Energy (DOE), by the University of California, Lawrence Berkeley National Laboratory (LBNL) under Contract No. DEAC03776SF00098 with DOE, by the University of California, Los Alamos National Laboratory (LANL) under Contract No. W-7405-ENG-36 with DOE, by Sandia Corporation, Sandia National Laboratories (SNL) under Contract No. DEAC0494AL85000 with DOE, and Lockheed Martin Energy Research Corporation, Oak Ridge National Laboratory (ORNL) under Contract No. DE-AC05-96OR22464 with DOE. The U.S. Government has certain reserved rights under its prime contracts with the Laboratories.

DISCLAIMER

Portions of this software were sponsored by an agency of the United States Government. Neither the United States, DOE, The Regents of the University of California, Sandia Corporation, Lockheed Martin Energy Research Corporation, nor any of their employees, makes any warranty, express or implied, or assumes any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights.

Printed in the United States of America.

HPSS Release 4.5 (Revision 2)
October 2002

High Performance Storage System is a registered trademark of International Business Machines Corporation.
IBM is a registered trademark of International Business Machines Corporation.
AIX and RISC/6000 are trademarks of International Business Machines Corporation.
Encina is a registered trademark of Transarc Corporation.
UNIX is a registered trademark of Unix System Laboratories, Inc.
Sammi is a trademark of Scientific Software Intercomp.
NFS and Network File System are trademarks of Sun Microsystems, Inc.
DST is a trademark of Ampex Systems Corporation.
ACLS is a trademark of Storage Technology Corporation.
Other brands and product names appearing herein may be trademarks or registered trademarks of third parties.

Table of Contents

Chapter 1 Overview 13

1.1 Client API 13
Purpose 13
Components 13
Constraints 16
Libraries 16
Environment Variables 16

1.2 Supplemental Data Transfer Functions 18
Purpose 18
Components 18
Constraints 19
Libraries 19

1.3 Non-DCE Client API 19
Purpose 19
Components 20
Constraints 20
Libraries 20
Environment Variables 20

1.4 64-bit Arithmetic Library 21
Purpose 21
Components 21
Constraints 22
Libraries 23

1.5 MPI-IO API 23
Purpose 23
Components 23
Constraints 24
Libraries 24
Environment Variables 25

1.6 Storage Concepts 25
Class of Service 25
Storage Class 26
Storage Hierarchy 26
File Family 26

1.7 User IDs 27

1.8 Access Control List API 27

Purpose 27
Components 28
Constraints 28
Libraries 28

1.9 DCE User Accounts 28

Chapter 2 Client API Functions 31

2.1 API Interfaces 31
hpss_Access 32
hpss_AcctCodeToName 33
hpss_AcctNameToCode 34
hpss_Chacct 35
hpss_ChacctByName 36
hpss_Chdir 37
hpss_Chmod 38
hpss_Chown 39
hpss_Chroot 40
hpss_ClientAPIReset 41
hpss_Close 42
hpss_Closedir 43
hpss_ConvertIdsToNames 43
hpss_ConvertNamesToIds 45
hpss_Create 46
hpss_DeleteACL 47
hpss_Fclear 49
hpss_FclearOffset 50
hpss_FileGetAttributes 50
hpss_FileGetXAttributes 51
hpss_FileSetAttributes 54
hpss_FilesetCreate 56
hpss_FilesetDelete 57
hpss_FilesetGetAttributes 58
hpss_FilesetListAll 59
hpss_FilesetSetAttributes 60
hpss_Fpreallocate 62
hpss_Fstat 64
hpss_Ftruncate 64
hpss_GetAcct 65
hpss_GetAcctName 66
hpss_GetACL 67
hpss_AsynchStatus 68
hpss_GetBFSStats 69
hpss_GetConfiguration 70
hpss_Getcwd 71
hpss_GetJunctions 72

hpss_GetListAttrs 73
hpss_JunctionCreate 74
hpss_JunctionDelete 75
hpss_Link 76
hpss_LoadThreadState 77
hpss_LoadDefaultThreadState 78
hpss_Lseek 79
hpss_Lstat 80
hpss_Migrate 81
hpss_Mkdir 82
hpss_Open 84
hpss_OpenBitfile 86
hpss_Opendir 87
hpss_Purge 88
hpss_PurgeLock 89
hpss_PurgeLoginContext 90
hpss_Read 91
hpss_ReadAttrs 92
hpss_Readdir 94
hpss_Readlink 95
hpss_ReadList 96
hpss_Rename 98
hpss_ReopenBitfile 99
hpss_Rewinddir 100
hpss_Rmdir 101
hpss_SetACL 102
hpss_SetAcct 104
hpss_SetAcctByName 104
hpss_SetBFSStats 106
hpss_SetConfiguration 106
hpss_SetCOSByHints 107
hpss_SetFileOffset 108
hpss_SetLoginContext 110
hpss_SiteIdToName 111
hpss_SiteNameToId 112
hpss_Stage 113
hpss_StageCallBack 114
hpss_Stat 115
hpss_Statfs 116
hpss_Statvfs 117
hpss_Symlink 118
hpss_ThreadCleanUp 119
hpss_Truncate 120
hpss_Umask 121
hpss_Unlink 122
hpss_UpdateACL 123

hpss_Utime 125
hpss_Write 126
hpss_WriteList 127
hpss_XLoadThreadState 129
free_ior_mem 130

2.2 Non-DCE Client API Specific Interfaces 131
hpss_PVRetrievals 131

2.3 Data Definitions 132
File Creation Hint Structure - hpss_cos_hints_t 132
Class of Service Priorities - hpss_cos_priorities_t 134
Class of Service Metadata Structure - hpss_cos_md_t 136
File Attribute Structure - hpss_fileattr_t 138
Extended File Attribute Structure - hpss_xfileattr_t 139
Name Server Attribute Structure - ns_Attrs_t 139
Name Server Fileset Attributes Structure – ns_FilesetAttrs_t 144
Name Server Object Handle Structure - ns_ObjHandle_t 145
Name Server Directory Entry - ns_DirEntry_t 147
Bitfile Volatile and Metadata Attributes - bf_attrib_t 147
Bitfile Volatile and Metadata Extended Attributes - bf_xattrib_t 148
Bitfile Metadata Attributes - bf_attrib_md_t 149
Bitfile Owner Record - bfs_owner_rec_t 151
Bitfile Server Storage Class Attributes - bf_sc_attrib_t 152
Bitfile Server Virtual Volume Attributes - bf_vv_attrib_t 153
Storage Server Physical Volume Attributes - pv_list_element_t 154
Storage Server Physical Volume Attributes Conformant Array - pv_list_t 154
Bitfile Server Statistics - bfs_stats_t 155
Account Record - acct_rec_t 156
API Configuration Structure – api_config_t 156
Name Server ACL Conformant Array - ns_ACLConfArray_t 159
Name Server Access Control List Entry - ns_ACLEntry_t 159
Global Fileset Entry Structure – hpss_global_fsent_t 160
Name Server Fileset Attribute Bits – ns_FilesetAttrBits_t 161
Name Server Object Attribute Bits – ns_FilesetAttrBits_t 161
Purge Lock Flag - purgelock_flag_t 161
API Name Specification – api_namespec_t 161
Bitfile Callback Address – bfs_callback_addr_t 162
HPSS Directory Entry – hpss_dirent_t 163
HPSS Security User Credentials – hsec_UserCred_t 163
Security Password Entry – SecPWent_t 164

Chapter 3 I/O Descriptor (IOD) and I/O Reply (IOR) 167

3.1 I/O Descriptor Purpose 167

3.2 I/O Reply Purpose 167

3.3 I/O Descriptor Components **167**

3.4 I/O Reply Components **168**

3.5 Data Definitions **168**

I/O Descriptor (IOD) - IOD_t 169
Source/Sink Descriptor - srcsinkdesc_t 172
Address Structure - address_t 173
I/O Reply (IOR) - IOR_t 182
Source/Sink Reply - srcsinkreply_t 186

Chapter 4 Supplemental Data Transfer Functions 189

4.1 API Functions **189**

Mover Socket (Parallel TCP/IP Data Transfer) Functions 189
Mover Protocol APIs 207
Mover Protocol Data Structures 213
Parallel Data Transfer Functions 217
Parallel Data Transfer Data Definitions 223
Network Options Functions 225
Network Options Data Definitions 226

Chapter 5 Math Library. 229

5.1 API Interfaces **229**

add64m 229
add64_3m 230
and64m 231
bld64m 232
cast64m 232
div64m 233
div2x64m 233
div_cl64m 234
div_2xcl64m 235
eqz64m 235
eq64m 236
ge64m 237
gt64m 237
high32m 238
le64m 239
low32m 239
lt64m 240
mod64m 240
mod2x64m 241
mul64m 242
neqz64m 242
not64m 243

or64m 244
shl64m 244
shr64m 245
sub64m 246
sub64_3m 246

5.2 Data Definitions 247
u_signed64 247
unsigned32 248

Chapter 6 MPI-IO API Functions 249

6.1 Application Programming Interfaces 249
File Manipulation 251
File Access 266
File Interoperability 315
File Consistency 319
Error Handling 323
File Hints 330

6.2 C++ Language Bindings 339

6.3 Fortran Language Bindings 346

6.4 Data Definitions 352
MPI-IO Standard Data Definitions 352

6.5 Troubleshooting 352
Compilation issues 353
Load/Link issues 354
Run-time errors 356
Thread-safety issues 356
Performance issues 357
Caching issues 357

Chapter 7 Site Interfaces 359

7.1 Gatekeeping 359
gk_site_Close 360
gk_site_Create 362
gk_site_CreateComplete 366
gk_site_CreateStats 366
gk_site_GetMonitorTypes 369
gk_site_Init 370
gk_site_Open 372
gk_site_OpenStats 373
gk_site_PassThru 374
gk_site_ReadSitePolicy 376
gk_site_Shutdown 378

gk_site_Stage 379
gk_site_StageComplete 380
gk_site_StageStats 381

7.2 Account Validation Site Interface 382
av_site_AcctIdxToName 383
av_site_AcctNameToIdx 386
av_site_Initialize 389
av_site_Shutdown 390
av_site_ValidateAccount 391
av_site_ValidateChacct 394
av_site_ValidateChown 397
av_site_ValidateCreate 400

Chapter 8 Access Control List API Functions 403

8.1 API Interfaces 403
hacl_ConvertACLToHACL 403
hacl_ConvertHACLToACL 404
hacl_ConvertHACLToString 404
hacl_ConvertHACLPermsToPerms 405
hacl_ConvertHACLTypeToType 406
hacl_ConvertPermsToHACLPerms 407
hacl_ConvertStringsToHACL 407
hacl_ConvertTypeToHACLType 408
hacl_DeleteHACL 409
hacl_GetHACL 410
hacl_SetHACL 411
hacl_SortHACL 412
hacl_UpdateHACL 413

8.2 Data Definitions 414
HACL-style Access Control List - hacl_acl_t 415
HACL-style Access Control List Entry - hacl_acl_entry_t 415

Appendix A Programming Examples. 417

Appendix B Makefile Example. 541

B.1 Main Makefile: 541

B.2 Include Makefile for AIX 4.2 and 4.3: 542

B.3 Include Makefile for Non-DCE AIX 4.2: 543

B.4 Include Makefile for Non-DCE AIX 4.3: 543

B.5 Include Makefile for Non-DCE Solaris 2.6: 543

B.6 Include Makefile for Solaris 2.6: 544

B.7 Include Makefile for Non-DCE IRIX 6.4: 544

Appendix C Notes 547

Appendix D Acronyms 559

Appendix E References. 561

Appendix F MPI-IO Programming Examples 563

Preface

This *High Performance Storage System (HPSS) Programmer's Reference Guide, Volume 1, Release 4.5*, documents client function calls which are provided by HPSS. It is designed for application programmers.

The document provides the programming reference for HPSS Release 4.5. In particular, the HPSS Client Application Programming Interfaces (APIs), 64-bit arithmetic library calls, Message Passing Interface - Input / Output (MPI-IO) interfaces, Site Interfaces, and ACL Interfaces are described. The 64-bit arithmetic library APIs are included since some Client APIs require 64-bit unsigned integer fields.

The objective of this document is to meet the following general goals:

- Define any known limitations of the APIs.
- Define the application programming interfaces (APIs) provided for use by other subsystems or clients.
- Define the data definitions referenced by the APIs.

Refer to the *HPSS User's Guide* for a description of the following command line interfaces: standard FTP, parallel FTP, NFS, IBM SP Parallel I/O File System Import / Export, DFS, and user utilities.

Refer to the *HPSS Error Messages Manual* for a list of all HPSS error and advisory messages which are output by the HPSS software. For each message, the following information is provided: message identifier and text, source file name(s) which generate the message, problem description, system action, and administrator action.

Refer to the *HPSS Installation Guide* and *HPSS Management Guide* for descriptions of the interfaces provided to HPSS administrators.

Refer to the *HPSS Programmer's Reference, Volume 2* for a description of application programming interfaces to the core HPSS servers. While it is envisioned that most users will access HPSS through the Client API, standard File Transfer Protocol (FTP), parallel FTP (PFTP), Network File System Version 2 or 3 (NFS), DFS, or XFS, well-defined programming interfaces are provided for each HPSS server. It should be noted that programming to the individual server level is a more complex programming model which requires a greater understanding of the HPSS servers. Volume 2 is appropriate for application / system's programmers with special needs (e.g. replacing the System's Management interface, adding a new Physical Volume Repository, etc.).

The *HPSS Programmer's Reference Guide, Volume 1* is structured as follows:

Chapter 1: Overview

This chapter provides an overview of each type of programmer interface, contraints, and required libraries.

Chapter 2: Client API

This chapter defines the Client API specifications and associated data definitions.

Chapter 3: I/O Descriptor (IOD) and I/O Reply (IOR)

This chapter describes the I/O Descriptor (IOD) and I/O Reply (IOR) Structures.

Chapter 4: Supplemental Data Transfer Functions

This chapter describes a set of support APIs to facilitate data transfers. Applications using hpss_ReadList and hpss_WriteList are potential users of these functions.

Chapter 5: Math Library

This chapter defines the 64-bit arithmetic library API specifications and associated data definitions.

Chapter 6: MPI-IO

This chapter defines the MPI-IO specifications and associated data definitions.

Chapter 7: Site Interfaces

This chapter defines the Account Validation and Gatekeeper Site Interface specifications and associated data definitions.

Chapter 8: ACL API

This chapter defines the specifications and associated data definitions for the Acces Control List (ACL) API.

Appendix A: Programming Examples

This appendix provides some example code for reading and writing HPSS files.

Appendix B: Makefile Examples

This appendix provides example Makefiles for building client applications on different platforms.

Appendix C: Notes

This appendix provides notes on IOD usage.

Appendix D: Acroynms

This appendix provides a list of acronyms used in this document.

Appendix E: References

This appendix lists documents cited in the text as well as other reference materials.

Appendix F: MPI-IO Programming Examples

This appendix provides example code for using the HPSS MPI-IO programming interface.

Typographic and Keying Conventions

This document uses the following typographic conventions:

Bold **Bold** words or characters represent system elements that you must use literally, such as functions, commands or keywords.

Italic *Italic* words or characters represent variable values to be supplied.

[ ] Brackets enclose optional items in syntax and format descriptions.

{ } Braces enclose a list of items to select in syntax and format descriptions.

Chapter 1 Overview

The High Performance Storage System (HPSS) provides scalable parallel storage systems for highly parallel computers as well as traditional supercomputers and workstation clusters. Concentrating on meeting the high end of storage system and data management requirements, HPSS is scalable and designed for large storage capacities, and to use network-connected storage devices to transfer data at rates up to multiple gigabytes per second. Listed below are the programming interfaces for accessing data from HPSS.

1.1 Client API

1.1.1 Purpose

The purpose of the Client API is to provide an interface which mirrors the POSIX.1 specification where possible to provide ease of use to the POSIX application programmer. In addition, extensions to allow the programmer to take advantage of the specific features provided by HPSS are provided (e.g., storage/access hints passed on file creation, parallel data transfers, migration, and purge).

1.1.2 Components

The Client API consists of these major parts:

- File Open/Creation and Close Operations
- File Data Access Operations
- Fileset/Junction Creation and Deletion Operations
- File Attribute Operations
- File Name Operations
- Directory Creation and Deletion Operations
- Directory Access Operations
- Working Directory Operations

- Client API Control Operations
- DCE Login Context Routines
- Bitfile Server Statistic Operations

File Open/Creation and Close Operations provide functions to create a file, open existing files and close previously opened files. The functions within this section are **hpss_Open**, **hpss_Close**, **hpss_Create**, **hpss_OpenBitfile** and **hpss_ReopenBitfile**.

File Data Access Operations provide functions to read from and write data to HPSS files. The functions within this section include **hpss_Lseek**, **hpss_Read**, **hpss_ReadList**, **hpss_SetFileOffset**, **hpss_Write**, and **hpss_WriteList**.

Fileset/Junciton Creation and Deletion Operation provide functions to create and delete filesets and junctions. Functions within this sections include **hpss_FilesetCreate**, **hpss_FilesetDeletes**, **hpss_FilesetGetAttributes**, **hpss_FilesetSetAttributes**, **hpss_FilesetListAll, hpss_JunctionCreate and hpss_JunctionDelete.**

File Attribute Operations include functions to query and alter a file's attribute values (both via POSIX consistent interfaces and extended HPSS interfaces), and determine a client's accessibility to a file or directory. Functions within this section are **hpss_Access**, **hpss_AcctCodeToName**, **hpss_AcctNameToCode, hpss_Chacct**, **hpss_ChacctByName, hpss_Chmod**, **hpss_Chown**, **hpss_ConvertIdsToNames, hpss_ConvertNamesToIds, hpss_DeleteACL**, **hpss_Fclear**, **hpss_FileGetAttributes**, **hpss_FileGetXAttributes**, **hpss_FileSetAttributes**, **hpss_Fstat**, **hpss_Ftruncate**, **hpss_GetAcct**, **hpss_GetAcctName, hpss_GetACL**, **hpss_GetListAttrs**, **hpss_Lstat**, **hpss_Migrate**, **hpss_Purge**, **hpss_SetACL**, **hpss_SetAcct**, **hpss_SetAcctByName, hpss_SiteIdToName, hpss_NameToId hpss_Stage**, **hpss_Stat**, **hpss_Statfs**, **hpss_Statvfs**, **hpss_Truncate**, **hpss_Umask**, **hpss_UpdateACL**, **hpss_Utime**, and **hpss_PurgeLock**.

File Name Operations provide functions to rename files and directories and remove a name associated with a file. Functions within this section include **hpss_Link**, **hpss_Readlink**, **hpss_Rename**, **hpss_Symlink**, and **hpss_Unlink**.

Directory Creation and Deletion Operations provide functions to make and remove directories. Functions within this section include **hpss_Mkdir** and **hpss_Rmdir**.

Directory Access Operations provide functions to read the directory entries from a directory. Functions within this section include **hpss_Closedir**, **hpss_Opendir**, **hpss_ReadAttrs**, **hpss_Readdir**, and **hpss_Rewinddir**.

Working Directory Operations provide functions to query and alter a thread's current working directory. Functions within this section include **hpss_Chdir**, **hpss_Chroot**, and **hpss_Getcwd**.

Client API Control Operations provide functions to update and clean up a thread's Client API state information. Functions within this section include **hpss_GetConfiguration**, **hpss_LoadThreadState, hpss_LoadDefaultThreadState**, **hpss_ThreadCleanUp**, and **hpss_SetConfiguration**. Also included is an important internal routine: **hpss_ClientAPIInit**. This API will not be used by most applications.

DCE Login Context Operations provide convenience functions to establish an application program's DCE login context, and subsequently purge the login context. An application program which is calling the Client API library must run on behalf of a DCE principal. Either the user can login to a DCE account prior to submitting the application program, or the **hpss_SetLoginContext** function may be called from the application. The name of the DCE principal and associated keytab file name are supplied to the function. Prior to exiting the application, the user must call **hpss_PurgeLoginContext** to delete the security context and terminate the thread which maintains the context.

Bitfile Server statistics operations provide the ability to get and reset the stage, migration, purge, and delete counts for the Bitfile Server. The functions that provide these capabilities are **hpss_GetBFSStats** and **hpss_SetBFSStats**.

1.1.3 Constraints

The following constraints are being imposed by the Client API:

- The validity of open files and directories at the time of **fork** is undefined in the child process.
- The validity of open files and directories is lost across calls to any of the family of **exec** calls.
- The designed client API works only with applications that make use of both the POSIX Name Server and the Bitfile Server. In particular, this API is not designed to meet the needs of clients that will perform the Name Server functionality internally and/or will bypass the Bitfile Server when performing storage operations.

1.1.4 Libraries

Applications issuing HPSS Client API calls must link the following library:

libhpss.a HPSS client library

In addition, the following libraries must be linked:

libEncina.a

libdce.a

1.1.5 Environment Variables

A description of environment variables used by the Client API is provided in this section. In most cases, explicit setting of these environment variables is only required if HPSS was installed with non-default values. Contact your administrator to determine the values being used or refer to the <hpss_directory>/config/hpss_env file (e.g. /opt/hpss/config/hpss_env) for the environment variable settings. The following environment variables can be used to control the Client API's working environment:

The **HPSS_LS_NAME** defines the CDS name of the Location Server RPC group entry for the HPSS system that the Client API will attempt to contact. The default is /.:/hpss/ls/group.

The **HPSS_MAX_CONN** identifies the integer value that will be used as the maximum number of allowed connections. The default is zero, which is equal to the default supported by the HPSS connection management software - currently 150.

The **HPSS_KTAB_PATH** defines the name of the file containing the DCE security keys necessary for successfully initializing the Client API. The default is /krb5/hpssclient.keytab.

The **HPSS_HOSTNAME** environment variable is used to specify the host name to be used for TCP/IP ports created by the Client API. The default value is the default host name of the machine on which the Client API is running. This value can have a significant impact on data transfer performance for data transfers that are handled by the Client API (i.e., those that use the hpss_Read and hpss_Write interfaces).

The **HPSS_TCP_WRITESIZE** environment variable is used to specify the amount of data to be written with each individual request to write data to a network connection during a data transfer. For some networks, writing less than the entire size of the client buffer has resulted in improved throughput. This environment variable may not affect the actual value used, based on the contents of the HPSS network options

file.

The **HPSS_TRANSFER_TYPE** environment variable is used to specify the data transport mechanism to be used for data transfers handled by the Client API. The value should be set to "TCP" (for TCP/IP transfers).

The **HPSS_PRINCIPAL** environment variable is used to specify the DCE principal to be used when initializing the HPSS security services. The default value is hpss_client_api. This variable is primarily intended for use by HPSS servers that utilize the Client API.

The **HPSS_SERVER_NAME** environment variable is used to specify the server name to be used when initializing the HPSS security services. The default value is "/.:/hpss/client". This variable is primarily intended for use by HPSS servers that utilize the Client API.

The Client API, if compiled with debugging enabled, uses two environment variables to control printing debug information. **HPSS_DEBUG**, if set to a non-zero value, will enable debug messages. By default, these messages will go to the standard output stream. If **HPSS_DEBUGPATH** is set, however, these messages will be directed to the file indicated by this environment variable. Two special cases for the debug path exist: "stdout" and "stderr", which will use the standard output or standard error I/O streams, respectively.

The **HPSS_DESC_NAME** enviornment variable is used to place a descriptive name in any HPSS messages logged by the Client API Library. The default value is “Client Application “. This variable is only used when logging is enabled in the library.

The **HPSS_BUSY_RETRIES** environment variable is used to control the number of retries to be performed when a request fails because the Bitfile Server does not currently have an available thread to handle that request. A value of zero indicates that no retries are to be performed. A value of negative one indicates that retries should be attempted until either the request succeeds or fails for another reason. The default value is 3.

The **HPSS_BUSY_DELAY** environment variable is used to control the number of seconds to delay between retry attempts. The default value is 15.

The **HPSS_RETRY_STAGE_INP** environment variable is used to control whether retries are attempted on opens of files in a Class of Service that is configured for background staging on open. A non-zero value indicates that opens which would return -EINPROGRESS to indicate that the file is being staged will be retried (using the same control mechanisms described in the previous paragraph). A value of zero indicates that the -EINPROGRESS return code will be returned to the client. The default value is zero.

The **HPSS_REUSE_CONNECTIONS** environment variable is used to control whether TCP/IP connections are to be left open as long as a file is opened or are to be closed after each read or write request. A non-zero value will cause connections to remain open, while a value of zero will cause connections to be closed. The default value is zero.

The **HPSS_USE_PORT_RANGE** environment variable is used to control whether the HPSS Mover(s) should use the configured port range when making TCP/IP connections for read and write requests. A non-zero value will cause the Mover(s) to use the port range. A value of zero will cause the Mover(s) to allow the operating system to select the port number.

The **HPSS_NUMRETRIES** environment variable is used to control the number of retries to attempt when an operation fails. Currently this class of operation includes library initialization and communications failures. A value of zero indicates that no retries are to be performed, and value of negative one indicates that the operation will be retried until successful. The default value is 4.

The **HPSS_TOTAL_DELAY** environment variable is used to control the number of total seconds to continue retrying requests. A value of zero indicates that no there is no time limit. The default value is 0.

The **HPSS_REGISTRY_SITE_NAME** environment variable is used to specify the name of the security registry used when inserting security information into connection binding handles. This is only needed when the client must support DFS in a cross-cell environment. The default registry is "/.../dce.clearlake.ibm.com"

The **HPSS_DMAP_WRITE_UPDATES** environment variable is used control the frequency of cache invalidates that are issued to the DMAPI file system while writing to a file that is mirror in HPSS. The default value is 20.

The **HPSS_GKTOTAL_DELAY** is used to control the total number of seconds to continue retrying a Gatekeeper request before the request times out. A value of zero indicates that there is no time limit. The default value is 600.

The **HPSS_LIMITED_RETRIES** is used to control the number of retry attempts before a limited retry error operation fails. The default value is 1

The **HPSS_DISABLE_CROSS_CELL** is used to control cross-cell traversal. When cross cell traversal is disabled, a client will not be allowed to access directories which are located in another DCE cell. The default value is 0.

The **HPSS_MNT_DISABLE_JUNCTIONS** is used to control mounting of exported paths which contain junctions. When this environment variable is set, it will not be possible to mount (via HPSS NFS) any exported path that contains a junction. The default value is 0 (unset).

The **HPSS_NFS_DISABLE_JUNCTIONS** is used to control junction traversal. When junction traversal is disabled, a client will not be allowed to access directories which require traversal to the directory via a DFS or HPSS junction. The default value is 0.

1.2 Supplemental Data Transfer Functions

1.2.1 Purpose

The purpose of the supplemental data transfer APIs is to provide a convenience library of functions for those clients supplying their own data transfer logic. For example, applications calling the hpss_ReadList or hpss_WriteList functions may want to use these APIs to handle their end of the data transfer.

1.2.2 Components

The supplement data transfer APIs consist of these major parts:

1.2.2.1 Mover Socket (Parallel TCP/IP Data Transfer)

The purpose of the Mover Socket library is to provide interfaces to send and receive parts of an HPSS parallel data transfer, when using TCP/IP as the data transport mechanism. Interfaces are provided for both the transfer responder (typically an HPSS Mover, which controls the order of the transfer) and the transfer

initiator (typically an HPSS client or Mover, which responds to requests made by the responder).

1.2.2.2 Mover Protocol

The purpose of the Mover Protocol library is to provide an interface that can be used to send and receive the various messages used by the HPSS Mover-to-Mover Protocol. The Mover Protocol allows a light-weight protocol that can be used for flow control during large data transfers, as well as supporting negotiated data transfer mechanisms and sizes of the current piece of the transfer.

1.2.2.3 Parallel Data Transfer

The purpose of the Parallel Data Transfer library is to provide an interface to send and receive the headers used by HPSS to delineate parts of a parallel data transfer when using TCP/IP as the data transport mechanism. The use of this library provides the transfer of data across many parallel data connections using multiple sockets and allows the data to be sent in the most efficient order. In other words, this library not only allows the data to be sent over parallel socket connections, but also allows the data to be sent in any order.

1.2.2.4 Network Options

The purpose of the Network Options library is to provide an interface to query the information contained in the HPSS network options configuration file for the local machine. This file contains information about network options that may be configured differently for specific network and/or nodes with which the local machine is communicating.

1.2.3 Constraints

The following constraints are being imposed by the supplemental APIs:

- Only 2 gigabytes of data may be transferred by any one library call.
- The client must supply a unique transfer identifier in the parallel data transfer calls to identify the data.

1.2.4 Libraries

The HPSS client libraries listed under 3.1.3 must be linked.

1.3 Non-DCE Client API

1.3.1 Purpose

The purpose of the Non-DCE Client API (NDAPI) is to provide an interface which mirrors the standard HPSS Client API specification to provide HPSS access to applications which run in environments lacking DCE and/or Encina.

1.3.2 Components

With the exception of ACL APIs, the NDAPI provides the same procedure calls as the standard Client API (See section 1.1.2).

In addition, the **hpss_PVRetrievals** function is provided to enable non-DCE clients to retrieve usage information about particular physical volumes.

1.3.3 Constraints

In addition to the constraints imposed by the standard Client API (See section 1.1.3), the following constraint is being imposed by the Non-DCE Client API:

- Client authentication is not yet supported. A client's HPSS/DCE identity is based on their Unix identity. Therefore, Unix and DCE UIDs should be consistent for client utilizing the Non-DCE Client API.

- Transactional support is not yet provided. An API returning with an EPIPE indicates a communications problem with the Non-DCE Client Gateway, between the time that the command was issued and the time the reply was received. In cases where this error is returned from a API that modify the state of an HPSS object, the failure or success of the operation can not be assumed and the state of the object should be queried before continuing.

1.3.4 Libraries

The Non-DCE Client API library has the same name as the standard Client API (See section 1.1.4).

libhpss.a	HPSS client library

However, when using the Non-DCE Client API, it is not necessary to link either **libEncina.a** or **libdce.a**.

1.3.5 Environment Variables

The Non-DCE Client API supports most of the environment variables supported by the standard Client API (See section 1.1.5) However, the following environment variables are not supported:

- **HPSS_LS_NAME**
- **HPSS_TRANSFER_TYPE**
- **HPSS_SERVER_NAME**
- **HPSS_DMAP_WRITE_UPDATES**

In addition to the standard Client API environment variables, the Non-DCE Client API supports the following environment variables:

The **HPSS_NDCG_NAME** environment variable is used to specify the server name to be used when initializing the HPSS security services. The default value is /.:/hpss/client.

The **HPSS_NDCG_TCP_PORT** environment variable defines the default port location for the Non-DCE Client Gateway with which the Non-DCE Client API will communicate. The value can be overridden by

appropriate entries in the HPSS_NDCG_SERVERS environment variable. The default value is 9590.

The **HPSS_NDCG_SERVERS** environment variable defines the name of the server on which the Non-DCE Client Gateway resides. Multiple servers may be separated by a colon (:). Also, it is possible to explicitly set the TCP port on a per server basis by following the server name with a forward slash (/) and a port number. The Non-Client API will randomly pick one of the specified entries to use as the gateways address. For example, a string "hpss/8002:pluto" would define two Non-DCE Client Gateways. One (hpss) uses an explicit port number, and the other (pluto) uses the value from the HPSS_NDCG_TCP_PORT.

The **HPSS_LOGGING_PORT** environment variable defines the port number of the Log Client to which log messages will be sent. The default value is 8001.

The **HPSS_LOGGING_TYPE** environment variable defines the types of messages to log. It consists of a list of log type strings, separated by colons (:). For example "CS_ALARM:CS_STATUS" would enable the logging of alarm and status messages. Valid log types are: CS_ALARM, CS_EVENT, CS_REQUEST, CS_SECURITY, CS_ACCOUNTING, CS_DEBUG, CS_TRACE, CS_STATUS. The default value is "CS_ALARM:CS_EVENT:CS_REQUEST:CS_SECURITY".

1.4 64-bit Arithmetic Library

1.4.1 Purpose

Some HPSS Client APIs require 64-bit fields. The operating system and C compiler on many workstation platforms may not support 64-bit integer operations. As a result, in order to support large integer fields, a set of math libraries have been supplied until 64-bit support is available on all pertinent vendor platforms.

1.4.2 Components

The Math Libraries consist of the following macros:

- **add64m** Add two 64-bit unsigned integers
- **add64_3m** Add two 64-bit unsigned integers and store the result in a separate parameter.
- **and64m** Perform a bitwise AND of two 64-bit unsigned integers
- **bld64m** Build a 64-bit unsigned integer from two 32-bit unsigned integers
- **cast64m** Cast a 32-bit unsigned integer into a 64-bit unsigned integer
- **div64m** Divide a 64-bit unsigned integer by a 32-bit unsigned integer
- **div2x64m** Divide a 64-bit unsigned integer by a 64-bit unsigned integer
- **div_cl64m** Divide a 64-bit unsigned integer by a 32-bit unsigned integer and return the ceiling
- **div_2xcl64m** Divide a 64-bit unsigned integer by a 64-bit unsigned integer and return the ceil-

ing

- **eqz64m** Determine if a 64-bit unsigned integer is zero
- **eq64m** Compare two 64-bit unsigned integers for equality
- **high32m** Extract the high order 32-bits of a 64-bit unsigned integer
- **le64m** Perform less than or equal to check between two 64-bit unsigned integers
- **low32m** Extract the low order 32-bits of a 64-bit unsigned integer
- **lt64m** Perform less than check between two 64-bit unsigned integers
- **ge64m** Perform greater than or equal to check between two 64-bit unsigned integers
- **gt64m** Perform greater than check between two unsigned 64-bit integers
- **mod64m** Modulus a 64-bit unsigned integer by a 32-bit unsigned integer
- **mod2x64m** Modulus a 64-bit unsigned integer by a 64-bit unsigned integer
- **mul64m** Multiply a 64-bit unsigned integer by a 32-bit unsigned integer
- **neqz64m** Determine if a 64-bit unsigned integer is nonzero
- **neq64m** Determine if two 64-bit unsigned integers are not equal
- **not64m** Perform a bitwise NOT of a 64-bit unsigned integer
- **or64m** Perform bitwise OR of two 64-bit unsigned integers
- **shl64m** Shift a 64-bit unsigned integer left by an unsigned 32-bit integer count
- **shr64m** Shift a 64-bit unsigned integer right by an unsigned 32-bit integer count
- **sub64m** Subtract a 64-bit unsigned integer from another 64-bit unsigned integer
- **sub64_3m** Subtract two 64-bit unsigned integers and store the result in a separate parameter.

1.4.3 Constraints

The following constraints are being imposed by the 64-bit arithmetic functions:

- 64-bit unsigned integer operations are sufficient, i.e. 64-bit signed arithmetic operations are not supported.
- Multiply functions are limited to 64-bit by 32-bit unsigned operations. For example, a 64-bit unsigned integer may be multiplied by a 32-bit unsigned integer. No 64-bit by 64-bit operations are supported for this category of functions.

1.4.4 Libraries

The 64-bit arithmetic functions are included in the libhpss libraries. Refer to Section 3.1.3 for a description of the libraries.

1.5 MPI-IO API

1.5.1 Purpose

The MPI-IO API is a subset of the MPI-2 standard. It gives applications written for a distributed memory programming model an interface that offers coordinated access to HPSS files from multiple processes. The interface also lets applications specify discontiguous patterns of access to files and memory buffers using the same "datatype" constructs that the Message-Passing Interface (MPI) offers. Files read and written through the HPSS MPI-IO can also be accessed through the HPSS Client API, so even though the MPI-IO subsystem does not offer all the migration, purging, and caching operations that are available in HPSS, parallel applications can still do these tasks through the HPSS Client API.

1.5.2 Components

The MPI-IO API consists of the following major categories:

- File Manipulation
- File Access
- File Interoperability
- File Consistency
- Error Handling
- File Hints

File Manipulation APIs are used to open (including create), close, or delete MPI-IO files, and to set or get characteristics of an open file, such as file size and file view. Some of these APIs are collective and some are noncollective; collective APIs require that all the processes that opened a file must participate in the operation. The File Manipulation APIs are: **MPI_File_open**, **MPI_File_close**, **MPI_File_delete**, **MPI_File_set_size**, **MPI_File_preallocate**, **MPI_File_get_size**, **MPI_File_get_group**, **MPI_File_get_amode**, **MPI_File_set_info**, **MPI_File_get_info**, **MPI_File_set_view**, **MPI_File_get_view**.

File Access APIs are used to read or write files. The APIs within this category may use either explicit offsets, individual (per-process) file pointers, or shared file pointers to specify the position in the file for reading or writing. These APIs may be either collective or noncollective. Furthermore, reads and writes may be either blocking or nonblocking.

The File Access APIs for explicit offset positioning include: **MPI_File_read_at**, **MPI_File_read_at_all**, **MPI_File_write_at**, **MPI_File_write_at_all**, **MPI_File_iread_at**, **MPI_File_iwrite_at**, **MPI_File_read_at_all_begin**, **MPI_File_read_at_all_end**, **MPI_File_read_at_all_end**, **MPI_File_write_at_all_begin**, and **MPI_File_write_at_all_end**.

The File Access APIs for individual file pointer positioning include: **MPI_File_read**, **MPI_File_read_all**, **MPI_File_write**, **MPI_File_write_all**, **MPI_File_iread**, **MPI_File_iwrite**, **MPI_File_read_all_begin**, **MPI_File_read_all_end**, **MPI_File_write_all_begin**, **MPI_File_Write_all_end**, **MPI_File_seek**, **MPI_File_get_position**, and **MPI_File_get_byte_offset**.

The File Access APIs for shared file pointer positioning include: **MPI_File_read_shared**, **MPI_File_write_shared**, **MPI_File_iread_shared**, **MPI_File_iwrite_shared**, **MPI_File_read_ordered**, **MPI_File_write_ordered**, **MPI_File_read_ordered_begin**, **MPI_File_read_ordered_end**, **MPI_File_write_ordered_begin**, **MPI_File_write_ordered_end**, **MPI_File_seek_shared**, and **MPI_File_get_position_shared**.

File Interoperability APIs are used to specify how file data must be converted when read or written, if the data representation in the file differs from that in the program. These APIs include: **MPI_File_get_type_extent** and **MPI_Register_datarep**.

File Consistency APIs are used to allow applications to coordinate accesses by multiple processes in order to guarantee the consistency of data in a file. These APIs include: **MPI_File_set_atomicity**, **MPI_File_get_atomicity**, and **MPI_File_sync**.

Error Handling APIs enable applications to modify the default MPI error handling facilities provided for files on a per-file-handle basis. These APIs include: **MPI_File_create_errhandler**, **MPI_File_set_errhandler**, **MPI_File_get_errhandler**, **MPI_File_call_errhandler**.

File Hints APIs are used to provide system specific information about a file, to enable MPI-IO to potentially optimize data accesses. These APIs include: **MPI_Info_create**, **MPI_Info_set**, **MPI_Info_delete**, **MPI_Info_get**, **MPI_Info_get_valuelen**, **MPI_Info_get_nkeys**, **MPI_Info_get_nthkey**, **MPI_Info_dup**, and **MPI_Info_free**.

1.5.3 Constraints

The following constraints are being imposed by the MPI-IO API:

- The host environment must provide a host MPI message passing library. That is, MPI-IO is currently layered over existing MPI-1 functionality. It is designed to be compatible with any MPI-2 implementation as well, although there may be redefinition conflicts in some MPI-2 environments.

- The MPI library must support multithreading; specifically, it must permit multiple threads within a process to issue MPI calls concurrently, subject to the limitations described in the MPI-2 standard.

- MPI applications must be compiled with the `mpio.h` header file to properly link with the MPI-IO library. The inclusion of **mpio.h** should replace the inclusion of the host MPI's **mpi.h**.

1.5.4 Libraries

Applications must be linked with the MPI-IO API library:

libmpioapi.a	HPSS MPI-IO library

In addition, the following HPSS library must be linked:

libhpss.a	HPSS client library

As with the HPSS client library, the library names for the Non-DCE versions are the same as the DCE libraries. Each HPSS configuration determines whether the DCE or Non-DCE versions are built.

There are platform-specific and MPI host-specific libraries on which the preceding libraries depend, so you must also link these libraries. Compilation and load/link issues are discussed in Sections 6.5.1 and 6.5.2.

1.5.5 Environment Variables

Any of the HPSS Client API environment variables may be used by an MPI-IO application. (Refer to the HPSS Client API environment variables, section 1.1.5). In particular, all MPI-IO applications need to specify an appropriate site-dependent value for:

HPSS_LS_NAME *HPSS_Location_Server*

If using the Non-DCE HPSS, an MPI-IO application will also need to specify which Non-DCE Gateway Server to use:

HPSS_NDCG_SERVERS *HPSS_NDCG_Server*

In addition, applications may wish to adjust the delay used by HPSS for busy retries. MPI-IO applications may issue concurrent HPSS accesses that result in some accesses having to wait while the file is busy. Since the HPSS client API sleeps before retrying, if the default wait period of 15 seconds is inappropriate, try adjusting **HPSS_BUSY_DELAY** to improve the application performance. For example, if the application is using concurrent, noncollective, small transfers to the same file, resetting the delay to 0 could result in better performance. On the other hand, if the application is using large transfers, resetting the delay to 30 could result in better performance.

When running a distributed MPI-IO application, use the following environment variables to guarantee the DCE login context will be available and consistent across the processes:

MPIO_LOGIN_NAME *login_name*

MPIO_KEYTAB_PATH *keytab_path*

where *login_name* is the DCE principal name used to create the keytab file, and *keytab_path* is the path name of the keytab file. Note that DCE authentication is used even with the non-DCE version of MPI-IO. See Section 6.6 for information on how to create a keytab file for MPI-IO authentication.

The following environment variable can be used to enable error messages to be issued from MPI-IO:

MPIO_DEBUG *any_nonzero_integer_value*

1.6 Storage Concepts

This section defines key HPSS storage concepts which have a significant impact on the usability of HPSS. Configuration of the HPSS storage objects and policies is the responsibility of your HPSS administrator.

1.6.1 Class of Service

Class of Service (COS) is an abstraction of storage system characteristics that allows HPSS users to select a particular type of service based on performance, space, and functionality requirements. Each COS describes a desired service in terms of characteristics such as minimum and maximum file size, transfer rate, access frequency, latency, and valid read or write operations. A file resides in a particular COS and the class is selected when the file is created. Underlying a COS is a storage hierarchy that describes how data for files in that class are to be stored in HPSS.

COS is specified at file create time. COS hints and priority structures are passed to HPSS in the **hpss_Open** function. Contact your HPSS administrator to determine the Classes of Service which have been defined. The following command may also be used to list the defined Classes of Service:

lshpss -cos

Refer to Chapter 5 of the *HPSS User's Guide* for information on the **lshpss** command. A class of service is implemented by a Storage Hierarchy of one to many Storage Classes. Storage Hierarchies and Storage Classes are not directly visible to the user, but are described below since they map to Class of Service. The relationship between storage class, storage hierarchy, and COS is shown in Figure 3-1.

1.6.2 Storage Class

An HPSS Storage Class is used to group storage media together to provide storage with specific characteristics for HPSS data. The attributes associated with a Storage Class are both physical and logical. Physical media in HPSS are called physical volumes. Physical characteristics associated with physical volumes are the media type, block size, the estimated amount of space on volumes in this class, and how often to write tape marks on the volume (for tape only). Physical media are organized into logical virtual volumes. This allows striping of physical volumes. Some of the logical attributes associated with the Storage Class are virtual volume block size, stripe width, data transfer rate, latency associated with devices supporting the physical media in this class, and storage segment size (disk only). In addition, the Storage Class has attributes that associate it with a particular migration policy and purge policy to help in managing the total space in the Storage Class.

1.6.3 Storage Hierarchy

An HPSS Storage Hierarchy consists of multiple levels of storage with each level representing a different storage media (i.e., a storage class). Files are moved up and down the Storage Hierarchy via stage and migrate operations, respectively, based upon storage policy, usage patterns, storage availability, and user request. For example, a Storage Hierarchy might consist of a fast disk, followed by a fast data transfer and medium storage capacity robot tape system, which in turn is followed by a large data storage capacity, but relatively slow data transfer tape robot system. Files are placed on a particular level in the hierarchy depending on the migration policy and staging operations. Multiple copies of a file may also be specified in the migration policy. If data is duplicated for a file at multiple levels in the hierarchy, the more recent data is at the higher level (lowest level number) in the hierarchy. Each hierarchy level is associated with a single storage class.

1.6.4 File Family

A file family is an attribute of an HPSS file that is used to group a set of files on a common set of tape virtual volumes. Release 4.1 supports grouping of files only on tape volumes. In addition, families can only be specified in Release 4.1 by associating a family with a fileset, and creating the file in the fileset. When a file is migrated from disk to tape, it is migrated to a tape virtual volume

assigned to the family associated with the file. If no family is associated with the file, the file is migrated to the next available tape not associated with a family (actually to a tape associated with family zero). If no tape virtual volume is associated with the family, a blank tape is reassigned from family zero to the file's family. The family affiliation is preserved when tapes are repacked. Configuring file families is a System Administrator function.

1.7 User IDs

After the HPSS system is configured, the necessary accounts must be created for HPSS users. Contact your HPSS administrator to add an account.

For Client API and MPI-IO access, a DCE account must be created. The administrator can use the following command to add a new DCE account. (Contact your HPSS administrator to add new DCE accounts.)

hpssuser -add *user* **-dce**

1.8 Access Control List API

1.8.1 Purpose

The access control list API is a set of routines for managing access control lists (ACLs). The routines provide a way to convert ACLs from string format into a form suitable for use by the client API routines. They also provide a way to call the client API routines using ACLs and string format, and a way to convert ACLs back from client API format to string format. In particular, the string conversion routines take care of translating user, group and cell names into UIDs, GIDs and Cell IDs respectively.

These APIs are supplied in a library named **libhacl.a**.

1.8.2 Components

The access control list library (libhacl) contains the following routines:

- **hacl_ConvertACLToHACL** - convert ACL to string format
- **hacl_ConvertHACLToACL** - convert ACL to HPSS format
- **hacl_ConvertHACLToString** - convert ACL to a form suitable for printing
- **hacl_ConvertHACLPermsToPerms** - convert permission string to HPSS format
- **hacl_ConvertHACLTypeToType** - convert ACL entry type to HPSS format
- **hacl_ConvertPermsToHACLPerms** - convert HPSS permission mask to string
- **hacl_ConvertStringsToHACL** - convert ACL strings to HACL format
- **hacl_ConvertTypeToHACLType** - convert ACL entry type to string
- **hacl_DeleteHACL** - delete selected entries from an object's ACL
- **hacl_GetHACL** - get an object's ACL
- **hacl_SetHACL** - replace an object's ACL with a new one
- **hacl_SortHACL** - sort ACL into canonical order
- **hacl_UpdateHACL** - change selected entries in an object's ACL

1.8.3 Constraints

These routines do not currently work with the non-DCE Client API library.

1.8.4 Libraries

The access control list APIs are available in in libhacl.a. Some of these routines also call client API functions defined in libhpss.a.

1.9 DCE User Accounts

As mentioned in the previous section, the Client API requires the user be logged into DCE. For MPI-IO, the user must provide a DCE login identity through a keytab facility, as described in Section 6.6.

The following command is used to issue a DCE login:

```
dce_login [principal_name] [password]
```

When this command is entered, the principal's identity is validated, and the network credentials are obtained. If *principal name* or *password* are not supplied, **dce_login** will prompt for them.

When the principal's DCE login context is no longer required, the following command may be used to destroy the login context and associated credentials:

```
kdestroy
```

Other DCE commands which might be of interest to the user are:

klist list the primary principal and tickets held in the DCE credentials cache

kinit Refresh a DCE credentials cache

Chapter 2 Client API Functions

This chapter specifies the HPSS client programming interface. Specifically, the following information is provided:

- Application Programming Interfaces (APIs)
- Data Definitions

2.1 API Interfaces

This section describes all API interfaces which are provided for use by another HPSS subsystem or by a client external to HPSS. The API interface specification includes the following information:

- Name
- Synopsis
- Description
- Parameters
- Return values
- Error conditions
- See also
- Notes

Note that for each thread that issues an HPSS Client API call, a call must be made to **hpss_ThreadCleanUp** with the thread id for that thread. This is necessary so that the client API can free state and memory allocated to that thread.

Note that there are a number of errors that may be returned from a Client API call which are not actually errors generated by performing the call, but are caused by a failure of the client API to successfully initialize. These values may be returned from any routine and include:

EAGAIN	An HPSS server is not ready or received a communication error, and the request could not be retried.
ENOCONNECT	The Client API could not connect to either the Location Server, Name Server or Bitfile Server.
ENOMEM	Memory could not be allocated for internal Client API State.
EPERM	The user's client credentials could not be established.
EIO	An internal HPSS error occurred.
ESTALE	The open file or directory is no longer valid - close and reopen the file or directory to reestablish a valid open descriptor. This error is likely due to the connections to the HPSS servers being reset.
ETIMEDOUT	An HPSS server request timed out or received a communication error, and the request could not be successfully retried.

2.1.1 hpss_Access

Purpose

Check file accessibility.

Synopsis

```
#include <unistd.h>
#include "hpss_api.h"

int
hpss_Access(
        char            *Path,          /* IN */
        int             Amode);         /* IN */
```

Description

The **hpss_Access** function checks the accessibility of the file named by Path for the file access indicated by Amode. Refer to POSIX.1 for more detailed information.

Parameters

Path	Points to the path name of the file for which client accessibility is being checked.
Amode	Indicates the type of file access being checked. Refer to POSIX.1 for possible values.

Return values

If the requested access is permitted, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value set by POSIX.1 access.

Error conditions

EACCES	The permissions specified by *Amode* are denied, or search permission is denied on a component of the path prefix.

EFAULT	The *Path* parameter is a NULL pointer.
EINVAL	An invalid value was specified for *Amode*.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.
ENOTDIR	A component of the *Path* prefix is not a directory.

See also

hpss_Chmod, **hpss_FileSetAttributes**.

Notes

None.

2.1.2 hpss_AcctCodeToName

Purpose

Finds the Account Name associated with the given Account Code

Synopsis

```
#include "hpss_api.h"

int

hpss_AcctCodeToName(

        acct_rec_t      AcctCode,       /* IN */
        uuid_t          *Site,          /* IN / OUT */
        char            *AcctName)      /* OUT */
```

Description

Finds and returns the Account Name associated with a given Account Code.

Parameters

AcctCode	The Account Code to look up
Site	Pointer to an area that contains the UUID of the site you are interested in. If this UUID is zeroed out or null, the index for the site managing the current working directory is used.
AcctName	The Account Name associated with the given Account Code. Account Name should be a string of at least HPSS_MAX_ACCOUNT_NAME (128) characters.

Return Values

Upon successful completion, hpss_AcctCodeToName returns zero. Otherwise, hpss_AcctCodeToName returns a negative value; the absolute value of that returned indicates the specific error.

Error Conditions

EINVAL	The given AccountCode is not valid.
EFAULT	The AcctName is a NULL pointer

See also

hpss_AcctNameToCode

Notes

None.

2.1.3 hpss_AcctNameToCode

Purpose

Finds the Account Code associated with the given Account Name

Synopsis

```
#include "hpss_api.h"

int

hpss_AcctNameToCode(

    char        *AcctName,      /* IN / OUT */
    uuid_t      *Site,          /* IN / OUT */
    acct_rec_t  *AcctCode)      /* OUT */
```

Description

Finds and returns the Account Code associated with a give Account Name. The AcctCode is returned as a string.

Parameters

AcctName	The Account Name to look up
Site	Pointer to an area that contains the UUID of the site you are interested in. If this UUID is zeroed out or null, the name of the site managing the current working directory is used.
AcctCode	The Account Code associated with the given Account Name

Return Values

Upon successful completion, hpss_AcctNameToCode returns zero. Otherwise, hpss_AcctNameToCode returns a negative value; the absolute value of that returned indicates the

specific error.

Error Conditions

EINVAL	The given Account Name is not valid.
EFAULT	The account name or account code is a NULL pointer.

See also

hpss_AcctCodeToName

Notes

None.

2.1.4 hpss_Chacct

Purpose

Change the account code of an HPSS file.

Synopsis

```
#include "hpss_api.h"

int
hpss_Chacct(
        char            *Path,          /* IN */
        acct_rec_t      AcctCode);      /* IN */
```

Description

The **hpss_Chacct** routine changes the accounting code for the file or directory named by *Path*.

Parameters

Path	Names the file for which the account code is being changed.
AcctCode	Specifies the new accounting code for the file.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which indicates the specific error.

Error conditions

EACCES	Search permission is denied on a component of the path prefix.
EFAULT	The *Path* parameter is a NULL pointer.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.
ENOTDIR	A component of the *Path* prefix is not a directory.

EPERM	The client does not have the appropriate privileges to perform the operation or is configured for Unix-style accounting.

See also

hpss_AcctCodeToName, hpss_AcctNameToCode, hpss_ChacctByName, hpss_GetAcct, hpss_GetAcctName, hpss_SetAcct, hpss_SetAcctByName, hpss_Chown, hpss_SetFileAttributes

Notes

None.

2.1.5 hpss_ChacctByName

Purpose

Change the account code of an HPSS file or directory by specifying the account's name.

Synopsis

```
#include "hpss_api.h"

int

hpss_Chacct(

    char        *Path,          /* IN */
    char        *AccountName)   /* IN */
```

Description

The hpss_ChacctByName routine changes the accounting code for the file or directory named by *Path*.

Parameters

Path	Names the file or directory for which the account code is being changed.
AccountName	The account name corresponding to the new accounting code for the file or directory.

Return Values

Upon successful completion, a value of zero is returned . Otherwise, a negative value is returned; the absolute value of that returned indicated the specific error.

Error Conditions

EACCES	Search permission is denied on a component of the path prefix.
EFAULT	The *Path* parameter is a NULL pointer.
ENAMETOOLONG	The length of the *Path* argument exceeds the system imposed limit,

	or a component of the pathname exceeds the system imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	The client does not have appropriate privilege to perform the operation or is configured for Unix-style accounting.
EINVAL	The specified *AccountName* is not a valid account name.

See also

hpss_GetAcct, hpss_SetAcct, hpss_Chown, hpss_SetFileAttributes, hpss_GetAcctName, hpss_SetAcctByName, hpss_Chacct

Notes

None.

2.1.6 hpss_Chdir

Purpose

Change current working directory.

Synopsis

```
#include "hpss_api.h"

int

hpss_Chdir(
        char                    *Path);             /* IN */
```

Description

The **hpss_Chdir** function changes a thread's current working directory to be the directory named by *Path*.

Parameters

Path	Specifies path name of the directory to which the current working directory is to be changed.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value set by POSIX.1 **chdir**.

Error conditions

EACCES	Search permission is denied on a component of the path name.
EFAULT	The *Path* parameter is a NULL pointer.

ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.
ENOTDIR	A component of the *Path* name is not a directory.

See also

hpss_Getcwd, **hpss_Chroot**.

Notes

None.

2.1.7 hpss_Chmod

Purpose

Change the file mode of an HPSS file.

Synopsis

```
#include "hpss_api.h"

int
hpss_Chmod(
        char            *Path,          /* IN */
        mode_t          Mode);          /* IN */
```

Description

The **hpss_Chmod** function alters the file mode associated with file named by Path.

Parameters

Path	Points to the path name of the file for which the file mode is being changed.
Mode	Specifies the new value to which the file mode is to be set.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value set by POSIX.1 **chmod**.

Error conditions

EACCES	Search permission is denied on a component of the path prefix.
EFAULT	The *Path* parameter is a NULL pointer.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.

ENOENT	The named file does not exist, or the *Path* argument points to an empty string.
ENOTDIR	A component of the *Path* prefix is not a directory.
EPERM	The client does not have the appropriate privileges to perform the operation.

See also

hpss_Chown, **hpss_Stat**, **hpss_FileGetAttributes**, **hpss_FileSetAttributes**.

Notes

None.

2.1.8 hpss_Chown

Purpose

Change owner and group of an HPSS File.

Synopsis

```
#include "hpss_api.h"

int
hpss_Chown(
        char        *Path,      /* IN */
        uid_t       Owner,      /* IN */
        gid_t       Group);     /* IN */
```

Description

The **hpss_Chown** function sets the user ID and group ID of the file named by *Path* to the values specified by *Owner* and *Group*, respectively.

Parameters

Path	Names the file for which the owner and group owner are being changed.
Owner	Specifies the new value for the owner of the file.
Group	Specifies the new value for the group owner of the file.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value set by POSIX.1 **chown**. If the owner is changed, the account code of the file or directory will also be changed to reflect that configured for the new owner.

Error conditions

EACCES	Search permission is denied on a component of the path prefix.
EFAULT	The *Path* parameter is a NULL pointer.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.
ENOTDIR	A component of the *Path* prefix is not a directory.
EPERM	The client does not have the appropriate privileges to perform the operation.

See also

hpss_Chmod, **hpss_Stat**, **hpss_FileGetAttributes**, **hpss_FileSetAttributes**.

Notes

None.

2.1.9 hpss_Chroot

Purpose

Change the root directory for the current client.

Synopsis

```
#include "hpss_api.h"

int

hpss_Chroot(
        char                    *Path);             /* IN */
```

Description

The **hpss_Chroot** function changes the root directory for the current client. After a successful call to **hpss_Chroot**, all absolute path name operations are done relative to *Path*, and relative operations cannot be made out of the subtree whose root is *Path*.

Parameters

Path	Specifies the path name of the directory that is to become the new root directory.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned,

the absolute value of which is equal to an errno value defined below.

Error conditions

EACCES	Search permission is denied on a component of the path name.
EFAULT	The *Path* parameter is a NULL pointer.
EINVAL	This call was made from the nonglobal Client API library.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.
ENOTDIR	A component of the *Path* name is not a directory.

See also

hpss_Chdir, **hpss_Getcwd**.

Notes

Note that as currently implemented, symbolic links could allow a client to access files outside the new root directory (since **hpss_Chroot** bookkeeping is maintained entirely in the client API but symbolic links are generally handled in the Name Server). If this is a problem (the only current projected client is the FTP server for anonymous FTP), changes could be made to ensure that the symbolic links do not access files outside the subtree - failing if they do, or possibly traversing the symbolic link contents within the client API.

2.1.10 hpss_ClientAPIReset

Purpose

Reset the current Client API control information.

Synopsis

```
#include "hpss_api.h"

void
hpss_ClientAPIReset(void);
```

Description

The **hpss_ClientAPIReset** routine will clean up the current Client API control information, including closing server connections. The next Client API call should then reinitialize the control information based on the current configuration information.

Parameters

None.

Return values

None.

Error conditions

None.

See also

hpss_GetConfiguration, **hpss_SetConfiguration**.

Notes

None.

2.1.11 hpss_Close

Purpose

Close a file.

Synopsis

```
#include "hpss_api.h"

int
hpss_Close(
        int                     Fildes);            /* IN */
```

Description

The **hpss_Close** function terminates the connection between the file handle, *Fildes*, and the file to which it is associated. The file handle and any associated resources are deallocated and can be reused by a subsequent call to **hpss_Open**.

Parameters

Fildes	Specifies the file handle obtained from a previous **hpss_Open**.

Return values

Upon successful completion, **hpss_Close** returns zero. Otherwise, **hpss_Close** returns a negative value; the absolute value of which is equal to an errno value set by POSIX.1 **close**.

Error conditions

EBADF	The specified file descriptor is out of range, or does not refer to an open file.
EBUSY	The file is currently in use by another client thread.

See also

hpss_Open, **hpss_OpenBitfile**, **hpss_ReopenBitfile**.

Notes

None.

2.1.12 hpss_Closedir

Purpose

Close an open directory stream.

Synopsis

```
#include "hpss_api.h"

int
hpss_Closedir(
        int                     Dirdes);            /* IN */
```

Description

The **hpss_Closedir** function closes the directory stream corresponding to the open directory stream handle *Dirdes*.

Parameters

Dirdes	Specifies the open directory stream handle corresponding to the stream to be closed.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value set by POSIX.1 *closedir*.

Error conditions

EBADF	The specified directory descriptor does not refer to an open directory.
EBUSY	The directory is currently in use by another client thread.

See also

hpss_Opendir.

Notes

None.

2.1.13 hpss_ConvertIdsToNames

Purpose

Convert UIDs to user names, GIDs to group names, and / or cell IDs to cell names.

Synopsis

```
int

hpss_ConvertIdsToNames(

    int              NumEntries,      /* IN */
    api_namespec_t   *Specs);         /* IN / OUT */
```

Description

The *hpss_ConvertIdsToNames* function converts a batch of Unix UIDs to user names, GIDs to group names, and / or cell ids to cell names. Each entry in the *Specs* array tells what kind of translation is needed for that entry. The function is provided to minimize the number of RPCs that must be used when a non-DCE client needs to translate ACLs.

Parameters

NumEntries	Number of entries in the Specs array.
Specs	Pointer to an area that contains information for defining a principal, which is to be converted, and the results of the conversion.

Return Values

Upon successful completion this routine returns zero. Otherwise it returns an error value describing the problem.

Error Conditions

EINVAL	The *Specs* array has an entry that contains a zero cell id.
ENOCONNECT	A problem with the security registry.
EFAULT	A name is too long to fit into specs
EAGAIN	Default for security registry error

See also

hpss_ConvertNamesToIds

Notes

The *Specs* array must contain NumEntries elements of type *api_name_spec_t*. Each array element has a *Type* field that determines how the element will be translated. If the *Type* is NAMESPEC_SKIP, then the *Id* and *CellId* fields will be ignored, and the *Name* and *CellName* fields will be set to undefined values. NAMESPEC_SKIP enables ACL editors to deal with ACL entry types, such as the mask object, which do not contain any ids.

If the *Type* is set to NAMESPEC_CELL, then the *CellId* field will be translated into a cell name, which is returned in the *CellName* field. In this case, the *Id* field will be ignored and the *Name* field will be set to an undefined value. On the other hand, if the *Type* is set to NAMESPEC_USER or NAMESPEC_GROUP, then the function returns the user or group name in the *Name* field as well as filling in the *CellName* field.

If, at any point during the translation process, an array entry is found that cannot be translated, the routine will return immediately. In this case, the *Name* and *CellName* fields of each *Specs* array entry should be considered undefined, but the *Type, Id* and *CellId* fields will remain unchanged from their initial values. Therefore the caller may need to keep a copy of the *Specs* array if the name fields will be needed again.

If the principal cannot be found in the desired cell, then the routine does not return an error, but rather just stores an ASCII representation of the principal's id in the *Name* field. Likewise, if the cell cannot be found in the trusted cell table, then the routine returns the cell id in the *CellName* field. This allows the caller to deal with principals and/or cells that no longer exist. The caller may detect this"error" by scanning for numeric data in these fields.

2.1.14 hpss_ConvertNamesToIds

Purpose

Convert user names to UIDs, group names to GIDs , and/or cell names to cell IDs.

Synopsis

```
int

hpss_ConvertNamesToIds(

    int                 NumEntries,         /* IN */
    api_namespec_t      *Specs);            /* IN/OUT */
```

Description

The *hpss_ConvertNamesToIds* function converts an array of user names to UIDs, group names to GIDs, and/or cell names to cell IDs. Each entry in the *Specs* array tells what kind of translation is needed for that entry. The function is provided to minimize the number of RPCs that must be used when a non-DCE client needs to translate ACLs.

Parameters

NumEntries	Number of entries in the *Specs* array.
Specs	Pointer to an area that contains information for defining a principal, which is to be converted, and the results of the conversion.

Return Values

Upon successful completion this routine returns zero. Otherwise it returns an error value describing the problem.

Error Conditions

ENOENT	One or more entries in the *Specs* array specified a principal or cell name that could not be found.
ENOCONNECT	A problem with the security registry.
EINVAL	Non-numeric principal id was specified.

See also

hpss_ConvertIdsToNames

Notes

The *Specs* array must contain NumEntries elements of type *api_name_spec_t*. Each array entry has a *Type* field that determines how that element will be translated. If the element's *Type* is NAMESPEC_SKIP, then the *Name* and *CellName* fields will be ignored and the *Id* and *CellId* fields will be set to undefined values. This is used to help ACL editors deal with ACL entry types, such as the mask object, which do not contain any names.

If the element's *Type* is set to NAMESPEC_CELL, the *CellName* field is translated into a cell id, which is returned in the *CellId* field. In this case, the *Name* field will be ignored and the *Id* field will be set to an undefined value. On the other hand, if *Type* is set to NAMESPEC_USER or NAMESPEC_GROUP, the function returns the UID or GID in the *Id* field as well as filling in the *CellId* field.

If the element's *Name* and / or *CellName* fields are strings or digits, the routine does not try to verify that the principal and / or cell actually exist. Rather, it just returns the corresponding values in the *Id* and *CellId* fields. This allows the caller to deal with principals and / or cells that no longer exist. In this situation, the routine does not return ENOENT.

If at any point during the translation process, an array entry is found that cannot be translated, the routine will return immediately. In this case, the *Id* and *CellId* fields of each *Specs* array entry should be considered undefined, but the *Type*, *Name*, and *CellName* fields will remain unchanged from their initial values. Therefore the caller may need to keep a copy of the *Specs* array if the id fields will be needed again.

2.1.15 hpss_Create

Purpose

Create an HPSS file.

Synopsis

```
#include "hpss_api.h"

int
hpss_Create(
        char                    *Path,          /* IN */
        mode_t                  Mode,           /* IN */
        hpss_cos_hints_t        *HintsIn,       /* IN */
        hpss_cos_priorities_t   *HintsPri,      /* IN */
        hpss_cos_hints_t        *HintsOut);     /* OUT */
```

Description

The **hpss_Create** function creates the specified file, if it does not already exist. The newly created or previously existing file is not opened (see **hpss_Open**).

Parameters

Path	Names the file to be opened or created.

Mode	Specifies the file mode used for determining the mode for the created file.
HintsIn	Points to an **hpss_cos_hints_t** structure which provides allocation hints to HPSS as to the expected structure or access of the file. This argument may be a NULL pointer.
HintsPri	Points to an **hpss_cos_priorities_t** structure which provides the relative priorities associated with the fields contained in the *HintsIn* structure. This arguement may be a NULL pointer.
HintsOut	Points to an **hpss_cos_hints_t** structure which will contain the values actually used when the file is created. This argument may be a NULL pointer.

Return values

Upon successful completion, **hpss_Create** returns zero. Otherwise, **hpss_Create** returns a negative value; the absolute value of which is equal to an errno value, defined below.

Error conditions

EACCES	Search permission is denied on a component of the *Path* prefix or the file does not exist and write permission is denied for the parent directory of the file to be created.
EEXIST	The named file exists.
EINVAL	One or more values in the *HintsIn* parameter is invalid.
ENAMETOOLONG	The length of the *Path* string exceeds the system-imposed path name limit or a path name component exceeds the system-imposed limit.
ENOSPC	Resources could not be allocated for the new file.
ENOTDIR	A component of the *Path* prefix is not a directory.

See also

hpss_Open, **hpss_Umask**.

Notes

This function differs from the POSIX **creat** in that no attempt is made to open the file and it behaves as if the O_EXCL flag were set (see EEXIST, above).

2.1.16 hpss_DeleteACL

Purpose

Removes entries from the Access Control List of a file.

Synopsis

```
#include "hpss_api.h"

int
hpss_DeleteACL(
        char                    *Path,          /* IN */
        unsigned32              Options,        /* IN */
        ns_ACLConfArray_t       *ACL);          /* IN */
```

Description

The **hpss_DeleteACL** function removes the ACL entries specified by *ACL* from the file named by *Path*.

Parameters

Path	Names the file for which the *ACL* is being removed.
Options	Bit vector used to specify what type of ACL is to be retrieved. One of: HPSS_ACL_OPTION_OBJ – return object's normal ACL. HPSS_ACL_OPTION_IO – return the initial-object ACL. (only valid for directory objects) HPSS_ACL_OPTION_IC – return the initial-container ACL. (only valid for directory objects)
ACL	Points to the list of ACL entries to be removed.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned.

Error conditions

EACCES	Search permission is denied on a component of the path prefix.
EFAULT	The *Path* or *ACL* parameter is a NULL pointer.
EINVAL	Exactly one of the HPSS_ACL_OPTION_* bits must be set in the *Options* bit vector to avoid receiving this error.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.
ENOTDIR	A component of the *Path* prefix is not a directory.
EPERM	The client does not have the appropriate privileges to perform the operation.

ESRCH	A specified ACL entry did not match an existing ACL entry for the file.

See also

hpss_GetACL, **hpss_SetACL**,**hpss_UpdateACL**.

Notes

This function is supported in the standard Client API library, but not in the non-DCE Client API library.

2.1.17 hpss_Fclear

Purpose

Clear part of a file.

Synopsis

```
#include "hpss_api.h"

int
hpss_Fclear(
        int             Fildes,         /* IN */
        u_signed64      Length);        /* IN */
```

Description

The **hpss_Fclear** routine clears part of an open file, specified by *Fildes*, the current file offset and *Length*. A hole will be created in the file covering the part of the file that was cleared, and its storage resource may be freed accordingly.

Parameters

Fildes	Specifies the file descriptor identifying the open file for which part is to be cleared.
Length	Specifies the number of bytes to be cleared.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EBADF	The specified file descriptor does not correspond to a file opened for writing.
EBUSY	The specified file descriptor is currently in use.

See also

hpss_FclearOffset, hpss_Truncate, **hpss_Ftruncate**.

Notes

None.

2.1.18 hpss_FclearOffset

Purpose

Clear part of a file beginning at the specified offset.

Synopsis

```
#include "hpss_api.h"

int
hpss_FclearOffset(
        int             Fildes,         /* IN */
        u_signed64      Offset,         /* IN */
        u_signed64      Length);        /* IN */
```

Description

The **hpss_FclearOffset** routine clears part of an open file, specified by *Fildes,* the current file *Offset* and *Length*. A hole will be created in the file covering the part of the file that was cleared, and storage resources may be freed accordingly.

Parameters

Fildes	Specifies the file descriptor identifying the open file of the part to clear.
Offset	Specifies where to begin clearing the file.
Length	Specifies the number of bytes to be cleared.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned; the absolute value of that returned is equal to an errno value defined below.

Error conditions

EBADF	The specified file descriptor does not correspond to a file opened for writing.
EBUSY	The specified file descriptor is currently in use.
EINVAL	The *Length* or *Offset* argument is invalid.

See also

hpss_Fclear.

Notes

None.

2.1.19 hpss_FileGetAttributes

Purpose

Get attributes for a file.

Synopsis

```
#include "hpss_api.h"

int
hpss_FileGetAttributes(
        char            *Path,          /* IN */
        hpss_fileattr_t *AttrOut);      /* OUT */
```

Description

The **hpss_FileGetAttributes** function returns the file attribute structure for the file named by *Path*. The attributes are returned in the structure pointed to by *AttrOut*.

Parameters

Path	Points to the path name of the file being queried.
AttrOut	Points to the structure that will hold the file attributes.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which indicates the specific error.

Error conditions

EACCES	Search permission is denied for a component of the path prefix.
EFAULT	The *Path* or *AttrOut* parameter is a NULL pointer.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.
ENOTDIR	A component of the *Path* prefix is not a directory.

See also

hpss_FileSetAttributes, **hpss_Stat**, **hpss_Fstat**, **hpss_Lstat**, **hpss_GetListAttrs**, **hpss_ReadAttrs**.

Notes

None.

2.1.20 hpss_FileGetXAttributes

Purpose

Get extended attributes for a file.

Synopsis

```
#include "hpss_api.h"

int
hpss_FileGetXAttributes(
        char            *Path,          /* IN */
        unsigned32      Flags,          /* IN */
        unsigned32      StorageLevel,   /* IN */
        hpss_xfileattr_t *AttrOut);     /* OUT */
```

Description

The **hpss_FileGetXAttributes** function returns the file extended attribute structure for the file named by *Path*. The file may currently be open, but is not required to be open. The attributes are returned in the structure pointed to by *AttrsOut*.

Parameters

Path — Points to the pathname of the file being queried.

Flags — Specifies the flag that indicates the behavior of the call. The acceptable values are:

API_GET_STATS_FOR_LEVEL - Returns bitfile attributes at the storage level specified by the StorageLevel argument.

API_GET_STATS_FOR_1STLEVEL - Returns bitfile attributes at the first storage level whether or not it contains bitfile data.

API_GET_STATS_FOR_OPTIMIZE - Returns only *StripeWidth* and *OptimumAccessSize* for storage level zero.

API_GET_STATS_ALL_LEVELS - Returns bitfile attributes across all storage class levels.

StorageLevel — Specifies the specific storage level to query when the API_GET_STATS_FOR_LEVEL flag is used.

AttrOut — Points to the structure that will hold the file attributes.

Return values

Upon successful completion, a value of zero is returned . Otherwise, a negative value is returned; the absolute value of that returned indicates the specific error.

Error conditions

EACCES	Search permission is denied for a component of the path prefix.
EFAULT	The *Path* or *AttrOut* parameter is a NULL pointer.
ENAMETOOLONG	The length of the *Path* argument exceeds the system imposed limit, or a component of the pathname exceeds the system imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.
ENOTDIR	A component of the path prefix is not a directory.

See also

hpss_FileSetAttributes, hpss_Stat, hpss_Fstat, hpss_Lstat, hpss_GetListAttrs, hpss_ReadAttrs.

Notes

This call allocates memory for the returned physical volume conformant array. After the successful completion of this call, the memory should be freed using code similar to the example below.

```
for(i=0;i<HPSS_MAX_STORAGE_LEVELS;i++)
{
   for(j=0;j<AttrOut.BFSAttr.SCAttrib[i].NumberOfVVs;j++)
   {
     if (AttrOut.BFSAttr.SCAttrib[i].VVAttrib[j].PVList != NULL)
     {
        free(AttrOut.BFSAttr.SCAttrib[i].VVAttrib[j].PVList);
      }
   }
}
```

2.1.21 hpss_FileSetAttributes

Purpose

Alter file attribute values.

Synopsis

```
#include "hpss_api.h"

int
hpss_FileSetAttributes(
        char            *Path,          /* IN */
        u_signed64      NSSelFlags,     /* IN */
        u_signed64      BFSSelFlags,    /* IN */
        hpss_fileattr_t *AttrIn,        /* IN */
        hpss_fileattr_t *AttrOut);      /* OUT */
```

Description

The **hpss_FileSetAttributes** function changes file attributes for the file named by *Path*, based on the attributes in the structure pointed to by *AttrIn*. The updated file attributes after the completion of the request are returned in the structure pointed to by *AttrOut*.

Parameters

Path — Points to the name of the file for which attribute values are to be changed.

NSSelFlags — Specifies the bitmask which indicates which attributes are to be set in the Name Server attribute.

ATTRINDEX_ACCOUNT
ATTRINDEX_ACL_MASK_PERMS
ATTRINDEX_BIT_FILE_ID
ATTRINDEX_CLASS_OF_SERVICE
ATTRINDEX_COMMENT
ATTRINDEX_COMPOSITE_PERMS
ATTRINDEX_EXPIRATION_DATE
ATTRINDEX_FILE_SIZE
ATTRINDEX_FOREIGN_PERMS
ATTRINDEX_GID
ATTRINDEX_GROUP_PERMS
ATTRINDEX_LINK_COUNT
ATTRINDEX_MAX_SEC_LABEL
ATTRINDEX_OTHER_PERMS
ATTRINDEX_SET_GID_ON_EXE
ATTRINDEX_SET_STICKY_BIT
ATTRINDEX_SET_UID_ON_EXE
ATTRINDEX_TIME_LAST_BILLED
ATTRINDEX_TIME_LAST_READ
ATTRINDEX_TIME_LAST_WRITTEN
ATTRINDEX_TIME_OF_METADATA_UPDATE
ATTRINDEX_TYPE
ATTRINDEX_UID
ATTRINDEX_UNAUTH_PERMS

ATTRINDEX_USER_PERMS

BFSSelFlags Bitmask which indicates which attributes are to be set in the Bitfile Server attributes:

BFS_SET_CURRENT_POSITION
BFS_SET_DATA_LEN
BFS_SET_CREATE_TIME
BFS_SET_MODIFY_TIME
BFS_SET_WRITE_TIME
BFS_SET_READ_TIME
BFS_SET_OWNER_REC
BFS_SET_COS_ID
BFS_SET_ACCT
BFS_SET_SECURITY
BFS_SET_REGISTER_BITMAP
BFS_SET_PREALLOC_LEN
BFS_SET_NO_PURGE_FLAG

AttrIn Points to a structure containing the new attribute values.

AttrOut Points to a structure that will contain the file attribute values after completion of this request.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which indicates the specific error.

Error conditions

EACCES Search permission is denied for a component of the path prefix.

EFAULT The *Path, AttrIn* or *AttrOut* parameter is a NULL pointer.

EINVAL An attribute value or selection flag is invalid.

ENAMETOOLONG The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.

ENOENT The named file does not exist, or the *Path* argument points to an empty string.

ENOSPC Resources could not be allocated to satisfy the request.

ENOTDIR A component of the *Path* prefix is not a directory.

EOPNOTSUPP The requested change is not supported.

EPERM The client does not have the appropriate privileges to change the file's attributes.

See also

hpss_FileGetAttributes, hpss_Chown, hpss_Chmod, hpss_Utime.

Notes

None.

2.1.22 hpss_FilesetCreate

Purpose

Create an HPSS fileset.

Synopsis

```
#include "hpss_api.h"

int
hpss_FilesetCreate(
        uuid_t                  *NameServer,            /* IN */
        ns_FilesetAttrBits_t    FilesetAttrBits,        /* IN */
        ns_FilesetAttrs_t       *FilesetAttrs,          /* IN */
        ns_AttrBits_t           ObjectAttrBits,         /* IN */
        ns_Attrs_t              *ObjectAttrs,           /* IN */
        ns_FilesetAttrBits_t    RetFilesetAttrBits,     /* IN */
        ns_AttrBits_t           RetObjectAttrBits,      /* IN */
        ns_FilesetAttrs_t       *RetFilesetAttrs,       /* OUT */
        ns_Attrs_t              *RetObjectAttrs,        /* OUT */
        ns_ObjHandle_t          *FilesetHandle);        /* OUT */
```

Description

The **hpss_FilesetCreate** function is called to create a new HPSS fileset. If a NULL NameServer UUID parameter is specified the root Name Server will be used. A handle to the newly created fileset is returned in the memory pointed to by FilesetHandle.

Parameters

NameServer	Points to the Name Server uuid to be used for the create.
FilesetAttrBits	Specifies which fileset attributes are to be set.
FilesetAttrs	Points to the fileset attributes to be set.
ObjectAttrBits	Specifies which object attributes are to be set.
ObjectAttrs	Points to the object attributes to be set.
RetFilesetAttrBits	Specifies which fileset attributes were set.
RetObjectAttrBits,	Specifies which object attributes were set.
RetFilesetAttrs	Points to the fileset attributes that were set.
RetObjectAttrs	Points to the object attributes that were set.

FilesetHandle	Points to the fileset handle created.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned; the absolute value of that returned is equal to an errno value.

Error conditions

EACCES	The user is not the root user or a trusted user.
EFAULT	The FilesetHandle, FilesetAttrs, ObjectAttrs, RetFilesetAttrs or the RetObjectAttrs parameter are NULL pointers.
EINVAL	The file attributes or attributes bits are invalid.
EEXIST	A file already exist with the specified identifier.

See also

hpss_FilesetDelete, **hpss_FilesetGetAttribute**, **hpss_FilesetSetAttributes.**

Notes

None.

2.1.23 hpss_FilesetDelete

Purpose

Delete and HPSS fileset.

Synopsis

```
#include "hpss_api.h"

int
hpss_FilesetDelete(
        char            *Name,          /* IN */
        u_signed64      *FilesetId,     /* IN */
        ns_ObjHandle_t  *FilesetHandle);/* IN */
```

Description

The **hpss_FilesetDelete** function is called to delete an existing HPSS fileset by either name, id or handle. A filesets can be identified by either a name, an ID, or the handle to its root. Only one type of identifier can be specified. The other values must be NULL pointers.

Parameters

Name	Specifies the name of the fileset to be deleted.
FilesetId	Specifies the id of the fileset to be deleted.

FilesetHandle	Specifies the handle of the fileset to be deleted.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned; the absolute value of that returned is equal to an errno value.

Error conditions

EACCES	The user is not the root user or a trusted user.
ENOENT	The specified fileset does not exist.
EINVAL	More that one type of fileset identifier was specified.
EFAULT	The Name, FilesetID and FilesetHandle arguments are all NULL pointers.

See also

hpss_FilesetCreate, **hpss_FilesetGetAttribute**, **hpss_FilesetSetAttributes**.

Notes

None.

2.1.24 hpss_FilesetGetAttributes

Purpose

Get attributes for an HPSS fileset.

Synopsis

```
#include "hpss_api.h"

int

hpss_FileGetSetAttributes(
        char                 *FilesetName,        /* IN */
        u_signed64           *FilesetId,          /* IN */
        ns_ObjHandle_t       *FilesetHandle,      /* IN */
        uuid_t               *NameServerUUID,     /* IN */
        ns_FilesetAttrBits_t  FilesetAttrBits,    /* IN */
        ns_FilesetAttrs_t    *FilesetAttrs);      /* OUT */
```

Description

The **hpss_FilesetGetAttributes** function is called to retrieve the attribute for a specified HPSS file set by supplying either a name, id or handle. Only one type of identifier can be specified. The other values must be NULL pointers. If NULL is specified for the Name Server UUID, then the local Name Server will be contacted; otherwise the specified Name Server will be contacted to retrieve the fileset information.

Parameters

Name	Specifies the name of the fileset to retrieve attributes for.
FilesetId	Specifies the id of the fileset to retrieve attributes for.
FilesetHandle	Specifies the handle of the fileset to retrieve attributes for.
NameServerUUID	The Name Server to contact managing the fileset information.
FilesetAttrBits	Specifies the fileset attribute bits that specify the fileset attributes to retrieve.
FilesetAttrs	Points to the returned fileset attributes.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned; the absolute value of that returned is equal to an errno value.

Error conditions

EACCES	The user is not the root user or a trusted user.
ENOENT	The specified fileset does not exist.
EINVAL	More that one type of fileset identifier was specified or invalid file set attribute bits were specified.
EFAULT	The Name, FilesetID and FilesetHandle arguments are all NULL pointers.

See also

hpss_FilesetSetAttributes, **hpss_FilesetCreate**, **hpss_FilesetDelete**.

Notes

None.

2.1.25 hpss_FilesetListAll

Purpose

Obtain a list of all the HPSS filesets.

Synopsis

```
#include "hpss_api.h"

int
hpss_FilesetListAll(
        u_signed64          OffsetIn,          /* IN */
        unsigned32          Entries,           /* IN */
```

```
unsigned32          *End,          /* OUT */
u_signed64          *OffsetOut,    /* OUT */
hpss_global_fsent_t *FSentPtr);    /* OUT */
```

Description

The **hpss_FilesetListAll** routine is called to get the global fileset attributes for all the filesets in the HPSS site.

Parameters

OffsetIn	The offset of the first fileset entry to be read. This should be set to zero to start before the first call, and subsequent entries can be read by providing the value returned in *OffsetOut*.
Entries	The number of the fileset entries referenced by *FSentPtr* in *hpss_global_fsent_t* entries, for which you have to allocate space.
End	Pointer to an area to contain an indication of whether the last fileset entry is included in the returned list. The value of *End* specifies if the end of the list was encountered before all the entries were accumulated.
OffsetOut	Pointer to an area to contain the offset of the next fileset entry following those returned by this call. *OffsetOut* is the location where the lookup stops when it has accumulated the specified number of fileset entries.
FSentPtr	Pointer to area to contain returned fileset entries.

Return values

Upon successful completion, a nonnegative value indicating the number of fileset entries is returned. Otherwise, a negative value is returned; the absolute value of that returned is equal to an errno value defined below.

Error conditions

EFAULT	The *End*, *OffsetOut* or *FSentPtr* parameter is a NULL pointer.
EINVAL	The *Entries* parameter specifies zero entries to read.

See also

hpss_FilesetCreate.

Notes

None.

2.1.26 hpss_FilesetSetAttributes

Purpose

Set attributes for an HPSS fileset.

Synopsis

```
#include "hpss_api.h"

int
hpss_FilesetSetAttributes(
        char                    *Name,                  /* IN */
        u_signed64              *FilesetId,             /* IN */
        ns_ObjHandle_t          *FilesetHandle,         /* IN */
        ns_FilesetAttrBits_t    FilesetAttrBitsIn,      /* IN */
        ns_FilesetAttrs_t       *FilesetAttrsIn,        /* IN */
        ns_FilesetAttrBits_t    FilesetAttrBitsOut,     /* IN */
        ns_FilesetAttrs_t       *FilesetAttrsOut);      /* OUT */
```

Description

The **hpss_FilesetSetAttributes** function is called to set the attribute for a specified Name Server file set by either name, id or handle. Only one type of identifier can be specified. The other values must be NULL pointers.

Parameters

Name	Specifies the name of the fileset to retrieve attributes for.
FilesetId	Specifies the id of the fileset to retrieve attributes for.
FilesetHandle	Specifies the handle of the fileset to retrieve attributes for.
FilesetAttrBits	Specifies the fileset attribute bits that specify the fileset attribute values that are to be set.
FilesetAttrs	Points to the fileset attribute values to be set.
FilesetAttrBitsOut	Specifies the fileset attribute bits that specify the fileset attribute values that are to be returned.
FilesetAttrsOut	Points to the returned fileset attribute values.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned; the absolute value of that returned is equal to an errno value.

Error conditions

EACCES	The user is not the root user or a trusted user.
ENOENT	The specified fileset does not exist.
EINVAL	More that one type of fileset identifier was specified or invalid file set attributes (or attribute bits) were specified.
EFAULT	The Name, FilesetID and FilesetHandle arguments are all NULL pointers.

See also

hpss_FilesetGetAttributes, **hpss_FilesetCreate**, **hpss_FilesetDelete**.

Notes

None.

2.1.27 hpss_Fpreallocate

Purpose

Set the length of a file and preallocate storage segments

Synopsis

```
#include "hpss_api.h"

int
hpss_Fpreallocate(
        int             Fildes,         /* IN */
        u_signed64      Length);        /* IN */
```

Description

The **hpss_Fpreallocate** routine sets the length of an open file, specified by the *Fildes* argument. The *Length* parameter specifies the requested length. It must be greater than the current size of the file. Additional storage space is preallocated for the file and a hole is created in the file from the current size to the requested length.

Parameters

Fildes	Speficies file descriptor identifying file to be queried.
Length	Specifies the desired length of the file.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned; the absolute value of that returned is equal to an errno value defined below.

Error conditions

EBADF	The specified file descriptor does not correspond to a file opened for writing.
EBUSY	The specified file descriptor is currently in use.
ENOSPC	The requested storage resources could not be allocated.
EINVAL	There is not a disk storage class at the top of the storage hierarchy.

See also

hpss_Preallocate, **hpss_Ftruncate, hpss_Truncate, hpss_Fclear, hpss_FileSetAttributes**.

Notes

There must be a disk storage class at the top of the storage hierarchy in which the file resides.

2.1.28 hpss_Fstat

Purpose

Get file status (POSIX).

Synopsis

```
#include "hpss_api.h"
int

hpss_Fstat(
        int             Fildes,         /* IN */
        struct stat     *Buf);          /* OUT */
```

Description

The **hpss_Fstat** function obtains information about the open file identified by *Fildes* and returns it in the structure pointed to by *Buf*. Refer to POSIX.1 for more detailed information.

Parameters

Fildes	Specifies the file descriptor identifying the file to be queried.
Buf	Points to a stat structure that will contain the information for the file.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value set by POSIX.1 **stat**.

Error conditions

EBADF	The file descriptor supplied does not correspond to an open file.
EFAULT	The *Buf* parameter is a NULL pointer.

See also

hpss_Chown, **hpss_Chmod**, **hpss_Utime**, **hpss_FileGetAttributes**, **hpss_FileSetAttributes**, **hpss_Stat**, **hpss_Lstat**, **hpss_GetListAttrs**, **hpss_ReadAttrs**.

Notes

None.

2.1.29 hpss_Ftruncate

Purpose

Set the length of a file.

Synopsis

```
#include "hpss_api.h"

int
hpss_Ftruncate(
        int             Fildes,         /* IN */
        u_signed64      Length);        /* IN */
```

Description

The **hpss_Ftruncate** routine sets the length of an open file, specified by the *Fildes* argument. If the new file length is less than the current length, the space allocated beyond the new length will be freed. If the new length is greater than the current length, a hole is created in the file.

Parameters

Fildes	Specifies the file descriptor identifying the open file for which the length is to be set.
Length	Specifies the new length of the file.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EBADF	The specified file descriptor does not correspond to a file opened for writing.
EBUSY	The specified file descriptor is currently in use.

See also

hpss_Truncate, **hpss_Fclear**, **hpss_FileSetAttributes**.

Notes

None.

2.1.30 hpss_GetAcct

Purpose

Query the default and current account codes.

Synopsis

```
#include "hpss_api.h"

int
hpss_GetAcct(
        acct_rec_t      *RetDefAcct,    /* OUT */
        acct_rec_t      *RetCurAcct);   /* OUT */
```

Description

The **hpss_GetAcct** routine returns the default and current account codes for the calling thread.

Parameters

RetDefAcct	Points to an area that will contain the default account code.
RetCurAcct	Points to an area that will contain the current account code.

Return values

Upon successful completion, **hpss_GetAcct** returns zero. Otherwise, **hpss_GetAcct** returns a negative value; the absolute value of which indicates the specific error.

Error conditions

EFAULT	The *RetDefAcct* or *RetCurAcct* parameter is a NULL pointer.

See also

hpss_AcctCodeToName, hpss_AcctNameToCode , hpss_Chacct, hpss_ChacctByName, hpss_GetAcctName, hpss_SetAcct, hpss_SetAcctByName

Notes

None.

2.1.31 hpss_GetAcctName

Purpose

Retrieve the current account name.

Synopsis

```
#include "hpss_api.h"

int

hpss_GetAcctName(

        char                    *AcctName)          /* OUT */
```

Description

The hpss_GetAcctName routine retrieves the name of the current session account for this thread. Since each site contacted by each thread in the Client API can have its own session account name, the account name for the site managing the current working directory is returned.

Parameters

AcctName	The name of the thread's current session account. Account name should be a string of at least HPSS_MAX_ACCOUNT_NAME (128) characters.

Return Values

Upon successful completion, hpss_GetAcctName returns 0. Otherwise, hpss_GetAcctName returns a negative value; the absolute value of that returned indicates the specific error.

Error Conditions

EINVAL	The client is configured for Unix-style accounting, and therefore the account name has no relevance. Or, a NULL AcctName was provided.

See also

hpss_GetAcct, hpss_Chacct, hpss_SetAcct, hpss_SetAcctByName, hpss_ChacctByName.

Notes

None.

2.1.32 hpss_GetACL

Purpose

Query the Access Control List of a file.

Synopsis

```
#include "hpss_api.h"

int
hpss_GetACL(
        char                *Path,          /* IN */
        unsigned32          Options,        /* IN */
        ns_ACLConfArray_t   **ACL);         /* OUT */
```

Description

The **hpss_GetACL** function returns the access control list information for the named file.

Parameters

Path	Names the file for which the *ACL* is being queried.
Options	Bit vector used to specify what type of ACL is to be retrieved. One of: HPSS_ACL_OPTION_OBJ – return object's normal ACL. HPSS_ACL_OPTION_IO – return the initial-object ACL. (only valid for directory objects) HPSS_ACL_OPTION_IC – return the initial-container ACL. (only valid for directory objects)
ACL	Points to the beginning of the returned list of ACL entries.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below:

Error conditions

EACCES	Search permission is denied on a component of the path prefix.
EFAULT	The *Path* or *ACL* parameter is a NULL pointer.
EINVAL	Exactly one of the HPSS_ACL_OPTION_* bits must be set in the *Options* bit vector to avoid receiving this error.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.
ENOTDIR	A component of the *Path* prefix is not a directory.
EPERM	The client does not have the appropriate privileges to perform the operation.

See also

hpss_SetACL, **hpss_DeleteACL**, **hpss_UpdateACL**.

Notes

This function is supported in the standard Client API library, but not in the non-DCE Client API library.

The use is responsible for freeing the ACL return parameter.

2.1.33 hpss_AsynchStatus

Purpose

Get status associated with a background request where callback info was previously provided.

Synopsis

```
#include "hpss_api.h"

int
hpss_AsynchStatus(
        signed32            CallBackId,         /* IN */
        hpssoid_t           *BfId,              /*IN*/
```

```
        signed32            *Status);          /*OUT*/
```

Description

The **hpss_GetAsynchStatus** routine queries the Root Name Server to obtain status information for the specified background request.

Parameters

CallBackId	Identifies the background request.
BfId	Bitfile identifier returned from the stage callback request.
Status	Returned stage status. This can be one of the following: HPSS_STAGE_STATUS_UNKNOWN - no status available. HPSS_STAGE_STATUS_QUEUED - request still in queue and not yet active. HPSS_STAGE_STATUS_ACTIVE - stage active and in progress.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EINVAL	The *BfId* parameter is not a valid pointer.
EFAULT	The *Status* parameter is a NULL pointer.

See also

hpss_StageCallBack, hpss_Stage.

Notes

None.

2.1.34 hpss_GetBFSStats

Purpose

Query Bitfile Server statistics.

Synopsis

```
#include "hpss_api.h"

int
hpss_GetBFSStats(
        bfs_stats_t         *StatsOut);        /* OUT */
```

Description

The **hpss_GetBFSStats** routine returns the number of stages, migrations, purges, and deletes.

Parameters

StatsOut	Points to an area that will contain the current BFS statistics values.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EFAULT	The *StatsOut* parameter is a NULL pointer.

See also

hpss_SetBFSStats.

Notes

None.

2.1.35 hpss_GetConfiguration

Purpose

Query the current Client API configuration information.

Synopsis

```
#include "hpss_api.h"

long
hpss_GetConfiguration(
        api_config_t            *ConfigOut);        /* OUT */
```

Description

The **hpss_GetConfiguration** routine returns the current configuration values for the Client API.

Parameters

ConfigOut	Points to an area that will contain the current configuration attribute value settings.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EFAULT	The *ConfigOut* parameter is a NULL pointer.

See also

hpss_SetConfiguration.

Notes

None.

2.1.36 hpss_Getcwd

Purpose

Get current working directory.

Synopsis

```
#include "hpss_api.h"

int
hpss_Getcwd(
        char            *Buf,           /* OUT */
        size_t          Size);          /* IN */
```

Description

The **hpss_Getcwd** function copies an absolute path name of the current working directory to the character array pointed to by *Buf*. The *Size* argument is the size in bytes of the array pointed to by *Buf*.

Parameters

Buf	Points to an array to contain the current working directory path name.
Size	Specifies the size, in bytes, of the array pointed to by *Buf*.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value set by POSIX.1 **getcwd**.

Error conditions

EACCES	Read or Search permission is denied on a component of the path name.
EFAULT	The *Buf* parameter is a NULL pointer.
EINVAL	The *Size* argument is zero.
ERANGE	The *Size* argument is greater than zero, but smaller than the length of the path name plus 1.

See also

hpss_Chdir, **hpss_Chroot**.

Notes

hpss_Getcwd is altered from POSIX.1 *getcwd* in that it returns an integer value to be more consistent with other HPSS calls.

2.1.37 hpss_GetJunctions

Purpose

Get a list of all junctions for the Root Name Server's name space.

Synopsis

```
#include "hpss_api.h"

int
hpss_GetJunctions(
        unsigned32          OffsetIn,          /* IN */
        unsigned32          Entries,           /* IN*/
        unsigned32          *End,              /* OUT */
        unsigned32          *OffsetOut,        /* OUT */
        hpss_junction_ent_t *JentPtr);         /* OUT */
```

Description

The **hpss_GetJunctions** routine is called to get an array of junctions that are being managed by the Root Name Server.

Parameters

OffsetIn — The offset of the first junction entry to be read. This should be set to zero to start before the first call, and subsequent entries can be read by providing the value returned in *OffsetOut*.

Entries — The number of the junction entires referenced by *JentPtr* in *hpss_junction_ent_t* entries, for which you have to allocate space.

End — Pointer to an area to contain an indication of whether the last junction entry is included in the returned list. The value of *End* specifies if the end of the list was encountered before all the entries were accumulated.

OffsetOut — Pointer to an area to contain the offset of the next junction entry following those returned by this call. *OffsetOut* is the location where the lookup stops when it has accumulated the specified number of junction entries.

JentPtr — Pointer to an area to contain the returned junction entries.

Return values

Upon successful completion, a nonnegative value indicating the number of junction entries read is returned. Otherwise, a negative value is returned; the absolute value of that returned is equal to an errno value defined below.

Error conditions

EFAULT	The *End*, *OffsetOut* or *JentPtr* parameter is a NULL pointer.
EINVAL	The *Entries* parameter specifies zero entries to read.

See also

hpss_FilesetCreate, hpss_JunctionCreateHandle, hpss_JunctionDelete.

Notes

None.

2.1.38 hpss_GetListAttrs

Purpose

Get attributes for a file, suitable for a directory listing.

Synopsis

```
#include "hpss_api.h"

int
hpss_GetListAttrs(
        char            *Path,          /* IN */
        ns_Attrs_t      *AttrOut);      /* OUT */
```

Description

The **hpss_GetListAttrs** function returns the attributes associated with the file named by *Path*. The attributes include information suitable for a long directory listing, including 64-bit file length and Class of Service.

Parameters

Path	Points to the path name of the file being queried.
AttrOut	Points to a stat structure that will contain the attribute information for the file.

Return values

Upon successful completion, a value of zero is returned . Otherwise, a negative value is returned, the absolute value of which indicates the specific error.

Error conditions

EACCES	Search permission is denied for a component of the path prefix.

EFAULT	The *Path* or *AttrOut* parameter is a NULL pointer.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.
ENOTDIR	A component of the *Path* prefix is not a directory.

See also

hpss_Chown, **hpss_Chmod**, **hpss_Utime**, **hpss_FileGetAttributes**, **hpss_FileSetAttributes**, **hpss_Stat**, **hpss_Fstat**, **hpss_GetListAttrs**, **hpss_ReadAttrs**.

Notes

None.

2.1.39 hpss_JunctionCreate

Purpose

Create a junction to an HPSS fileset or directory.

Synopsis

```
#include "hpss_api.h"

int

hpss_JunctionCreate(
        char                *Path,              /* IN */
        ns_ObjHandle_t      *SourceHandle,      /* IN */
        ns_ObjHandle_t      *JunctionHandle);   /* OUT */
```

Description

The **hpss_JunctionCreate** function is called to create a HPSS junction to the specified directory or fileset handle.

Parameters

Path	Specifies path name of the new junction.
SourceHandle	Points to the directory or fileset handle that is used for the source of the new junction.
JunctionHandle	Specifies the returned handle for the newly created junction.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned; the absolute value of that returned is equal to an errno value.

Error conditions

EFAULT	Either the *Path* or *SourceHandle* parameter is a NULL pointer.
ENAMETOOLONG	The length of the *Path* argument exceeds the system imposed limit, or a component of the pathname exceeds the system imposed limit.
ENOENT	The *Path* argument points to an empty string.
EEXISTT	The named path already exists in the HPSS name space.
EACCES	The requesting client is not the root user or a trusted user with write permissions.
EINVAL	The *SourceHandle* parameter doesn't point to a directory handle.

See also

hpss_FilesetCreate, **hpss_JunctionDelete**.

Notes

None.

2.1.40 hpss_JunctionDelete

Purpose

Delete a junction.

Synopsis

```
#include "hpss_api.h"

int
hpss_JunctionDelete(
        char                    *Path);             /* IN */
```

Description

The **hpss_JunctionDelete** is called to delete the junction specified by the *Path* input parameter.

Parameters

Path	Specifies the name of the junction to be deleted.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned; the absolute value of that returned is equal to an errno value.

Error conditions

ENOENT	The *Path* parameter is an empty string or doesn't refer to an existing object
EFAULT	The *Path* parameter is NULL.
EINVAL	The *Path* parameter doesn't refer to a junction.
EACCES	The requesting client is not the root user or a trusted user with write permissions.

See also

hpss_JunctionCreate, **hpss_JunctionCreateHandle**.

Notes

None.

2.1.41 hpss_Link

Purpose

Create a hard link to an existing HPSS file.

Synopsis

```
#include "hpss_api.h"

int
hpss_Link(
        char        *Existing,      /* IN */
        char        *New);          /* IN */
```

Description

The **hpss_Link** routine creates a hard link to an existing file (hard links to directories are not currently supported), given the path name of the existing file, *Existing*, and the path name of the new link, *New*.

Parameters

Existing	Specifies the path name of the existing file to which the link is to be created.
New	Specifies the path name of the new link.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned,

the absolute value of which is equal to an errno value defined below.

Error conditions

EACCES	Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the new link.
EEXIST	The object identified by *New* already exists.
EFAULT	The *Existing* or *New* parameter is a NULL pointer.
EPERM	The object specified by *Existing* is a directory.
EMLINK	The number of links to the file named by *Existing* would exceed the system-imposed limit.
ENAMETOOLONG	The length of the *Existing* or *New* argument exceeds the system-imposed path name limit or a path name component exceeds the system-imposed limit.
ENOENT	No entry exists for the specified file.
ENOTDIR	A component of the path prefix is not a directory.

See also

hpss_Symlink, **hpss_Unlink**.

Notes

None.

2.1.42 hpss_LoadThreadState

Purpose

Updates the user credentials and file/directory creation mask for the current thread's Client API state.

Synopsis

```
#include "hpss_api.h"

int
hpss_LoadThreadState(
        uid_t           UserID,         /* IN */
        mode_t          Umask);         /* IN */
```

Description

The hpss_LoadThreadState routine updates the user credentials and file/directory creation mask found in the current thread's Client API state.

Parameters

UserID	Specifies the user ID for the user whose credentials are to be loaded.
Umask	Specifies the new file/directory creation mask.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

ENOENT	Credentials for the specified user could not be obtained.

See also

hpss_XLoadThreadState, hpss_LoadDefaultThreadState

Notes

None.

2.1.43 hpss_LoadDefaultThreadState

Purpose

Special interface to allow well behaved clients to manipulate the global thread state so that all threads will use the new state.

Synopsis

```
#include "hpss_api.h"

int
hpss_LoadDefaultThreadState(
        uid_t       UserID,          /* IN */
        mode_t      Umask,           /* IN */
        char        *ClientFullName); /* IN */
```

Description

The **hpss_LoadDefaultThreadState** routine updates the global thread state so that all threads will use the new state. After this call, **hpss_LoadThreadState** routine is effectively disabled because for each new API the credentials are reloaded from the global thread state.

Parameters

UserID	Specifies the user ID for the user whose credentials are to be loaded.
Umask	Specifies the new file/directory creation mask.
ClientFullName	Specifies the client full quailified name in the following format /

	.../{dce cell name}/username (e.g., /.../dce.sandia.gov/jtjoker)

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

ENOENT	Credentials for the specified user could not be obtained.

See also

hpss_LoadThreadState, hpss_XLoadThreadState

Notes

None.

2.1.44 hpss_Lseek

Purpose

Set the current file offset for an open file, given a 32-bit value.

Synopsis

```
#include <unistd.h>
#include "hpss_api.h"

off_t
hpss_Lseek(
        int         Fildes,         /* IN */
        off_t       Offset,         /* IN */
        int         Whence);        /* IN */
```

Description

The **hpss_Lseek** function sets the file offset for the open file handle, *Fildes*. Refer to POSIX.1 for more detailed information.

Parameters

Fildes	Specifies the open file handle for which the file offset is to be set.
Offset	Specifies the number of bytes to be used in calculating the new file offset - dependent on the value of *Whence* as to the final effect on the new file offset.
Whence	Specifies how to interpret the *Offset* parameter in setting the file pointer associated with the *Fildes* parameter. Values for the *Whence* parameter are as follows:

SEEK_SET - file offset set to *Offset*.

SEEK_CUR - file offset set to current offset plus *Offset*.

SEEK_END - file offset set to current end of file plus *Offset*.

Return values

Upon successful completion, **hpss_Lseek** returns a nonnegative value representing the resulting offset as measured in bytes from the beginning the file. Otherwise, **hpss_Lseek** returns a negative value; the absolute value of which is equal to an errno value set by POSIX.1 **lseek**.

Error conditions

EBADF	The specified file descriptor does not refer to an open file.
EBUSY	The file is currently in use by another client thread.
EFBIG	Could not represent the resulting offset in the return value.
EINVAL	The *Whence* parameter is invalid or the resulting offset would be invalid.
ENOSPC	Resources could not be allocated to satisfy the request.

See also

hpss_Read, **hpss_Write**, **hpss_SetFileOffset**.

Notes

None.

2.1.45 hpss_Lstat

Purpose

Get file status (POSIX), returning status about a symbolic link if the named file is a symbolic link.

Synopsis

```
#include "hpss_api.h"

int
hpss_Lstat(
        char            *Path,          /* IN */
        struct stat     *Buf);          /* OUT */
```

Description

The **hpss_Lstat** function obtains information about the file named by Path and returns it in the structure pointed to by *Buf*. Refer to POSIX.1 for more detailed information. This function differs from **hpss_Stat**, however, in that if the named file is a symbolic link, information is returned about

the link itself, not about the file to which the link points.

Parameters

Path	Points to the path name of the file being queried.
Buf	Points to a stat structure that will contain the information for the file.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value set by POSIX.1 **stat**.

Error conditions

EACCES	Search permission is denied for a component of the path prefix.
EFAULT	The *Path* or *Buf* parameter is a NULL pointer.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.
ENOTDIR	A component of the *Path* prefix is not a directory.

See also

hpss_Chown, **hpss_Chmod**, **hpss_Utime**, **hpss_FileGetAttributes**, **hpss_FileSetAttributes**, **hpss_Stat**, **hpss_Fstat**, **hpss_GetListAttrs**, **hpss_ReadAttrs**.

Notes

None.

2.1.46 hpss_Migrate

Purpose

Migrate a file from a specified level in the storage hierarchy.

Synopsis

```
#include "hpss_api.h"

int
hpss_Migrate(
        int             Fildes,             /* IN */
        unsigned32      SrcLevel,           /* IN */
        unsigned32      Flags,              /* IN */
        u_signed64      *RetBytesMigrated); /* OUT */
```

Description

The **hpss_Migrate** routine migrates an open file from a level in the storage hierarchy, specified by *SrcLevel*. The *Flags* argument is used to control behavior of the request.

Parameters

Fildes	Specifies the file descriptor corresponding to the file to be migrated.
SrcLevel	Identifies the level in the storage hierarchy from which the data is to be migrated.
Flags	Controls the behavior of the migrate request. Anticipated values include: BFS_MIGRATE_ALL - migrate entire file (required).
RetBytesMigrated	Points to an area to contain the number of bytes migrated.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EBADF	The supplied file descriptor does not correspond to a file opened for writing.
EBUSY	The specified file descriptor is in use.
EFAULT	The *RetBytesMigrated* paramter is a NULL pointer.
EINVAL	The *Flags* argument is invalid.
EPERM	The client does not have the appropriate privileges to issue explicit file migration requests.

See also

hpss_Purge, **hpss_Stage**.

Notes

None.

2.1.47 hpss_Mkdir

Purpose

Create a directory.

Synopsis

```
#include "hpss_api.h"

int
hpss_Mkdir(
        char            *Path,          /* IN */
        mode_t          Mode);          /* IN */
```

Description

The **hpss_Mkdir** function creates a new directory with the name *Path*. The file permission bits of the new directory are initialized by *Mode* and modified by the file creation mask of the thread.

Parameters

Path	Specifies the path name to be used for the newly created directory.
Mode	Specifies permission bits to be used in setting the mode of the new directory.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value set by POSIX.1 **mkdir**.

Error conditions

EACCES	Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the directory to be created.
EEXIST	The named file exists.
EFAULT	The *Path* parameter is a NULL pointer.
EMLINK	The link count of the parent directory would exceed the maximum allowed number of links.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the pathname exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.
ENOTDIR	A component of the *Path* prefix is not a directory.

See also

hpss_Umask, **hpss_Rmdir**.

Notes

None.

2.1.48 hpss_Open

Purpose

Optionally create and open an HPSS file.

Synopsis

```
#include <fcntl.h>

#include "hpss_api.h"

int
hpss_Open(
        char                    *Path,          /* IN */
        int                     Oflag,          /* IN */
        mode_t                  Mode,           /* IN */
        hpss_cos_hints_t        *HintsIn,       /* IN */
        hpss_cos_priorities_t   *HintsPri,      /* IN */
        hpss_cos_hints_t        *HintsOut);     /* OUT */
```

Description

The **hpss_Open** function establishes the connection between a file, named by the *Path* argument, and a file handle. If O_CREAT is specified in *Oflag* and the file does not exist, an attempt will be made to create the file.

Parameters

Path

Names the file to be opened or created.

Oflag

Specifies the file status and file access modes to be assigned. Applicable values given below may be OR'ed together. Refer to POSIX.1 for specific behavior.

O_RDONLY
O_WRONLY
O_RDWR
O_APPEND
O_CREAT
O_EXCL
O_TRUNC

Mode

Specifies the file mode for a file that is created as a result of O_CREAT.

HintsIn

Points to an **hpss_cos_hints_t** structure which provides allocation hints to HPSS as to the expected structure or access of the file. This argument may be a NULL pointer. This parameter is only used during file creation.

HintsPri

Points to an **hpss_cos_priorities_t** structure which provides the relative priorities associated with the fields contained in the *HintsIn* structure. This parameter is only used during file creation, and may be a NULL pointer.

HintsOut	Points to an **hpss_cos_hints_t** structure which will contain the values actually used when the file is created. This argument may be a NULL pointer. This parameter is only used during file creation.

Return values

Upon successful completion, **hpss_Open** returns a nonnegative value that is the newly allocated file handle. Otherwise, **hpss_Open** returns a negative value; the absolute value of which is equal to an errno value set by POSIX.1 **open**.

Error conditions

EACCES	One of the following conditions occurred: Search permission is denied on a component of the path prefix. The file exists and the permissions specified by *Oflag* are denied. The file does not exist and write permission is denied for the parent directory of the file to be created. O_TRUNC is specified and write permission is denied.
EEXIST	O_CREAT and O_EXCL are set and the named file exists.
EFAULT	The *Path* parameter is a NULL pointer.
EINPROGRESS	The file is currently being staged. The open should be retried at a later time.
EINVAL	*Oflag* is not valid, or one or more values input in the *HintsIn* parameter is invalid.
EISDIR	The named file is a directory. Note that opening directories via **hpss_Open** is not supported in any mode.
EMFILE	The client open file table is already full.
ENFILE	Too many files are open in the system.
ENAMETOOLONG	The length of the *Path* string exceeds the system-imposed path name limit or a path name component exceeds the system-imposed limit.
ENOENT	The named file does not exist and the O_CREAT flag was not specified, or the *Path* argument points to an empty string.
ENOSPC	Resources could not be allocated for the new file.
ENOTDIR	A component of the *Path* prefix is not a directory.

See also

hpss_Close, **hpss_Umask**, **hpss_OpenBitfile**, **hpss_Create**, **hpss_ReopenBitfile**.

Notes

Note that opening directories with **hpss_Open** is not supported.

2.1.49 hpss_OpenBitfile

Purpose

Open an HPSS file, specified by bitfile ID.

Synopsis

```
#include "hpss_api.h"

int
hpss_OpenBitfile(
        hpssoid_t           *BitFileID,        /* IN */
        int                 OFlag,             /* IN */
        hsec_UserCred_t     *Ucred,            /* IN */
        gss_token_t         *AuthzTicket);     /* IN */
```

Description

The **hpss_OpenBitfile** routine attempts to open the bitfile identified by *BitFileID*. Note that this routine cannot be used to create a bitfile; rather, **hpss_Open** must be used for this purpose.

Parameters

BitFileID — Points to bitfile identifier. The BitfileID is usually obtained using **hpss_FileGetAttributes**.

Oflags — Specifies file status and file access modes to be assigned. Applicable values given below may be OR'ed together. Refer to POSIX.1 for specific behavior.

O_RDONLY
O_WRONLY
O_RDWR
O_APPEND
O_TRUNC

Ucred — Points to client's user credentials.

AuthzTicket — Points to client's authorization for this file.

Return values

Upon successful completion, a nonnegative file descriptor is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EACCES — The client does not have permission for the requested file access.

EFAULT	The *BitFileID* or *AuthzTicket* parameter is a NULL pointer.
EINPROGRESS	The file is currently being staged. The open should be retried at a later time.
EINVAL	*Oflag* is not valid.
EMFILE	The client file table is already full.
ENFILE	Too many files are already open in the system.
ENOENT	No entry exists for the specified bitfile ID.

See also

hpss_Open, **hpss_ReopenBitfile**, **hpss_Close**.

Notes

The user cannot generate a valid Authorization Ticket. Authorization Tickets are typically used by authorized clients. Unauthorized clients should pass a zero value for AuthzTicket.

2.1.50 hpss_Opendir

Purpose

Open an HPSS directory.

Synopsis

```
#include "hpss_api.h"

int
hpss_Opendir(
        char                    *DirName);          /* IN */
```

Description

The **hpss_Opendir** function opens a directory stream corresponding to the directory named by *DirName*. The directory stream is positioned at the first entry in the directory.

Parameters

DirName	Specifies the path name of the directory to be opened.

Return values

Upon successful completion, **hpss_Opendir** returns a nonnegative value that is the newly allocated directory stream handle. Otherwise, **hpss_Opendir** returns a negative value; the absolute value of which is equal to an errno value set by POSIX.1 **opendir**.

Error conditions

EACCES	Search permission is denied on a component of the path prefix, or

	read permission is denied on the directory itself.
EFAULT	The *DirName* parameter is a NULL pointer.
EMFILE	The open file table is already full.
ENAMETOOLONG	The length of the *DirName* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *DirName* argument points to an empty string.
ENOTDIR	A component of *DirName* is not a directory.

See also

hpss_Readdir, **hpss_Rewinddir**, **hpss_Closedir**.

Notes

The return value is changed from POSIX, primarily to make handling open directories and files in the client API consistent.

2.1.51 hpss_Purge

Purpose

Purge a piece of a file from a specified level in the storage hierarchy.

Synopsis

```
#include "hpss_api.h"

int
hpss_Purge(
        int             Fildes,             /* IN */
        u_signed64      Offset,             /* IN */
        u_signed64      Length,             /* IN */
        unsigned32      StorageLevel,       /* IN */
        unsigned32      Flags,              /* IN */
        u_signed64      *RetBytesPurged);   /* OUT */
```

Description

The **hpss_Purge** routine purges part of an open file, specified by *Fildes*, *Offset* and *Length* from a level in the storage hierarchy, specified by *StorageLevel*. The *Flags* argument is used to control behavior of the request.

Parameters

Fildes	Specifies the file descriptor corresponding to the file to be purged.
Offset	Specifies the offset of the start of the data to be purged. Currently must be 0.
Length	Specifies the length of the data to be purged. Currently must be 0.
StorageLevel	Identifies the level in the storage hierarchy from which the data is to be purged.
Flags	Controls the behavior of the purge request. Valid values include: BFS_PURGE_ALL - purge the entire file (required).
RetBytesPurged	Points to an area that will contain the number of bytes purged.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EBADF	The supplied file descriptor does not correspond to an open file.
EBUSY	The specified file descriptor is in use.
EFAULT	The *RetBytesPurged* parameter is a NULL pointer.
EINVAL	The *Flags*, *Offset* or *Length* argument is invalid.
EPERM	The client does not have the appropriate privileges to perform explicit purge operations.

See also

hpss_Migrate, **hpss_Stage**, **hpss_PurgeLock**

Notes

None.

2.1.52 hpss_PurgeLock

Purpose

Lock (or unlock) a file into the top level of its hierarchy.

Synopsis

```
#include "hpss_api.h"

int
hpss_PurgeLock(
```

int	*Fildes,*	/* IN */
purgelock_flag_t	*Flag*	/* IN */

Description

The **hpss_PurgeLock** routine either locks a file in the top level of its hierarchy from being purged or removes an existing lock.

Parameters

Fildes Specifies the file descriptor corresponding to the file to be locked/unlocked.

Flag Controls whether the request locks or unlocks the file. Possible values are:

PURGE_LOCK - Purge lock the file

PURGE_UNLOCK - Purge unlock the file.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EBADF The supplied file descriptor does not correspond to an open file.

EBUSY The specified file descriptor is in use.

ESTALE The connection for this entry is not valid.

See also

hpss_Purge.

Notes

None.

2.1.53 hpss_PurgeLoginContext

Purpose

Purge a login context and stop the refresh thread.

Syntax

```
#include <hpsscomm.h>

signed32

hpss_PurgeLoginContext(void);
```

Description

This routine purges the login context established by the routine **hpss_SetLoginContext**. It also stops the refresh thread that was maintaining the context.

Parameters

None.

Return values

Upon successful completion, a value of zero is returned. Otherwise, one of the error conditions below is returned.

Error conditions

HPSS_E_NOERROR	Successful completion
sec_login_s_context_invalid	login context was invalid
other	errno values from any pthread_* routine

See also

hpss_SetLoginContext.

Notes

None.

2.1.54 hpss_Read

Purpose

Read a contiguous section of an HPSS file, beginning at the current file offset, into a client buffer.

Synopsis

```
#include "hpss_api.h"

ssize_t
hpss_Read(
        int         Fildes,         /* IN */
        void        *Buf,           /* IN */
        size_t      Nbyte);         /* IN */
```

Description

The **hpss_Read** function attempts to read *Nbyte* bytes from the file associated with the open file handle, *Fildes*, into the client buffer pointed to by *Buf*.

Parameters

Fildes	Specifies the open file handle associated with the file from which data is to be read.

Buf	Point to a buffer where the data is to be placed.
Nbyte	Specifies the number of bytes to be read.

Return values

Upon successful completion, **hpss_Read** returns a nonnegative value that is the number of bytes read, including any holes encountered. Otherwise, **hpss_Read** returns a negative value; the absolute value of which is equal to an errno value set by POSIX.1 **read**.

Error conditions

EBADF	The specified file descriptor does not correspond to a file opened for reading.
EBUSY	The file is currently in use by another client thread.
EFAULT	The *Buf* parameter is out of range.
EIO	An input/output or HPSS internal error occurred.

See also

hpss_Open, **hpss_OpenBitfile**, **hpss_Write**, **hpss_Lseek**, **hpss_SetFileOffset**, **hpss_ReadList**, **hpss_WriteList**.

Notes

None.

2.1.55 hpss_ReadAttrs

Purpose

Read directory entries and optionally return entry attributes.

Synopsis

```
#include "hpss_api.h"

int
hpss_ReadAttrs(
        int             Dirdes,         /* IN */
        unsigned32      OffsetIn,       /* IN */
        unsigned32      BufferSize,     /* IN */
        unsigned32      GetAttributes,  /* IN */
        unsigned32      *End,           /* OUT */
        unsigned32      *OffsetOut,     /* OUT */
        ns_DirEntry_t   *DirentPtr);    /* OUT */
```

Description

The **hpss_ReadAttrs** routine returns a list of directory entries, which optionally includes file / directory attributes.

Parameters

Dirdes	Specifies the open directory stream handle corresponding to the directory being read.
OffsetIn	Specifies the starting directory offset. If zero, the list will be from the beginning of the directory.
BufferSize	Specifies the size of the buffer pointed to by *DirentPtr*, in bytes. The maximum number of entries that fit in the buffer will be returned, until the end of the directory is reached.
GetAttributes	Indicates, if nonzero, that attributes will be returned with the directory entries.
End	Points to an area that will contain an indication of whether the last entry is returned in the current list.
OffsetOut	Points to an area that will contain the directory offset at the end of the returned list. This value can be supplied to a subsequent call to continue with the next directory entry.
DirentPtr	Points to a buffer to hold the returned list of directory entries and attributes.

Return values

Upon successful completion, the return value indicates the number of directory entries in the returned list. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EBADF	The specified directory descriptor does not refer to an open directory.
EBUSY	The directory is currently in use by another client thread.
EFAULT	The *DirentPtr*, *End* or *OffsetOut* parameter is a NULL pointer.
EINVAL	The *BufferSize* parameter is zero.

See also

hpss_Opendir, **hpss_Closedir**.

Notes

Calling **hpss_ReadAttrs** does not affect the directory offset or cached directory entries that are manipulated via **hpss_Readdir** and **hpss_Rewinddir**.

2.1.56 hpss_Readdir

Purpose

Read a directory entry.

Synopsis

```
#include <dirent.h>
#include "hpss_api.h"

int
hpss_Readdir(
        int                 Dirdes,             /* IN */
        hpss_dirent_t       *DirentPtr);        /* OUT */
```

Description

The **hpss_Readdir** function returns a structure, *DirentPtr*, representing the directory entry at the current position in the open directory stream. Reference the *hpss_dirent_t* structure definition for more detailed information.

Parameters

Dirdes	Specifies the open directory stream handle corresponding to the directory being read.
DirentPtr	Points to a structure that will contain the directory entry information.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value set by POSIX.1 **readdir**.

Error conditions

EBADF	The specified directory descriptor does not refer to an open directory.
EBUSY	The directory is currently in use by another client thread.
EFAULT	The *DirentPtr* parameter is a NULL pointer.

See also

hpss_Opendir, **hpss_Rewinddir**, **hpss_Closedir**.

Notes

hpss_Readdir is altered from POSIX.1 *readdir* to be more consistent with other HPSS calls. These differences are that **hpss_Readdir** 1) accepts an integer directory stream handle (see **hpss_Opendir**) and 2) moves the returned structure pointer to the argument list rather than the return value.

When the end of the directory is encountered, the *d_name* field will be set to an empty string, and

the *d_namelen* field will be set to zero. These fields are in *DirentPtr* structure.

2.1.57 hpss_Readlink

Purpose

Read the value of a symbolic link (i.e., the data stored in the symbolic link).

Synopsis

```
#include "hpss_api.h"

int
hpss_Readlink(
        char        *Path,          /* IN */
        char        *Contents,      /* OUT */
        size_t      BufferSize);    /* IN */
```

Description

The **hpss_Readlink** routine returns the value of a symbolic link (not including any terminating null character) specified by *Path* into the buffer specified by *Contents*. The size of the buffer is specified by *BufferSize*.

Parameters

Path	Specifies the name of the symbolic link to be read.
Contents	Points to buffer to contain the value of the symbolic link.
BufferSize	Specifies the size of the buffer pointed to by *Contents*.

Return values

Upon successful completion, the length of the symbolic link name is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EACCES	Search permission is denied on a component of the path prefix, or read permission is denied on the symbolic link.
EFAULT	The *Path* or *Contents* parameter is a NULL pointer.
EINVAL	The specified file is not a symbolic link.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The specified path name does not exist.
ENOTDIR	A component of the *Path* prefix is not a directory.

ERANGE	The size of the *Contents* buffer is not big enough to contain the contents of the symbolic link or the value of the *BufferSize* parameter is zero

See also

hpss_Symlink.

Notes

None.

2.1.58 hpss_ReadList

Purpose

Read data from an HPSS file, specifying lists for data sources and sinks.

Synopsis

```
#include "hpss_api.h"

int
hpss_ReadList(
        IOD_t           *IODPtr,        /* IN */
        unsigned32      Flags,          /* IN */
        IOR_t           *IORPtr);       /* OUT */
```

Description

The **hpss_ReadList** function reads the file data specified by the source descriptor list in the IOD pointed to by *IODPtr* and moves the data to destinations specified by the sink descriptor list in the IOD. Results of the request will be returned in the structure pointed to by *IORPtr*. Refer to Chapter 3 for a description of the IOD.

Parameters

IODPtr->Function	Specifies the IOD function type. Set to **IOD_READ**.
IODPtr->SrcDescLength	Specifies the number of entries in the source descriptor list.
IODPtr->SinkDescLength	Specifies the number of entries in the sink descriptor list.
IODPtr->SrcDescList	Specifies a list of descriptors specifying the parts of the file to be read.
IODPtr->SinkDescList	Specifies a list of descriptors containing the destinations of the data.
Flags	Specifies bitmap containing flags modifying the operation of the read request. Valid values are: HPSS_READ_SEQUENTIAL - causes data to be read in order at

	the Bitfile Server level (i.e., at any point in time, the next byte in transfer order is being processed - not waiting for a byte later in transfer order).
IORPtr->RequestID	Specifies request identifier assigned to this request by the Client API.
IORPtr->Flags	Specifies status flags. Valid values are: IOR_COMPLETE - indicates the request has completed. IOR_ERROR - indicates an error was encountered. IOR_GAPINFO_VALID - indicates that the request specific reply structure returned in the IOR contains information describing a hole that was encounted during the read.
IORPtr->ReqSpecReply	Specifies information describing a gap in the file (a hole in which no data has been written) encountered during the read, if IOD_GAPINFO_VALID is set in the *IORPtr->Flags* field.
IORPtr->SrcReplyLength	Specifies the number of source reply descriptors.
IORPtr->SinkReplyLength	Specifies the number of sink reply descriptors.
IORPtr->SrcReplyList	Specifies a list of descriptors containing the data source results.
IORPtr->SinkReplyList	Specifies a list of descriptors containing the data sink results.

Return values

Upon successful completion, **hpss_ReadList** returns zero. Otherwise, **hpss_ReadList** returns a negative value; the absolute value of which indicates the specific error.

Error conditions

EBADF	A specified file descriptor in the source descriptor list does not correspond to a file opened for reading.
EBUSY	The file is currently in use by another client thread.
EFAULT	A memory buffer specified in the sink descriptor list is out of range.
EINVAL	A source descriptor did not specify a client file address, a sink descriptor specified an invalid address type, or *Flags* was invalid.
EIO	An input/output or HPSS internal error occurred.

See also

hpss_Open, **hpss_OpenBitfile**, **hpss_Read**, **hpss_Write**, **hpss_WriteList**, **free_ior_mem**.

Notes

Data will be transferred up to the point of where a gap is encountered.

Normally, the structure pointed to by the *IORPtr* parameter should be zeroed out, otherwise pointers in that structure will be used by the RPC mechanism as if they point to previously allocated memory.

After the client has completed using the reply information returned in the IOR, the pointers returned as part of the IOR should be freed using **rpc_ss_client_free**(). The following pointers must be freed (if non-NULL pointers are returned):

IORPtr->ReqSpecReply

IORPtr->SrcReplyList (each element in the list must be freed)

IORPtr->SinkReplyList (each element in the list must be freed)

Memory allocated to the returned I/O Reply can be freed by calling free_ior_mem(), and supplying the same IORPtr that was passed to the hpss_ReadList() call.

2.1.59 hpss_Rename

Purpose

Rename a file or directory.

Synopsis

```
#include "hpss_api.h"

int
hpss_Rename(
        char            *Old,           /* IN */
        char            *New);          /* IN */
```

Description

The hpss_Rename function changes the name of the file or directory currently named by *Old*, to *New*.

Parameters

Old	Specifies the path name that currently names the file or directory.
New	Specifies the path name to which the name is to be changed.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value set by POSIX.1 **rename**.

Error conditions

EACCES	Search permission is denied on a component of the path prefix, or one of the directories containing *Old* or *New* denies write permission, or write permission is required and denied for a directory pointed to by the *Old* or *New* arguments.

EFAULT	The *Old* or *New* parameter is a NULL pointer.
EISDIR	The *New* argument points to a directory, and the *Old* argument points to a file that is not a directory.
EMLINK	The file named by *Old* is a directory, and the link count of the parent directory of *New* already contains the maximum allowed number of links.
ENAMETOOLONG	The length of the *Old* or *New* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *Old* or *New* argument points to an empty string.
ENOTDIR	A component of the path prefix is not a directory.
ENOTEMPTY	The path named by *New* is a directory containing entries other than dot and dot-dot.

See also

hpss_Unlink.

Notes

None.

2.1.60 hpss_ReopenBitfile

Purpose

Given the bitfile ID, reopen an HPSS file using the same file table entry.

Synopsis

```
#include "hpss_api.h"

int
hpss_ReopenBitfile(
        int                 Fildes,             /* IN */
        hpssoid_t           *BitFileID,         /* IN */
        int                 Oflag,              /* IN */
        hsec_UserCred_t     *Ucred,             /* IN */
        gss_token_t         *AuthzTicket);      /* IN */
```

Description

The **hpss_ReopenBitfile** routine reopens the bitfile specified by *BitFileID*, using the file table entry currently associated with *Fildes*.

Parameters

Fildes	Specifies the file handle for the currently open file.
BitFileID	Points to the bitfile ID of the file to be reopened.
Oflags	Specifies the file status and file access modes to be assigned. Applicable values given below may be OR'ed together. Refer to POSIX.1 for specific behavior. O_RDONLY O_WRONLY O_RDWR O_APPEND O_TRUNC
Ucred	Points to client's user credentials.
AuthzTicket	Points to area where authorization ticket is to be returned.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EBADF	*Fildes* does not refer to an open file.
EBUSY	The open file descriptor is in use by another thread.
EFAULT	The *BitFileID* or *AuthzTicket* parameter is a NULL pointer.
EINPROGRESS	The file is currently being staged. The open should be retried at a later time.
EINVAL	Oflag does not contain a valid access mode.
ENOENT	No entry exists for the specified bitfile ID.

See also

hpss_Open, **hpss_OpenBitfile**.

Notes

If this routine fails, the file table entry identified by *Fildes* is not freed (it is marked as STALE), so that a subsequent effort can be made for this same file table entry.

2.1.61 hpss_Rewinddir

Purpose

Reset position of an open directory stream.

Synopsis

```
#include "hpss_api.h"

int
hpss_Rewinddir(
        int                     Dirdes);            /* IN */
```

Description

The **hpss_Rewinddir** function resets the position of an open directory stream corresponding to *Dirdes* to the beginning of that directory.

Parameters

Dirdes	Specifies the open directory stream handle for which the position is to be reset.

Return values

Upon successful completion, a value of zero is returned. If an error is encountered, a negative value is returned whose absolute value is described below.

Error conditions

EBADF	The specified directory descriptor does not correspond to an open directory.
EBUSY	Another client thread is currently using this directory descriptor.

See also

hpss_Opendir, **hpss_Readdir**, **hpss_Closedir**.

Notes

hpss_Rewinddir is altered from POSIX to return the values described above (whereas the POSIX *rewinddir* function does not return a value). Providing a failure indication was thought to be more important than strict POSIX compatibility.

2.1.62 hpss_Rmdir

Purpose

Remove an HPSS directory.

Synopsis

```
#include "hpss_api.h"

int
hpss_Rmdir(
        char                    *Path);             /* IN */
```

Description

The **hpss_Rmdir** function removes the directory named by *Path*. The directory will only be removed if the directory is empty.

Parameters

Path	Specifies the path name of the directory to be removed.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value set by POSIX.1 **rmdir**.

Error conditions

EACCES	Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the directory to be removed.
EBUSY	The named directory is currently in use and cannot be removed.
EFAULT	The *Path* parameter is a NULL pointer.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.
ENOTDIR	A component of the *Path* prefix is not a directory.
ENOTEMPTY	The named directory contains entries other than dot and dot-dot.

See also

hpss_Mkdir, **hpss_Unlink**.

Notes

None.

2.1.63 hpss_SetACL

Purpose

Set the Access Control List of a file.

Synopsis

#include "hpss_api.h"

int

```
hpss_SetACL(
        char                    *Path,          /* IN */
        unsigned32              Options,        /* IN */
        ns_ACLConfArray_t       *ACL);          /* IN */
```

Description

The **hpss_SetACL** function replaces the access control list for the file named by *Path*.

Parameters

Path	Names the file for which the *ACL* is being replaced.
Options	Bit vector used to specify what type of ACL is to be retrieved. One of: HPSS_ACL_OPTION_OBJ – return object's normal ACL. HPSS_ACL_OPTION_IO – return the initial-object ACL. (only valid for directory objects) HPSS_ACL_OPTION_IC – return the initial-container ACL. (only valid for directory objects)
ACL	Points to the new access control list for the file.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below:

Error conditions

EACCES	Search permission is denied on a component of the path prefix.
EFAULT	The *Path* or *ACL* parameter is a NULL pointer.
EINVAL	Exactly one of the HPSS_ACL_OPTION_* bits must be set in the *Options* bit vector to avoid receiving this error.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.
ENOTDIR	A component of the *Path* prefix is not a directory.
EPERM	The client does not have the appropriate privileges to perform the operation.

See also

hpss_DeleteACL, **hpss_GetACL**, **hpss_UpdateACL**.

Notes

This function is supported in the standard Client API library, but not in the non-DCE Client API library.

2.1.64 hpss_SetAcct

Purpose

Change the current account code.

Synopsis

```
#include "hpss_api.h"

int
hpss_SetAcct(
        acct_rec_t              NewCurAcct);        /* IN */
```

Description

The **hpss_SetAcct** routine changes the accounting code used when creating files and directories for the current thread.

Parameters

NewCurAcct	Specifies the value to be used for the account code for created files and directories.

Return values

Upon successful completion, **hpss_SetAcct** returns zero. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below:

Error conditions

EINVAL	The client is configured for Unix-style accounting, and therefore the account code cannot be changed.

See also

hpss_AcctCodeToName, hpss_AcctNameToCode , hpss_Chacct, hpss_ChacctByName, hpss_GetAcct, hpss_GetAcctName, hpss_SetAcctByName

Notes

None.

2.1.65 hpss_SetAcctByName

Purpose

Change the current account name.

Synopsis

#include "hpss_api.h"

int

hpss_SetAcctByName(

char **NewAcctName*) /* IN */

Description

The hpss_SetAcctByName routine changes the current session account name for the current site for this thread. Since each site contacted by each thread of the Client API will have its own session account, the account name returned is that of the site managing the current working directory.

Parameters

NewAcctName When using site-style accounting, this is the new session account name for the current site. However, when using UID accounting, *NewAcctName* can have one of several special meanings:

"" – means set the current session account to the UID of the user

String form of an account number (eg. "123") - means change the current session account index to 123.

User name (eg."smithj") – means use the UID of user "smithj" as the new current session account index.

Return Values

Upon successful completion, hpss_SetAcctByName returns 0. Otherwise, hpss_SetAcctByName returns a negative value; the absolute value of that returned indicates the specific error.

Error Conditions

ENOENT The NewAcctName specified is not valid.

EPERM The user does not have sufficient privilege to change the account name to NewAcctName, or NewAcctName is not defined at the site indicated by the client's current working directory.

See also

hpss_GetAcct, hpss_Chacct, hpss_SetAcct, hpss_GetAcctName, hpss_ChacctByName.

Notes

If the user's cell ID is foreign, then a message is returned advising that site-style accounting is required. If the NewAcctName is null, the the user's uid will be returned. If the NewAcctName is a number, then the same number will be returned. Otherwise, the account name is looked up in the DCE registry. If the entry is found, then the corresponding uid is returned. If the entry is not found, ENOENT is returned.

2.1.66 hpss_SetBFSStats

Purpose

Reset Bitfile Server statistics.

Synopsis

```
#include "hpss_api.h"

int
hpss_SetBFSStats(
        bfs_stats_t             *StatsOut);         /* OUT */
```

Description

The **hpss_SetBFSStats** routine resets the stage, migration, purge, and delete counts in the Bitfile Server and sets the last reset time to the current time.

Parameters

StatsOut	Points to an area that will contain the current BFS statistics values.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EFAULT	The *StatsOut* parameter is a NULL pointer.

See also

hpss_GetBFSStats.

Notes

None.

2.1.67 hpss_SetConfiguration

Purpose

Update the current Client API configuration information.

Synopsis

```
#include "hpss_api.h"
#include “api_internal.h”

long
hpss_SetConfiguration(
        api_config_t            *ConfigIn);         /* IN */
```

Description

The **hpss_GetConfiguration** routine updates the current configuration values for the Client API.

Parameters

ConfigIn	Points to a structure that contains the new configuration attributes value settings.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EFAULT	The *ConfigIn* parameter is a NULL pointer.
EINVAL	Invalid configuration attribute value setting.

See also

hpss_GetConfiguration, **hpss_ClientAPIReset**.

Notes

None.

2.1.68 hpss_SetCOSByHints

Purpose

Create an HPSS file.

Synopsis

```
#include "hpss_api.h"

int
hpss_SetCOSByHints(
        int                     Filedes,        /* IN */
        unsigned32              Flags,          /* IN */
        hpss_cos_hints_t        *HintsPtr,      /* IN */
        hpss_cos_priorities_t   *PrioPtr,       /* IN */
        hpss_cos_md_t           *COSPtr);       /* OUT */
```

Description

The **hpss_SetCOSByHints** routine is used to attempt to place a file in an appropriate Class of Service before any data has been written to that file. This interface is primarily used when the file size is not known at the time the file is created, but based on the knowledge of the file at the time of the first write a better Class of Service may be determined.

Parameters

Filedes	Specifies the open file handle for which the Class of Service is to be

	set.
Flags	Specifies flags which affect the processing of this request. Valid values are: BFS_RESET_SEGSIZE - Indicates that the request is only to set a new storage segment size.
HintsPtr	Points to a structure that provides attribute values for selection of a new Class of Service or storage segment size.
PrioPtr	Points to a structure that contains relative priorities for the attribute values indicated by the *HintsPtr* parameter.
COSPtr	Points to a structure that will contain the attribute values for the selected Class of Service and storage segment size.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value, defined below.

Error conditions

EBADF	The specified file descriptor does not refer to an open file.
EBUSY	The file is currently in use by another client thread, or the Bitfile Server could not complete the request at the current time.
EFAULT	One of *HinstPtr*, *PrioPtr*, or *COSPtr* is a NULL pointer.
EINVAL	The specified COS hints are invalid.
EPERM	The client does not have the appropriate privileges to perform the request.
ENOTDIR	A component of the *Path* prefix is not a directory.

See also

hpss_Open, **hpss_OpenBitfile**.

Notes

None.

2.1.69 hpss_SetFileOffset

Purpose

Set the current file offset for an open file, given a 64-bit offset value.

Synopsis

```
#include <unistd.h>
#include "hpss_api.h"

int
hpss_SetFileOffset(
        int             Fildes,         /* IN */
        u_signed64      OffsetIn,       /* IN */
        int             Whence,         /* IN */
        int             Direction,      /* IN */
        u_signed64      *OffsetOut);    /* OUT */
```

Description

The **hpss_SetFileOffset** function sets the file offset for the open file handle, *Fildes*. Refer to the POSIX.1 *lseek* function for more detailed information. Both input and output offset values are 64-bit values, to provide accessibility to the full range of HPSS file sizes. Note that since the *OffsetIn* value is unsigned, the*Direction* parameter is provided to specify whether *OffsetIn* should be used to move forward or backward in the file.

Parameters

Fildes
Specifies the open file handle for which the file offset is to be set.

OffsetIn
Specifies the number of bytes to be used in calculating the new file offset - dependent on the value of *Whence* and *Direction* as to the final effect on the new file offset.

Whence
Specifies how to interpret the *OffsetIn* parameter in setting the file pointer associated with the *Fildes* parameter. Values for the *Whence* parameter are as follows:

SEEK_SET - file offset set to *OffsetIn*.
SEEK_CUR - file offset set to current offset plus *OffsetIn*.
SEEK_END - file offset set to current end of file plus *OffsetIn*.

Direction
Specifies whether *OffsetIn* should be used to move forward or backward in the file.

HPSS_SET_OFFSET_FORWARD - consider *OffsetIn* as being a nonnegative number.
HPSS_SET_OFFSET_BACKWARD - consider *OffsetIn* as being a negative number.

OffsetOut
Points to an area to contain the file offset as a result of processing this request.

Return values

Upon successful completion, **hpss_SetFileOffset** returns zero. Otherwise, **hpss_SetFileOffset** returns a negative value; the absolute value of which indicates the specific error.

Error conditions

EBADF
The specified file descriptor does not refer to an open file.

EBUSY
The file is currently in use by another client thread.

EFAULT	*OffsetOut* is a NULL pointer.
EINVAL	The *Whence* or *Direction* parameter is invalid or the resulting offset would be invalid (a negative value or beyond the largest supported file size).
ENOSPC	Resources could not be allocated to satisfy the request.

See also

hpss_Read, **hpss_Write**, **hpss_Lseek**.

Notes

None.

2.1.70 hpss_SetLoginContext

Purpose

Establish a security context for an application.

Synopsis

```
#include <hpsscomm.h>

signed32 hpss_SetLoginContext(
        char        *PrincipalName,     /* IN */
        char        *KeytabName);       /* IN */
```

Description

This routine establishes a security context for an application. The routine gets the application's key from the named key table. If either of the arguments is NULL, default values will be obtained from environment variables `HPSS_PRINCIPAL` and `HPSS_KTAB_PATH`, respectively. This routine may be called as part of the application startup procedure. If this routine or a similar routine is not called, the application will run in the security context defined by `dce_login`.

Parameters

PrincipalName	Specifies the server's principal name.
KeytabName	Specifies the file name for the key table.

Return values

Upon successful completion, a value of zero is returned. Otherwise, one of the error conditions below is returned.

Error conditions

HPSS_E_NOERROR	Successful completion
HPSS_EINVAL	Invalid (NULL) parameters

DCE error codes

See also

hpss_PurgeLoginContext

Notes

In most cases, application programs should explicitly specify the principal name and keytab file name.

This routine may only be called once. On subsequent calls, it returns error HPSS_EINVAL.

2.1.71 hpss_SiteIdToName

Purpose

Convert a given HPSS site id to its corresponding HPSS site name.

Synopsis

```
#include "hpss_api.h"

int

hpss_SiteIdToName(

        uuid_t      *SiteId,        /* IN */
        char        *SiteName)      /* OUT */
```

Description

The hpss_SiteIdToName routine converts the HPSS site id given in *SiteId* to its corresponding string representation, returned in *SiteName*. When an error is encountered, hpss_SiteIdToName will return immediately.

Parameters

SiteId	If the site id pointer is null, or points to a zeroed parameter, the current local id is used.
SiteName	An HPSS site's text name. If the site name is null, return the current local site name. Site name should be a string of at least HPSS_MAX_DESC_NAME (32) bytes.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a negative value is returned; the absolute value of that returned is equal to an errno value defined below.

Error Conditions

ENOENT	The value passed in *SiteId* does not correspond to a known HPSS site.

ECONN	There was a communication problem during the translation.

See also

hpss_SiteNameToId

Notes

None.

2.1.72 hpss_SiteNameToId

Purpose

Convert a given HPSS site name to its corresponding HPSS site id.

Synopsis

```
#include "hpss_api.h"

int

hpss_SiteNameToId(

        char        *SiteName,      /* IN */
        uuid_t      *SiteId)        /* OUT */
```

Description

The hpss_SiteNameToId routine converts the HPSS site name given in *SiteName* to its corresponding HPSS site id, returned in *SiteId*. When an error is encountered, hpss_SiteIdToName will return immediately.

Parameters

SiteName	An HPSS site's text name. If the site name pointer is null, or points to an empty string, the current local site name is used.
SiteId	A pointer to an HPSS site's UUID. If the site id is null, return the current local site id.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a negative value is returned; the absolute value of that returned is equal to an errno value defined below.

Error Conditions

ENOENT	The value passed in *SiteName* does not correspond to a known HPSS site.
ECONN	There was a communication problem during the translation.

See also

hpss_SiteIdToName

Notes

None.

2.1.73 hpss_Stage

Purpose

Stage a piece of a file to a specified level in the storage hierarchy.

Synopsis

```
#include "hpss_api.h"
int

hpss_Stage(
        int             Fildes,         /* IN */
        u_signed64      Offset,         /* IN */
        u_signed64      Length,         /* IN */
        unsigned32      StorageLevel,   /* IN */
        unsigned32      Flags);         /* IN */
```

Description

The **hpss_Stage** routine stages part of an open file, specified by *Fildes, Offset* and *Length* to a level in the storage hierarchy, specified by *StorageLevel*. The *Flags* argument is used to control behavior of the request.

Parameters

Fildes	Specifies the file descriptor, identifying the file to be staged.
Offset	Specifies the offset of the start of the data to be staged.
Length	Specifies the length of the data to be staged.
StorageLevel	Identifies the level in the storage hierarchy to which the data is to be staged. Currently, the only supported value is 0.
Flags	Controls the behavior of the stage request. Valid values include: BFS_STAGE_ALL - stage entire file. BFS_ASYNC_CALL - return after initiating stage.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EBADF	The supplied file descriptor does not correspond to an open file.
EINVAL	The *Flags, Offset* or *Length* argument is invalid.

EBUSY	The specified file descriptor is currently in use.
EPERM	The client does not have the appropriate privileges to perform the operation.

See also

hpss_Migrate, **hpss_Purge, hpss_StageCallBack**.

Notes

None.

2.1.74 hpss_StageCallBack

Purpose

Initiate staging a piece of a file in the background.

Synopsis

```
#include "hpss_api.h"

int
hpss_StageCallBack(
        char                    *Path,          /* IN */
        u_signed64              Offset,         /* IN */
        u_signed64              Length,         /* IN */
        unsigned32              StorageLevel,   /* IN */
        bfs_callback_addr_t     *CallBackPtr,   /* IN */
        unsigned32              Flags,          /* IN */
        signed32                *ReqID,         /* OUT */
        hpssoid_t               *BitfileID);    /* OUT */
```

Description

The **hpss_StageCallBack** routine initiates a background stage of part of a file, specified by *Fildes, Offset* and *Length* to a level in the storage hierarchy, specified by *StorageLevel*. The *Flags* argument is used to control behavior of the request.

Parameters

Path	Specifies the pathname of the file to be staged.
Offset	Specifies the offset of the start of the data to be staged.
Length	Specifies the amount of the data to be staged.
StorageLevel	Identifies the level in the storage hierarchy to which the data is to be staged.
CallBackPtr	Callback information. If NULL, no callback actions are taken.

Flags	Controls the behavior of the stage request. Valid values include: BFS_STAGE_ALL - stage entire file.
ReqID	Assigned request identification number.
BitfileID	Bitfile ID.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EACCESS	Search permission is denied for a component of the path prefix.
EFAULT	The *Path* parameter is a NULL pointer.
EINVAL	The value of the *Offset* parameter is beyond the end of the file or the *StorageLevel* parameter is invalid for the storage hierarchy.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the pathname exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.
ENOTDIR	A component of the *Path* prefix is not a directory.

See also

hpss_Migrate, **hpss_Purge, hpss_Stage**.

Notes

None.

2.1.75 hpss_Stat

Purpose

Get file status (POSIX).

Synopsis

```
#include "hpss_api.h"

int
hpss_Stat(
        char            *Path,          /* IN */
        struct stat     *Buf);          /* OUT */
```

Description

The **hpss_Stat** function obtains information about the file named by *Path* and returns it in the structure pointed to by *Buf*. Refer to POSIX.1 for more detailed information.

Parameters

Path	Points to the path name of the file being queried.
Buf	Points to a **stat** structure that will contain the status information for the file.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value set by POSIX.1 **stat**.

Error conditions

EACCES	Search permission is denied for a component of the path prefix.
EFAULT	The *Path* or *Buf* parameter is a NULL pointer.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.
ENOTDIR	A component of the *Path* prefix is not a directory.

See also

hpss_Chown, **hpss_Chmod**, **hpss_Utime**, **hpss_FileGetAttributes**, **hpss_FileSetAttributes**, **hpss_Lstat**, **hpss_Fstat**, **hpss_GetListAttrs**, **hpss_ReadAttrs**.

Notes

Note that if the named file is a symbolic link, information is returned for the file to which the contents of the link point. See **hpss_Lstat** to obtain information about the symbolic link itself .

2.1.76 hpss_Statfs

Purpose

Returns file system information for a Class of Service.

Synopsis

#include "hpss_api.h"

```
int
hpss_Statfs(
        unsigned long          COSId,              /* IN */
        struct statfs          *StatfsBuffer);     /* OUT */
```

Description

The **hpss_Statfs** routine returns file system information as defined in the *statfs* structure.

Parameters

COSId	Specifies the identifier for the Class of Service that is being queried.
StatfsBuffer	Points to area to contain the file system information.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EFAULT	The *StatfsBuffer* parameter is a NULL pointer.
EINVAL	The specified Class of Service does not exist.

See also

None.

Notes

None.

2.1.77 hpss_Statvfs

Purpose

Returns file system information for a Class of Service.

Synopsis

```
#include "hpss_api.h"

int

hpss_Statvfs(

        unsigned32             CosId,              /* IN */
        struct statvfs         *StatvfsBuffer)     /* OUT */
```

Description

The hpss_Statvfs routine returns file system information as defined in the *statvfs* structure for the

given Class of Service as a total from all Bitfile Servers which manage files in the Class of Service.

Parameters

CosId	The identifier for the Class of Service that is being queried.
StatvfsBuffer	Pointer to area to contain the file system information.

Return Values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned; the absolute value of that returned is equal to an errno value defined below.

Error Conditions

EFAULT	The *StatvfsBuffer* parameter is a NULL pointer.
EINVAL	The specified Class of Service does not exist.

See also

None.

Notes

The block size is that of the top storage class in the hierarchy.

The f_name field of the statvfs structure will return the CDS service name of the root Name Server. The f_pack field of the statvfs structure will return the CDS name of the local Bitfile Server that contains the most free space for the given Class of Service.

2.1.78 hpss_Symlink

Purpose

Create a symbolic link.

Synopsis

```
#include "hpss_api.h"

int
hpss_Symlink(
        char            *Contents,      /* IN */
        char            *Path);         /* IN */
```

Description

The **hpss_Symlink** routine creates a symbolic link pointing to the pathname specified in *Contents* with the link's name identified by *Path*.

Parameters

Contents	Specifies the path name to which the symbolic link will point.

Path	Specifies the name of the symbolic link to be created.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EACCES	Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the specified path name.
EFAULT	The *Path* or *Contents* parameter is a NULL pointer.
EEXIST	The specified file already exists.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	No entry exists for a component of the path name.
ENOTDIR	A component of the *Path* prefix is not a directory.

See also

hpss_Readlink.

Notes

None.

2.1.79 hpss_ThreadCleanUp

Purpose

Cleans up a thread's Client API state.

Synopsis

```
#include "hpss_api.h"

int
hpss_ThreadCleanUp(
        pthread_t          ThreadID);          /* IN */
```

Description

The **hpss_ThreadCleanUp** routine frees resources used by a thread's Client API context.

Parameters

ThreadID	Specifies the thread identifier for the thread whose resources are to

be freed.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

ENOENT	State for the specified thread could not be found.

See also

None.

Notes

The **hpss_ThreadCleanUp** routine should be called once for each thread which terminates and has previously called the Client API.

2.1.80 hpss_Truncate

Purpose

Set the length of a file.

Synopsis

```
#include "hpss_api.h"

int
hpss_Truncate(
        char            *Path,          /* IN */
        u_signed64      Length);        /* IN */
```

Description

The **hpss_Truncate** routine sets the length of a file, specified by the *Path* argument. If the new file length is less than the current length, the space allocated beyond the new length will be freed. If the new length is greater than the current length, a hole is created in the file.

Parameters

Path	Specifies the path name of the file to be truncated.
Length	Specifies the new length of the file.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EACCES	Search permission is denied on a component of the path prefix, or

	write permission is denied on the file.
EFAULT	The *Path* parameter is a NULL pointer.
EINVAL	*Path* specifies a directory.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The specified path name does not exist.
ENOTDIR	A component of the *Path* prefix is not a directory.

See also

hpss_Ftruncate, **hpss_Fclear**, **hpss_FileSetAttributes**.

Notes

None.

2.1.81 hpss_Umask

Purpose

Set the file creation mask.

Synopsis

```
#include "hpss_api.h"

mode_t

hpss_Umask(
        mode_t          Cmask);          /* IN */
```

Description

The **hpss_Umask** function sets the file mode creation mask of the thread and returns the previous value of the mask. Refer to POSIX.1 **umask** for further details.

Parameters

Cmask	Specifies the file mode creation mask to be used by subsequent **hpss_Open**, **hpss_Create** and **hpss_Mkdir** calls.

Return values

hpss_Umask returns the previous file mode creation mask for the thread.

Error conditions

The **hpss_Umask** function is always successful and no return values are reserved to indicate an error.

See also

hpss_Open, **hpss_Create**, **hpss_Mkdir**.

Notes

None.

2.1.82 hpss_Unlink

Purpose

Remove an entry from an HPSS directory.

Synopsis

```
#include "hpss_api.h"

int
hpss_Unlink(
        char                    *Path);             /* IN */
```

Description

The **hpss_Unlink** function removes the entry named by the *Path*, and decrements the link count of the file. If the link count becomes zero, the file will be deleted when it is no longer open by any client.

Parameters

Path	Names the directory entry to be removed.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value set by POSIX.1 **unlink**.

Error conditions

EACCES	Search permission is denied on a component of the path prefix, or write permission is denied on the directory containing the link to be removed.
EFAULT	The *Path* parameter is a NULL pointer.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.

ENOTDIR	A component of the *Path* prefix is not a directory.
EPERM	The file named by *Path* is a directory.

See also

hpss_Close, **hpss_Link**.

Notes

Note that using **hpss_Unlink** to remove directory names is not supported.

Also note that if the *Path* refers to a symbolic link, the link itself shall be removed.

2.1.83 hpss_UpdateACL

Purpose

Update entries in the Access Control List of a file.

Synopsis

```
#include "hpss_api.h"

int
hpss_UpdateACL(
        char                    *Path,          /* IN */
        unsigned32              Options,        /* IN */
        ns_ACLConfArray_t       *ACL);          /* IN */
```

Description

The **hpss_UpdateACL** function updates entries, specified by *ACL*, in the access control list for the file named by *Path*.

Parameters

Path	Names the file for which the *ACL* is being updated.
Options	A bit vector containing bits which control the behavior of **hpss_UpdateACL**. It is possible to specify which ACL is to be updated, and to specify the behavior of **hpss_UpdateACL** while calculating the MASK_OBJ.

Options can be used to mimic the behavior of the following **acl_edit** options: -n, -c, and -p. This mimicking is done using the following mutually-exclusive constants:

HPSS_ACL_DONT_CALC_MASK

Specifies that a new MASK_OBJ should not be calculated. This option is useful when the ACL operations require the calculation of a new MASK_OBJ, but doing so would result in an error. This

option allows the operations to be carried out, but a new MASK_OBJ is not calculated.

HPSS_ACL_CALC_MASK_IGNORE_ERRORS

Creates or modifies the object's MASK_OBJ entry with permissions equal to the union of all entries other than type USER_OBJ, OTHER_OBJ, and UNAUTHENTICATED. This creation or modification is done after all other modifications to the ACL are performed. The new MASK_OBJ is set even if it grants permissions previously masked out. It is recommended that this option be used only if not specifying it results in an error. This option is useful only for objects that support the MASK_OBJ entry type and are required to recalculate a new MASK_OBJ after they are modified.

HPSS_ACL_PURGE_MASKED_PERMS

Purges all masked permissions (before any other modifications are made). This option is useful only for ACLs that contain an entry of type MASK_OBJ. Use it to prevent unintentionally granting permissions to an existing entry when a new MASK_OBJ is calculated as a result of adding or modifying an ACL entry.

Options can also be used to specify which ACL is to be updated. The following mutually-exclusive constants can be used to make this selection:

HPSS_ACL_OPTION_OBJ

HPSS_ACL_OPTION_IC

HPSS_ACL_OPTION_IO

If an update operation creates a MASK_OBJ that unintentionally adds permissions to an existing ACL entry, the modification causing the MASK_OBJ recalculation will abort with an error unless the HPSS_ACL_CALC_MASK_IGNORE_ERRORS or HPSS_ACL_DONT_CALC_MASK options are specified.

ACL	Points to the ACL entries to be updated.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EACCES	Search permission is denied on a component of the path prefix.
EFAULT	The *Path* or *ACL* parameter is a NULL pointer.
EINVAL	There are two sets of mutually exclusive flags available through the *Options* parameter. Some invalid combination of flags was provided.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.

ENOTDIR	A component of the *Path* prefix is not a directory.
EPERM	The client does not have the appropriate privileges to perform the operation.

See also

hpss_DeleteACL, **hpss_GetACL**, **hpss_SetACL**.

Notes

This function is supported in the standard Client API library, but not in the non-DCE Client API library.

2.1.84 hpss_Utime

Purpose

Set access and modification times of an HPSS file.

Synopsis

```
#include <utime.h>
#include "hpss_api.h"

int
hpss_Utime(
        char                    *Path,          /* IN */
        const struct utimbuf    *Times);        /* IN */
```

Description

The **hpss_Utime** function sets the access and modification times of the file named by *Path* to the values specified in the structure pointed to by *Times*. Refer to POSIX.1 for more detailed information.

Parameters

Path	Names the file for which times are being changed.
Times	Points to a structure containing the new time values.

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value set by POSIX.1 **utime**.

Error conditions

EACCES	Search permission is denied on a component of the path prefix.
EFAULT	The *Path* parameter is a NULL pointer.
ENAMETOOLONG	The length of the *Path* argument exceeds the system-imposed

	limit, or a component of the path name exceeds the system-imposed limit.
ENOENT	The named file does not exist, or the *Path* argument points to an empty string.
ENOTDIR	A component of the *Path* prefix is not a directory.
EPERM	The client does not have the appropriate privileges to perform the operation.

See also

hpss_Stat, **hpss_FileGetAttributes**, **hpss_FileSetAttributes**.

Notes

None.

2.1.85 hpss_Write

Purpose

Write data from a client buffer to a contiguous section of an HPSS file, beginning at the current file offset.

Synopsis

```
#include "hpss_api.h"

ssize_t

hpss_Write(
        int             Fildes,         /* IN */
        const void      *Buf,           /* IN */
        size_t          Nbyte);         /* IN */
```

Description

The **hpss_Write** function attempts to write *Nbyte* bytes from the client buffer pointed to by *Buf* to the file associated with the open file handle, *Fildes*.

Parameters

Fildes	Specifies the open file handle associated with the file to which data is to be written.
Buf	Points to a buffer where the data is to be found.
Nbyte	Specifies the number of bytes to be written.

Return values

Upon successful completion, **hpss_Write** returns a nonnegative value that is the number of bytes written. Otherwise, **hpss_Write** returns a negative value; the absolute value of which is equal to an errno value set by POSIX.1 **write**.

Error conditions

EBADF	The specified file descriptor does not correspond to a file opened for writing.
EBUSY	The file is currently in use by another client thread.
EFAULT	The *Buf* parameter is out of range.
EFBIG	The write operation would cause the file to exceed the system-imposed maximum file length.
EIO	An input/output or HPSS internal error occurred.
ENOSPC	There is no free space remaining to satisfy the write request.

See also

hpss_Open, **hpss_OpenBitfile**, **hpss_Read**, **hpss_Lseek**, **hpss_SetFileOffset**, **hpss_ReadList**, **hpss_WriteList**.

Notes

None.

2.1.86 hpss_WriteList

Purpose

Write data to an HPSS file, specifying lists for data source and sink.

Synopsis

```
#include "hpss_api.h"

int

hpss_WriteList(
        IOD_t           *IODPtr,        /* IN */
        unsigned32      Flags,          /* IN */
        IOR_t           *IORPtr);       /* OUT */
```

Description

The **hpss_WriteList** function writes data to an HPSS file specified by the sink descriptor list in the IOD pointed to by *IODPtr*, moving the data from the sources specified by the source descriptor list in the IOD. Results of the request are returned in the structure pointed to by *IORPtr*. Refer to Chapter 3 for a description of the IOD.

Parameters

IODPtr->Function	Specifies the IOD function type. Set to **IOD_WRITE**.
IODPtr->SrcDescLength	Specifies the number of entries in the source descriptor list.
IODPtr->SinkDescLength	Specifies the number of entries in the sink descriptor list.
IODPtr->SrcDescList	Specifies a list of descriptors specifying the sources for the data.
IODPtr->SinkDescList	Specifies a list of descriptors specifying the parts of the file to be written.
Flags	Specifies a bitmap containing flags to modify the write. Currently only a value of zero (0) is valid.
IORPtr->RequestID	Specifies the request identifier assigned to this request by the Client API.
IORPtr->Flags	Specifies status flags. Valid values are: IOR_COMPLETE - indicates the request has completed. IOR_ERROR - indicates an error was encountered.
IORPtr->SrcReplyLength	Specifies the number of source reply descriptors.
IORPtr->SinkReplyLength	Specifies the number of sink reply descriptors.
IORPtr->SrcReplyList	Specifies a list of descriptors specifying the data source results.
IORPtr->SinkReplyList	Specifies a list of descriptors specifying the data sink results.

Return values

Upon successful completion, **hpss_WriteList** returns zero. Otherwise, **hpss_WriteList** returns a negative value; the absolute value of which indicates the specific error.

Error conditions

EBADF	A specified file descriptor in the sink descriptor list does not correspond to a file opened for writing.
EBUSY	The file is currently in use by another client thread.
EFAULT	A memory buffer address in the source descriptor list is out of range.
EFBIG	An attempt was made to write a file that would exceed the HPSS-defined maximum file size.
EINVAL	A sink descriptor did not specify a client file address, a source descriptor specified an invalid address type, or *Flags* was invalid.
EIO	An input/output or HPSS internal error occurred.
ENOSPC	There is no free space remaining to satisfy the write request.

See also

hpss_Open, **hpss_OpenBitfile**, **hpss_Read**, **hpss_Write**, **hpss_ReadList**, **free_ior_mem**.

Notes

Memory allocated to the returned I/O Reply can be freed by calling free_ior_mem(), and supplying the same IORPtr that was passed to the hpss_ReadList() call

2.1.87 hpss_XLoadThreadState

Purpose

Updates the user credentials and file/directory creation mask for the current thread's Client API state, based on the specified user ID and fully-qualified DCE client name. The fully-qualified client name allows for correct DCE cross-cell authentication and authorization to be performed.

Synopsis

```
#include "hpss_api.h"

int
hpss_XLoadThreadState(
        uid_t           UserID,          /* IN */
        mode_t          Umask,           /* IN */
        char            *ClientFullName); /* IN */
```

Description

The **hpss_XLoadThreadState** routine updates the user credentials and file/directory creation mask found in the current thread's Client API state, using the specified user ID and fully-qualified DCE client name.

Parameters

UserID	Specifies the user ID for the user whose credentials are to be loaded.
Umask	Specifies the new file/directory creation mask.
ClientFullName	Specifies the fully-qualified client name in the following format /.../{dce cell name}/username (e.g., /.../dce.sandia.gov/jtjoker)

Return values

Upon successful completion, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

ENOENT	Credentials for the specified user could not be obtained.

See also

hpss_LoadThreadState, hpss_LoadDefaultThreadState

Notes

This routine is primarily used for DCE cross-cell authentication.

Normally, the structure pointed to by the *IORPtr* parameter should be zeroed out, otherwise pointers in that structure will be used by the RPC mechanism as if they point to previously allocated memory.

After the client has completed using the reply information returned in the IOR, the pointers returned as part of the IOR should be freed using **rpc_ss_client_free**(). The following pointers must be freed (if non-NULL pointers are returned):

IORPtr->SrcReplyList (each element in the list must be freed)

IORPtr->SinkReplyList (each element in the list must be freed)

2.1.88 free_ior_mem

Purpose

Free memory allocated to a returned I/O Reply structure.

Synopsis

```
#include "traniod.h

void
free_ior_mem(
        IOR_t                   *IORPtr);           /* IN */
```

Description

The **free_ior_mem** function releases the memory that was allocated to the I/O Reply pointed to by IORPtr. The memory would have been previously allocated during a call to **hpss_ReadList** or **hpss_WriteList**.

Parameters

IORPtr	Points to the I/O Reply that contains pointers to memory areas which are to be released.

Return values

None.

Error conditions

None.

See also

hpss_ReadList, **hpss_WriteList**.

Notes

free_ior_mem should only be used to free memory that was allocated to an I/O Reply returned by

a call to **hpss_ReadList** or **hpss_WriteList**.

2.2 Non-DCE Client API Specific Interfaces

This section describes only those API's that are available through the Non-DCE Client API which are not available through the standard Client API. For notes on how the NDAPI differs from the standard Client API, see section 1.3

Note that there are two errors (in additon to the ones listed in section 2.1) that may be returned from a Non-Client API call which are not actually errors generated by performing the call, but are caused by a failure of the library to successfully communicate with the Non-DCE Client Gateway. These values may be returned from any routine and include:

EPIPE	This indicates a communications problem with the Non-DCE Client Gateway, between the time that the command was issued and the time the reply was received. In cases where this error is returned from a API that modifies the state of an HPSS object, the failure or success of the operation can not be assumed and the state of the object should be queried before continuing.An HPSS server is not ready or received a communication error, and the request could not be retried.
ENOCONNECT	This indicates a communication problem either between between the Non-DCE Client Library and the Non-DCE Client Gateway, or the standard Client Library and one of the core HPSS servers (Name Server, Bitfile Server or Location Server).

2.2.1 hpss_PVRetrievals

Purpose

Get recent requests on a physical volume.

Synopsis

```
#include "hpss_api.h

int
hpss_PVRetrievals(
        char            *PVName,                /* IN */
        unsigned32      *MountCntSinceService,  /* OUT */
        unsigned32      *MountCntSinceMaint,    /* OUT */
        unsigned32      *NumReads,              /* OUT */
        unsigned32      *NumWrites);            /* OUT */
```

Description

The **hpss_PVRerrievals** function is used to get physical volume access statistics for a selected volume. These statistics include number of mounts since last service, number of mounts since last maintenance, number of reads, and number of writes.

Parameters

PVName	The physical volume name.

MountCntSinceService	The number of mounts since last service.
MountCntSinceMaint	The number of mounts since last maintainence.
NumReads	The number of reads.
NumWrites	The number of writes.

Return values

None.

Error conditions

None.

See also

None.

Notes

None.

2.3 Data Definitions

This section describes key internal data definitions and all externally used data definitions which are provided by this subsystem. A data definition may be represented by constructs such as data structures and constants. For each data definition, a description, format (including parameter descriptions), and clients which access the data definition are provided. Note: Descriptions of the IOD and IOR structures may be found in Chapter 3.

2.3.1 File Creation Hint Structure - hpss_cos_hints_t

Description

The file creation hint structure contains information that allows clients to specify preferences or knowledge of file structure or access patterns that may affect operations of HPSS.

Format

The COS hints has the following format:

```
typedef struct hpss_cos_hints {
   unsigned32           COSId;
   char                 COSName [HPSS_MAX_OBJECT_NAME];
   u_signed64           OptimumAccessSize;
   u_signed64           MinFileSize;
   u_signed64           MaxFileSize;
   unsigned32           AccessFrequency;
   unsigned32           TransferRate;
   unsigned32           AvgLatency;
   unsigned32           WriteOps;
   unsigned32           ReadOps;
   unsigned32           StageCode;
   unsigned32           StripeWidth;
   u_signed64           StripeLength;
} hpss_cos_hints_t;
```

COSId

The class of service type. It indicates the classes of service requested for the bitfile.

COSName

Specifies the name of the class of service for this bitfile.

OptimumAccessSize

Specifies the block size in bytes for this class of service that yields the maximum data transfer rate.

MinFileSize

Specifies the minimum size in bytes of a bitfile in this class of service.

MaxFileSize

Specifies the maximum size in bytes to which the bitfile can grow and remain in this class of service.

AccessFrequency

Specifies the expected rate of access for the bitfile.

FREQ_HOURLY

FREQ_DAILY

FREQ_WEEKLY

FREQ_MONTHLY

FREQ_ARCHIVE

TransferRate

Specifies the approximate file transfer rate in kilobytes per second.

AvgLatency

Specifies the time in seconds from when a request is received by a storage server until data actually begins to transmit. This is typically non-zero for tape media.

WriteOps

Specifies the valid write operations for the bitfile:

HPSS_OP_WRITE	Allow write operations.
HPSS_OP_APPEND	Allow append operations.

ReadOps

Specifies the valid read operations for the bitfile:

HPSS_OP_READ	Allow read operations.

StageCode

Specifies the staging behavior desired:

COS_STAGE_NO_STAGE	File is not to be staged on open. The data will be read from the current level in the hierarchy, or data may be explicitly staged by the client.
COS_STAGE_ON_OPEN	Entire file is to be staged to the top level in the hierarchy before open returns.
COS_STAGE_ON_OPEN_ASYNC	Entire file is to be staged to the top level in the hierarchy without blocking in open. Reads / writes are blocked only until the portion of the file being accessed is staged.
COS_STAGE_ON_OPEN_BACKGROUND	File is to be staged in a background task.

StripeWidth

Specifies the stripe width of the class of service.

StripeLength

Specifies the stripe length of the class of service.

2.3.2 Class of Service Priorities - hpss_cos_priorities_t

Description

The class of service priorities structure assists a client in selecting a COS for a bitfile.

Structure use - dynamic memory tables.

Format

The COS priorities has the following format:

```
typedef struct hpss_cos_priorities {
   unsigned32          COSIdPriority;
   unsigned32          COSNamePriority;
   unsigned32          OptimumAccessSizePriority;
   unsigned32          MinFileSizePriority;
   unsigned32          MaxFileSizePriority;
   unsigned32          AccessFrequencyPriority;
   unsigned32          TransferRatePriority;
   unsigned32          AvgLatencyPriority;
   unsigned32          WriteOpsPriority;
   unsigned32          ReadOpsPriority;
   unsigned32          StageCodePriority;
   unsigned32          StripeWidthPriority;
   unsigned32          StripeLengthPriority;
} hpss_cos_priorities_t;
```

COSIdPriority

Specifies the class of service ID priority for the class of service the bitfile should be in.

COSNamePriority

Specifies the class of service name priority for this bitfile.

OptimumAccessSizePriority

Specifies the priority for the block size for this class of service that yields the maximum data transfer rate.

MinFileSizePriority

Specifies the priority for the minimum size in bytes of a bitfile in this class of service.

MaxFileSizePriority

Specifies the priority for the maximum size in bytes to which the bitfile can grow and remain in this class of service.

AccessFrequencyPriority

Specifies the priority for the expected rate of access for the bitfile.

TransferRatePriority

Specifies the priority for the class of service file transfer rate.

AvgLatencyPriority

Specifies the class of service priority for the average latency time from request time until data begins to transfer.

WriteOpsPriority

Specifies the priority for the valid write operations for the bitfile.

ReadOpsPriority

Specifies the priority for the valid read operations for the bitfile.

StageCodePriority

Specifies the priority for the desired stage code.

Following are the possible priority values:

NO_PRIORITY

LOW_PRIORITY

DESIRED_PRIORITY

HIGHLY_DESIRED_PRIORITY

REQUIRED_PRIORITY

StripeWidthPriority

Specifies the stripe width priority of the storage class.

StripeLengthPriority

Specifies the stripe length priority of the storage class.

2.3.3 Class of Service Metadata Structure - hpss_cos_md_t

Description

The Class of Service metadata structure contains information about the configuration of a Class of Service.

Format

The Class of Service metadata structure has the following format:

```
typedef struct {
   unsigned32           COSId;
   unsigned32           HierId;
   char                 COSName [HPSS_MAX_OBJECT_NAME];
   unsigned32           OptimumAccessSize;
   unsigned32           Flags
   u_signed64           MinFileSize;
   u_signed64           MaxFileSize;
   unsigned32           AccessFrequency;
   unsigned32           TransferRate;
   unsigned32           AvgLatency;
   unsigned32           WriteOps;
   unsigned32           ReadOps;
   unsigned32           StageCode;
} hpss_cos_md_t;
```

COSId

The class of service type. It indicates which of several classes of service the bitfile is in.

HierId

The storage hierarchy associated with this Class of Service.

COSName

Specifies the name of the class of service for this bitfile.

OptimumAccessSize

Specifies the block size in bytes for this class of service that yields the maximum data transfer rate.

Flags

Optionally specifies one of the following options:

COS_ENFORCE_MAX_FILE_SIZE	If ON, bitfiles cannot be created in this COS with a size greater than MaxFileSize. Attempts to do so will result in the request being rejected with an error.
COS_FORCE_SELECTION	If ON, a client must explicitly select this COS in order to have a file assigned to it. If the client merely supplies general COS hints for a bitfile, this COS will not be selected.

MinFileSize

Specifies the minimum size in bytes of a bitfile in this class of service.

MaxFileSize

Specifies the maximum size in bytes to which the bitfile can grow and remain in this class of service.

AccessFrequency

Specifies the expected rate of access for the bitfile.

FREQ_HOURLY

FREQ_DAILY

FREQ_WEEKLY

FREQ_MONTHLY

FREQ_ARCHIVE

TransferRate

Specifies the approximate file transfer rate in kilobytes per second.

AvgLatency

Specifies the time in seconds from when a request is received by a storage server until data actually begins to transmit. This is typically non-zero for tape media.

WriteOps

Specifies the valid write operations for the bitfile:

HPSS_OP_WRITE	Allow write operations.
HPSS_OP_APPEND	Allow append operations.

ReadOps

Specifies the valid read operations for the bitfile:

HPSS_OP_READ	Allow read operations.

StageCode

Specifies the staging behavior desired

COS_STAGE_NO_STAGE	File is not to be staged on open. The data will be read from the current level in the hierarchy, or data may be explicitly staged by the client.
COS_STAGE_ON_OPEN	Entire file is to be staged to the top level in the hierarchy before open returns.
COS_STAGE_ON_OPEN_ASYNC	Entire file is to be staged to the top level in the hierarchy without blocking in open. Reads / writes are blocked only until the portion of the file being accessed is staged.
COS_STAGE_ON_OPEN_BACKGROUND	File is to be staged as a background task.

2.3.4 File Attribute Structure - hpss_fileattr_t

Description

The file attribute structure contains file attributes that are managed by both the Name Server and the Bitfile Server.

Format

The file attribute structure has the following format:

```
typedef struct hpss_filattr {
   ns_ObjHandle_t        NSObjectHandle;
   ns_Attrs_t            NSAttr;
   bf_attrib_t           BFSAttr;
} hpss_fileattr_t;
```

NSObjectHandle

Specifies the handle that refers to the open file or directory. Refer to section 2.3.8 for more detailed information.

NSAttr

Specifies the file attributes managed by the Name Server. Reference sect for more detailed information.

BFSAttr

Specifies the file attributes managed by the Bitfile Server. Reference the Bitfile Server design document for more detailed information.

2.3.5 Extended File Attribute Structure - hpss_xfileattr_t

Description

The file attribute structure contains file attributes that are managed by both the Name Server and the Bitfile Server. This structure allows the user to retrieve the extended attributes for the Bitfile Server.

Format

The file attribute structure has the following format:

```
typedef struct hpss_xfilattr {
   ns_ObjHandle_t       NSObjectHandle;
   ns_Attrs_t           NSAttr;
   bf_xattrib_t         BFSAttr;
} hpss_xfileattr_t;
```

NSObjectHandle

Specifies the handle that refers to the open file or directory. Refer to section 2.3.8 for more detailed information.

NSAttr

Specifies the file attributes managed by the Name Server. Reference section 2.3.6 for more detailed information.

BFSAttr

Specifies the extended file attributes managed by the Bitfile Server. Reference the Bitfile Server design document for more detailed information.

2.3.6 Name Server Attribute Structure - ns_Attrs_t

Description

The Name Server attribute structure contains fields for the various attributes (metadata) that the Name Server maintains for an object.

Format

The Name Server attributes structure has the following format:

```
typedef struct {
   unsigned32       Account;
   unsigned32       ACLOptions;
   hpssiod_t        BitFileId;
   unsigned32       ClassOfService;
   unsigned char    Comment[HPSS_MAX_COMMENT_LENGTH];
   unsigned32       CompositePerms;
   byte             DMHandle[MAX_DMEPI_HANDLE_SIZE];
   unsigned32       DMHandleLength;
   unsigned32       EntryCount,
   unsigned32       FamilyId;
   ns_ObjHandle_t   FilesetHandle;
   u_signed64       FilesetId;
   unsigned32       FilesetRootRSN;
   unsigned32       FilesetStateFlags;
   unsigned32       FilesetType;
   u_signed64       FileSize;
   unsigned32       Flags;
   uuid_t           GatewayUUID;
   unsigned32       GID;
   unsigned32       GroupPerms;
   unsigned32       LinkCount;
   unsigned32       Location;
   unsigned32       MACSecLabel;
   unsigned32       OtherPerms;
   unsigned32       SetGIDOnExe;
   unsigned32       SetStickyBit;
   unsigned32       SetUIDOnExe;
   timestamp_t      TimeLastRead;
   timestamp_t      TimeLastWritten;
   timestamp_t      TimeOfCreation;
   timestamp_t      TimeOfMetadataUpdate;
   unsigned32       Type;
   unsigned32       UID;
   unsigned32       UserPerms;
} ns_Attrs_t;
```

Account

Specifies opaque accounting information.

ACLOptions

Specifies Access Control List options used when setting the group permissions.

BitFileId

Specifies the bitfile identifier.

ClassOfService

Specifies the class of service of a file object. This field is not settable.

Comment

Specifies the uninterpreted client supplied ASCII text.

CompositePerms

Specifies the permission to an object after all ACLs have been examined and applied.

DMHandle

Specifies a handle that points back to a DMAP managed object. This field is opaque data to the Name Server.

.DMHandleLength

Specifies the byte length of DMHandle.

EntryCount

Specifies a read-only field which contains the number of entries contained in a directory. If the object is not a directory, the value is not defined.

FamilyId

Identifies the fileset family identifier.

FilesetHandle

Specifies a Name Server object handle used to point to the root node of a fileset.

FilesetId

Specifies the fileset identifier that uniquely identifes the fileset an object belongs to.

FilesetRootRSN

Specifies a read-only field which contains the Relative Sequence Number of the root node of this fileset.

FilesetStateFlags

Contains flag bits indicating the state of the fileset. The following constants define the possible states:

NS_FS_STATE_READ	Read is permitted.
NS_FS_STATE_WRITE	Write is permitted.
NS_FS_STATE_DESTROYED	The fileset has been destroyed. Neither reading nor writing will be permitted
NS_FS_STATE_READ_WRITE	A combination of READ and WRITE.
NS_FS_STATE_COMBINED	A combination of all bit settings above.

FilesetType

Specifies the type of the fileset the attributes are for. This is a read-only field. The following constants define the fileset types:

NS_FS_TYPE_HPSS_ONLY This fileset is an HPSS-only fileset.

NS_FS_TYPE_ARCHIVED This fileset is a backup copy of some other fileset.

NS_FS_TYPE_DFS_ONLY This fileset is native to some other file system such as DFS.

NS_FS_TYPE_MIRRORED This fileset is a mirrored copy of some other fileset such as a DFS fileset.

Filesize

Specifies the byte size of a file, directory, or symbolic link object. This field is not settable.

Flags

Specifies a bit vector which contains information that can be expressed in boolean form. The following constants define the bits in this field:

NS_ATTRS_FLAGS_EXTENDED_ACLS

Set to 1 if the object has extended ACL entries. Extended ACL entries are all entries other than user_obj, group_obj, and other_obj.

GID

Specifies the principal group identifier.

GroupPerms

Specifies the permissions granted to group members.

LinkCount

Specifies the number of hard links to a file object.

Location

On input, contains the DCE cell identifier of the GROUP_OBJ. On output, it may contain a DCE cell identifier; however, an output value of zero indicates the local cell.

MACSecLabel

Specifies the Mandatory Access Control Security Label.

OtherPerms

Specifies the permissions granted to ‘other’ clients.

SetGIDOnExe

For file objects:

0 = do not set GID to owner.

1 = set GID to owner.

SetStickyBit

For file objects:

0 = do not set the sticky bit.

1 = set the sticky bit.

SetUIDonExe

For file objects:

0 = do not set UID to owner.

1= set UID to owner.

TimeLastRead

Specifies the last time the object was accessed.

TimeLastWritten

Specifies the last time the object was updated.

TimeOfCreation

Specifies the time the object was created.

TimeOfMetadataUpdate

Specifies the last time the metadata was updated.

Type

Specifies the 'type' of the object:

NS_DIRECTORY_OBJECT	directory
NS_FILE _OBJECT	file
NS_JUNCTION_OBJECT	junction
NS_SYM_LINK_OBJECT	symbolic link
NS_HARD_LINK_OBJECT	hard link.

This field is not settable

UID

Specifies the User Identifier of the object's owner.

UserPerms

Specifies the permissions granted to the owner of the object.

2.3.7 Name Server Fileset Attributes Structure – ns_FilesetAttrs_t

Description

The Name Server fileset attribute structure contains fields for the various attributes (metadata) that the Name Server maintains for a fileset.

Format

```
typedef struct {
   u_signed64           RetisterBitMap;
   unsigned32           ClassOfService;
   unsigned32           FamilyId;
   ns_ObjHandle_t       FilesetHandle;
   u_signed64           FilesetId;
   unsigned char        FilesetName[NS_FS_MAX_FS_NAME_LENGTH];
   unsigned32           FilesetType;
   uuid_t               GatewayUUID;
   unsigned32           StateFlags;
   unsigned32           SubSystemId;
   byte                 UserData[NS_FS_MAX_USER_DATA];
   u_signed64           DirectoryCount;
   u_signed64           FileCount;
   u_signed64           HardLinkCount;
   u_signed64           JunctionCount;
   u_signed64           SymLinkCount;
} ns_FilesetAttrs_t;
```

RegisterBitMap

A bit vector where each bit corresponds to a field in the record.

ClassOfService

The COS service configured for this fileset.

FamilyId

The fileset family identifier. This id is opaque to the Name Server.

FilsetHandle

A Name Server object handle which points to the root node of the fileset.

FilesetId

The unique identifier of this fileset.

FilesetName

The unique human readable fileset name.

FilesetType

Specifies the type of the fileset the attributes are for. This is a read-only field. The following constants define the fileset types:

NS_FS_TYPE_HPSS_ONLY	This fileset is an HPSS-only fileset.
NS_FS_TYPE_ARCHIVED	This fileset is a backup copy of some other fileset.
NS_FS_TYPE_DFS_ONLY	This fileset is native to some other file system such as DFS.
NS_FS_TYPE_MIRRORED	This fileset is a mirrored copy of some other fileset such as a DFS fileset.

GatewayUUID

The identifier of the gateway that processes DMAP requests for the fileset.

StateFlags

The flags that defined the state of the fileset. Valid values include:

NS_FS_STATE_READ	The fileset allows reads.
NS_FS_STATE_WRITE	The fileset allows writes.
NS_FS_STATE_DESTROYED	The fileset allows no access.

SubSystemId

CDS name of the HPSS location server. This field is not currently used.

UserData

Uninterpreted data supplied by the client. This data can be ASCII, binary, or both.

DirectoryCount

The current number of directories in the fileset.

FileCount

The current number of files in the fileset.

HardLinkCount

The current number of hard links in the fileset.

JunctionCount

The current number of junctions in the fileset.

SymLinkCount

The current number of symbolic links in the fileset.

2.3.8 Name Server Object Handle Structure - ns_ObjHandle_t

Description

The Name Server object handle structure contains information that allows the Name Server to identify the SFS record where the metadata for the object is stored.

Format

The Name Server object handle structure has the following format:

```
typedef struct {
   unsigned32        ObjId;
   unsigned32        FileId;
   unsigned char     Flags;
   unsigned char     Pad1;
   unsigned char     Pad2;
   unsigned char     Pad3;
   unsigned16        Generation;
   unsigned char     Type;
   unsigned char     Version;
   uuid_t            NameServerUUID;
} ns_ObjHandle_t;
```

ObjId

Specifies a unique Name Server object identifier. (The Relative Sequence Number (RSN) of the SFS record containing the metadata for the object.)

FileId

If the Type field specifies a hardlink this is the RSN of the SFS record containing the metadata for the original file. For all other Types this field is equal to the ObjId.

Flags

Specifies a bit vector whose bits convey additional information about the object handle. The defined bit positions for the Flags field are:

NS_OH_FLAG_FILESET_ROOTHandle is for the root node of a fileset.

Pad1

Reserved for future use.

Pad2

Reserved for future use.

Pad3

Reserved for future use.

Generation

Specifies a random number used to detect stale object handles.

Type

Specifies the ‘type’ of the object: file, directory, junction, symbolic link, or hard link.

Version

Specifies the Name Server version number.

NameServerUUID

Specifies the UUID of the Name Server that issued this object handle.

2.3.9 Name Server Directory Entry - ns_DirEntry_t

Description

The Name Server directory entry structure defines the contents of a Name Server directory entry.

Format

The Name Server directory entry structure has the following format:

```
typedef struct DirEntryTag {
   ns_ObjHandle_t        ObjHandle;
   unsigned char         Name[HPSS_MAX_FILE_NAME];
   unsigned32            ObjOffset;
   struct DirEntryTag    *Next;
   ns_Attrs_t            Attrs;
} ns_DirEntry_t;
```

ObjHandle

Specifies the Name Server object handle of the directory entry.

Name

Specifies the name of the directory entry.

ObjOffset

Specifies the offset of the entry within the directory.

Next

Points to the next directory entry.

Attrs

Specifies attributes of the directory entry.

2.3.10 Bitfile Volatile and Metadata Attributes - bf_attrib_t

Description

The attributes structure for the bitfile object contains all the volatile and metadata bitfile attributes. These are parameters relating to a bitfile.

Format

The bitfile attributes structure has the following format:

```
typedef struct bf_attrib {
   u_signed64       CurrentPosition;
   signed32         OpenCount;
   unsigned32       FamilyId;
   bf_attrib_md_t   BfAttribMd;
} bf_attrib_t;
```

CurrentPosition

Specifies the current byte position in the bitfile.

OpenCount

Specifies the current number of clients that have the bitfile open.

FamilyId

Specifies the family identifier for the bitfile.

BfAttribMd

Specifies the structure of bitfile metadata attributes that are stored in the data base.

2.3.11 Bitfile Volatile and Metadata Extended Attributes - bf_xattrib_t

Description

The attributes structure for the bitfile object contains all the volatile and metadata bitfile extended attributes. These are parameters relating to a bitfile and location of vaild data.

Format

The bitfile extended attributes structure has the following format:

```
typedef struct bf_xattrib {
   u_signed64       CurrentPosition;
   signed32         OpenCount;
   unsigned32       FamilyId;
   bf_sc_attrib_t   SCAttrib[HPSS_MAX_STORAGE_LEVELS];
   bf_attrib_md_t   BfAttribMd;
} bf_xattrib_t;
```

CurrentPosition

Specifies the current byte position in the bitfile.

OpenCount

Specifies the current number of clients that have the bitfile open.

FamilyId

Specifies the family identifier for the bitfile.

SCAttrib

Specifies the storage class attributes at each valid level in the hierarchy.

BfAttribMd

Specifies the structure of bitfile metadata attributes that are stored in the data base.

2.3.12 Bitfile Metadata Attributes - bf_attrib_md_t

Description

This structure contains the bitfile attributes metadata. These are parameters relating to a bitfile.

LinkCount is always 1 for a existing bitfile in currrent HPSS release. On one bfs_Bitfile(Open)SetAttrs call, reverse maps (OwnerRec) can be either added or deleted. Both cannot be accomplished on the same call.

Format

The bitfile attributes metadata structure has the following format:

```
typedef struct bf_attrib_md {
   u_signed64           DataLen;
   signed32             ReadCount;
   signed32             WriteCount;
   signed32             LinkCount;
   timestamp_sec_t      CreateTime;
   timestamp_sec_t      ModifyTime;
   timestamp_sec_t      WriteTime;
   timestamp_sec_t      ReadTime;
   unsigned32           COSId;
   unsigned32           NewCOSId;
   acct_rec_t           Acct;
   unsigned32           Flags;
   unsigned32           StorageSegMult;
   bfs_owner_rec_t      OwnerRec;
   u_signed64           RegisterBitmap;
   unsigned32           CellId;
} bf_attrib_md_t;
```

DataLen

Specifies the number of bytes of actual data that the bitfile contains.

ReadCount

Specifies the count of the number of times that all or part of the bitfile has been read.

WriteCount

Specifies the count of the number of times that data has been written to the bitfile.

LinkCount

Specifies the number of links to this bitfile by Name Servers. This also indicates how many reverse map IDs are in the bf_rev_map record for this bitfile.

CreateTime

Specifies the date and time the bitfile was created.

ModifyTime

Specifies the date and time the bitfile was last modified.

WriteTime

Specifies the date and time when data was last written to the bitfile.

ReadTime

Specifies the date and time when the bitfile was last read.

COSId

Specifies the class of service type (unsigned32) and indicates which of several classes of service the bitfile is in. This ID references a class of service record that defines the parameters for this particular class of service. When changing a file's COS, this field is used by the hpss_FileSetAttributes function.

NewCOSId

Indicates the new class of service that a file is to be changed to when the client changes the class of service on a bitfile. When the change has been completed, the value of this field is moved into COSId and this field is cleared. This field is read only. Use the COSId file to change a file's COS.

Acct

Specifies the accounting metadata for the bitfile. It includes information needed to charge for data storage, access, transfers, quotas, etc.

Flags

Contains the flag settings. Not currently used.

StorageSegMult

Storage segment multiple used to adjust size of disk storage segments.

OwnerRec

Defines the reverse map entries for a bitfile and indicates which ones are active or NULL.

RegisterBitmap

Used to indicate the attributes that the SSM wants to receive notifications for when the attributes change.

BFS_REG_OPEN_COUNT

BFS_REG_DATA_LEN

BFS_REG_READ_COUNT

BFS_REG_WRITE_COUNT

BFS_REG_LINK_COUNT

BFS_REG_CREATE_TIME

BFS_REG_MODIFY_TIME

BFS_REG_WRITE_TIME

BFS_REG_READ_TIME

BFS_REG_OWNER_REC

BFS_REG_COS_ID

BFS_REG_ACCT

BFS_REG_SECURITY

This vector is also set to indicate which fields in the attributes structure have changed on notify requests. If the REG_OWNER_REC field is set, then the SetRevMapFlags field in the bf_attrib struct will be set to indicate which reverse map entries have changed.

CellId

The client DCE cell identifier.

2.3.13 Bitfile Owner Record - bfs_owner_rec_t

Description

This structure defines the reverse map entries for a bitfile and indicates which ones are active or NULL.

Format

The bitfile owner record has the following format:

```
typedef struct bfs_owner_rec {
   signed32         RevMapCount;
   unsigned32       Pad;
   rev_map_t        RevMap[BFS_NUM_REV_MAPS];
} bfs_owner_rec_t;

typedef struct rev_map {
   byte             RevMapId[BFS_REV_MAP_LEN];
} rev_map_t;

BFS_REV_MAP_LEN = 32;
```

RevMapCount

Specifies the number of valid reverse map entries.

Pad

Specifies the pad for 64-bit alignment.

RevMap[BFS_NUM_REV_MAPS]

Specifies the array of opaque reverse mapping fields supplied by a client.

2.3.14 Bitfile Server Storage Class Attributes - bf_sc_attrib_t

Description

This structure contains storage class information for a specific storage hierarchy level at which the specified bitfile exists.

Format

The Bitfile Server storage class attributes have the following format:

```
typedef struct bf_sc_attrib {
   bf_vv_attrib_t   VVAttrib[BFS_MAX_VV_TO_RETURN_AT_LEVEL];
   unsigned32       NumberOfVVs;
   u_signed64       BytesAtLevel;
   unsigned32       OptimumAccessSize;
   unsigned32       StripeWidth;
   u_signed64       StripeLength;
   unsigned32       Flags;
} bf_sc_attrib_t;
```

VVAttrib

An array of virtual volume on which bitfile segments are contained.

NumberOfVVs

Specifies the number virtual volume entries in the array.

BytesAtLevel

Specifies the amount of data that exist at this level (in bytes).

OptimumAccessSize

Specifies the optimum access size of the storage class.

StripeWidth

Specifies the stripe width of the storage class.

StripeLength

Specifies the stripe length of the storage class.

Flags

The flags that defined the state of the fileset. Valid values include:

BFS_BFATTRS_LEVEL_IS_DISK	This is a disk storage level.
BFS_BFATTRS_LEVEL_IS_TAPE	This is a tape storage level.
BFS_BFATTRS_DATAEXIST_AT_LEVEL	Data for the bitfile exists at this level.
BFS_BFATTRS_ADDITIONAL_VV_EXIST	Data at this level is contained on more than `BFS_MAX_VV_TO_RETURN_AT_LEVEL` virtual volumes.

2.3.15 Bitfile Server Virtual Volume Attributes - bf_vv_attrib_t

Description

This structure contains Bitfile Server virtual volume attributes for a specific storage level in the hierarchy.

Format

The bitfile virtual volume attributes have the following format:

```
typedef struct bf_vv_attrib {
   hpssoid_t        VVID;
   signed32         RelPosition;
   u_signed64       BytesOnVV;
   pv_list_t        *PVList;
} bf_vv_attrib_t;
```

VVID

Specifies the virtual volume identifier.

RelPosition

Specifies the relative start position of first bitfile segment on this virtual volume.

BytesOnVV

Specifies the number of bytes of the given file stored on this VV from the current contiguous set of bitfile segments. It is possible for BytesOnVV to be less than the total number of bytes of the given file that reside on this VV if non-adjacent bitfile segments are stored on the same VV. For example, if a file consists of three 1MB bitfile segments, with the first and third on VV 'A' and the second on VV 'B', an hpss_FileGetXAttributes() call will return a list of three VVs (bf_vv_attrib_t types), each with BytesOnVV = 1MB. On the other hand, if a file consists of three 1MB bitfile segments, with the first and second on VV 'A' and the third on VV 'B' an hpss_FileGetXAttributes() call will return a list of two VVs, the first with BytesOnVV = 2MB and the second with BytesOnVV = 1MB.

PVList

A conformant array of physical volume attributes.

2.3.16 Storage Server Physical Volume Attributes - pv_list_element_t

Description

This structure contains physical volume location information for specified physical volume.

Format

The storage server physical volume attributes have the following format:

```
typedef struct pv_list_element {
   char             Name[HPSS_PV_NAME_SIZE];
   unsigned32       Flags;
} pv_list_element_t;
```

Name

Specifies the physical volume name.

Flags

Specifies the location of the physical volume. This field will be zero if the physical volume is in the robot, or the bit corresponding to the value **PV_ON_SHELF** will be set if the physical volume has been shelved.

2.3.17 Storage Server Physical Volume Attributes Conformant Array - pv_list_t

Description

The pv_list_t structure describes a template for a conformant array of Storage Server Physical Volume Attribute elements.

Format

The storage server physical volume attribute conformant array has the following format:

```
typedef struct pv_list {
   signed32             Length;
   pv_list_element_t*   List[*];
} pv_list_t;
```

Length

Specifies the number of physical volume attribute elements in the array.

List

A conformant array of physical volume attribute elements.

2.3.18 Bitfile Server Statistics - bfs_stats_t

Description

This structure contains statistical information which includes a count of stages, migrates, purges, and deletions. In addition, the structure includes a timestamp indicating when the counts began.

Format

The Bitfile Server statistics have the following format:

```
typedef struct bfs_stats {
   unsigned32       StageCount;
   unsigned32       MigrationCount;
   unsigned32       PurgeCount;
   unsigned32       DeleteCount;
   timestamp_sec_t  TimeLastReset;
} bfs_stats_t;
```

StageCount

Specifies the number of stages which have occurred since the last reset.

MigrationCount

Specifies the number of migrations which have occurred since the last reset.

PurgeCount

Specifies the number of purges which have occurred since the last reset.

DeleteCount

Specifies the number of deletes which have occurred since the last reset.

TimeLastReset

Specifies the time of the last reset (all counts were set to 0).

2.3.19 Account Record - acct_rec_t

Description

The account record contains the HPSS account identifier number.

Format

The API configuration structure has the following format:

```
typedef unsigned32 acct_rec_t;
```

2.3.20 API Configuration Structure – api_config_t

Description

The API configuration structure contains values that control optional features of the Client API configuration.

Format

The API configuration structure has the following format:

```
typedef struct api_config {
   long                 Flags;
   long                 DebugValue;
   long                 TransferType;
   long                 NumRetries;
   int                  BusyDelay;
   int                  BusyRetries;
   int                  TotalDelay;
   int                  LimitedRetries;
   long                 MaxConnections;
   int                  ReuseDataConnections;
   int                  UsePortRange;
   long                 RetryStageInp;
   int                  DMAPWriteUpdates;
   char                 ServerName[HPSS_MAX_DCE_NAME];
   char                 DescName[HPSS_MAX_DESC_NAME];
   char                 PrincipalName[HPSS_MAX_PRINCIPAL_NAME];
   char                 KeytabPath[HPSS_MAX_PATH_NAME];
   char                 DebugPath[HPSS_MAX_PATH_NAME];
   char                 HostName[HPSS_MAX_HOST_NAME];
   char                 RegistrySiteName[HPSS_MAX_DCE_NAME];
} api_config_t;
```

Flags

Contains a bitmap of configuration flags. Valid values include:

API_INIT_HSEC Client API should perform HPSS security initialization.

API_INIT_CONN Client API should perform connection initialization.

API_INIT_TRPC Client API should perform TRPC initialization.

API_ENABLE_LOGGING If logging compiled into Client API library, perform HPSS logging on errors.

API_GLOBAL_FILETABLE Indicates whether Client API was built using global or per-thread file table (this flag is informational only - it cannot be set using hpss_SetConfiguration).

API_USE_ENV Modify configuration based on environment variables.

API_DISABLE_CROSS_CELL If set, this flag prevents the Client API from contacting any servers outside of the local cell. Once set, this flag cannot be unset. This flag is set automatically when FTP or NDCG log-in without security.

API_DISABLE_JUNCTIONS If set, this flag prevents the Client API from processing any requests which require it to traverse a junction. Once set, this flag cannot be unset. NFS will always set this flag explicitly.

DebugValue

If zero, indicates that Client API will not send debug messages to an output file; otherwise messages will be sent (note that all debug messages are conditionally compiled into the library).

TransferType

Indicates what data transfer mechanism is to be used for transfers handled by the Client API. The only valid value is:

API_TRANSFER_TCP Use TCP/IP

NumRetries

Used to control the number of retries to attempt when an operation fails. Currently this class of operation includes library initialization and communication failures. A value of zero indicates that no retries are to be performed and a value of negative one indicates that operation will be retried until successful.

BusyDelay

Used to control the number of seconds to delay between retry attempts.

BusyRetries

Used to control the number of retries to be performed when a request fails because the Bifile Server does not currently have an available thread to handle that request. A value of zero indicates that no retries are to be performed. A value of negative one indicates that retries should be attempted until either the request succeeds of fails for another reason.

TotalDelay

Used to control the number of total seconds to continue retrying a request.

LimitedRetries

Used to control the number of retry attempts for limited retry type errors.

MaxConnections

Maximum number of connections for use by the HPSS connection management service.

ReuseConnections

Used to control whether TCP/IP connections are to be left as long as a file is opened or are to be closed after each read or write request. A non-zero value will cause connections to remain open, while a zero will cause connections to be closed.

UsePortRange

Used to control whether the HPSS Mover(s) should use the configured port range when making TCP/IP connections for read and write requests. A non-zero value will cause the Mover(s) to use the port range. A value of zero will cause the Mover(s) to allow the operating system to select the port number.

RetryStageInp

Used to control whether retries are attempted on opens of files in a Class of Service that is configured for background staging on open. A non-zero value indicates that open which would return -EINPROGRESS to indicate the file is being staged will be retried. A value of zero indicates that the -EINPROGRESS return code will be returned to the client.

DMAPWriteUpdates

Controls the frequency of cache invalidates that are issued to the XDSM file system.

ServerName

Name to use when initializing HPSS security services.

DescName

Name to use when generating HPSS log messages.

PrincipalName

DCE principal name to use for HPSS security initialization.

KeytabPath

Pathname of the DCE security keytab file.

DebugPath

If generation of debug message is enabled, the pathname of the file to which log messages will be directed. Special cases are "stdout" and "stderr".

HostName

Specifies the interface name to use for TCP/IP communications.

RegistrySiteName

Specifies the security registry used when inserting security information into connection binding handles.

2.3.21 Name Server ACL Conformant Array - ns_ACLConfArray_t

Description

The ns_ACLConfArray_t structure describes a template for a conformant array of Name Server ACL entries.

Format

The ns_ACLConfArray_t structure has the following format:

```
typedef struct {
   signed32             Length;
   ns_ACLEntry_t        ACLEntry[*];
} ns_ACLConfArray_t;
```

Length

Specifies the number of ACL entries in the array.

ACLEntry

The array of ACL entries.

2.3.22 Name Server Access Control List Entry - ns_ACLEntry_t

Description

The ns_ACLEntry_t structure describes a Name Server ACL entry. Each entry contains information such as the type of entry (i.e., for a group or individual user), the identity and location of the user or group and the permissions that are allowed.

Format

The ns_ACLEntry_t structure has the following format:

```
typedef struct ACLEntryTag {
   unsigned char        EntryType;
   unsigned char        Perms;
   unsigned16           ExpirationDate;
   unsigned32           EntryId;
   unsigned32           Location;
   struct ACLEntryTag   *Next;
} ns_ACLEntry_t;
```

EntryType

Identifies the type of this ACL entry. These correspond to the DFS ACL tag types: user_obj, user_obj_delegate, user, user_delegate, foreign user, foreign_user_delegate, group_obj, group_obj_delegate, group, group_delegate, foreign_group, foreign_group_delegate, other_obj, other_obj_delegate, foreign_other, foreign_other_delegate, any_other, any_other_delegate, mask_obj, or unauthenticated.

Perms

Specifies the permissions or access rights.

ExpirationDate

Currently not used.

EntryId

Depending on the EntryType, it can specify an identifier (usually a UID or GID).

Location

Specifies the identifier of the DCE cell.

Next

Points to the next ACL Entry.

2.3.23 Global Fileset Entry Structure – hpss_global_fsent_t

Description

The global fileset entry structure contains global fileset information.

Format

The global fileset entry structure has the following format:

```
typedef struct {
       u_signed64                FilesetId;
       unsigned char             FilesetName[HPSS_MAX_FS_NAME];
       uuid_t                    GatewayUUID;
       uuid_t                    NameServerUUID;
} hpss_global_fsent_t;
```

FilesetId

The unique fileset identifier.

FilesetName

The unique name of the fileset.

GatewayUUID

The identifier of the DMAP gateway that manages the fileset.

NameServerUUID

The identifier of the Name Server that manages the fileset.

2.3.24 Name Server Fileset Attribute Bits – ns_FilesetAttrBits_t

Description

Bits specifying the Name Server fileset attribute bits to retrieve or set.

Format

```
typedef u_signed64    ns_FilesetAttrBits_t;
```

2.3.25 Name Server Object Attribute Bits – ns_FilesetAttrBits_t

Description

Bits specifying the Name Server object attributes to retrieve or set.

Format

```
typedef u_signed64    ns_AttrBits_t;
```

2.3.26 Purge Lock Flag - purgelock_flag_t

Description

Flag specifying whether a file should have its purgelock status set or cleared.

Format

```
typedef enum {
    PURGE_UNLOCK = 0,   /* purge unlock the file */
    PURGE_LOCK          /* purge lock the file   */
} purgelock_flag_t;
```

2.3.27 API Name Specification – api_namespec_t

Description

The API Name Specification structure is used for converting HPSS cell and principal ids to/from their associated names. The API Name specification structure contains the translation type along with the principal and cell information that is to be converted.

Format

```
typedef struct api_namespec {
    namespec_type_t   Type;
    int               Id;
    int               CellId;
    char              Name[HPSS_MAX_DCE_NAME];
    char              CellName[HPSS_MAX_DCE_NAME];
} api_namespec_t
```

Type

The type of translation that is requested. The valid values are:

NAMESPEC_CELL	- Translate cell information only
NAMESPEC_USER	- Translate user and cell information
NAMESPEC_GROUP	- Translate group and cell information
NAMESPEC_SKIP	- Do not translate this entry

Id

The uid or gid of the principal.

CellId

The HPSS cell id where the principal resides.

Name

The name of the principal.

CellName

The name of the cell where the principal resides.

2.3.28 Bitfile Callback Address – bfs_callback_addr_t

Description

The Bitfile Callback Address structure contains the host information, port number and an identification number which facilitates call backs during a stage process.

Format

```
typedef struct bfs_callback_addr {
    unsigned32        addr;
    unsigned16        port;
    unsigned16        family;
    signed32          id;
} bfs_callback_addr_t;
```

addr

Host address in network byte order.

port

Port number.

family

Address family.

id

Id to be returned during a callback.

2.3.29 HPSS Directory Entry – hpss_dirent_t

Description

The HPSS directory entry structure contains the directory's name and a handle to the Name Server for this directory. In addition, it contains the offset of the next directory entry.

Format

```
typedef struct hpss_dirent {
    unsigned32        d_offset;
    ns_ObjHandle_t    d_handle;
    unsigned16        d_reclen;
    unsigned16        d_namelen;
    char              d_name[HPSS_MAX_FILE_NAME];
} hpss_dirent_t
```

d_offset

The offset of the next directory entry.

d_handle

The handle to the Name Server for the directory.

d_reclen

The record length of the directory.

d_namelen

The number of characters in the directory name.

d_name

The name of the directory.

2.3.30 HPSS Security User Credentials – hsec_UserCred_t

Description

The HPSS Security User Credentials structure contains information about the user credentials. This information can only be obtained by an authorized client using HPSS security mechanisms.

Format

```
typedef struct hsec_UserCred {
    SecPWent_t        SecPWent;
    uuid_t            SecUuid;
    unsigned32        SecLabel;
    acct_rec_t        DefAccount;
    acct_rec_t        CurAccount;
    unsigned32        NumGroups;
    long              AltGroups[HPSS_NGROUPS_MAX];
    unsigned32        DCECellId;
} hsec_UserCred_t
```

SecPWent

Security information about the principal.

SecUuid

Secure Unix user identification number.

SecLabel

Secure label that is associated with the SecUuid.

DefAccount

The accounting code that is used when a current account code has not been specified.

CurAccount

When a current accounting code is specified, this code is applied to new files or directories.

NumGroups

The number of groups to which this principal is a member.

AltGroups

An array of groups to which this principal is a member.

DCECellId

The DCE Cell Id where the principal resides

2.3.31 Security Password Entry – SecPWent_t

Description

The Security Password Entry structure provides information needed to lookup a principal in the HPSS security registry.

Format

```
typedef struct hsec_PWent {
    char              Name[HPSS_MAX_USER_NAME];
    unsigned32        Uid;
    unsigned32        Gid;
} SecPWent_t
```

Name

Name of the principal

Uid

User id of the principal

Gid

Group id of the principal

Chapter 3

I/O Descriptor (IOD) and I/O Reply (IOR)

3.1 I/O Descriptor Purpose

The I/O Descriptor (IOD) is used to describe I/O requests within HPSS. The IOD contains the information required to describe the function requested and the data sources and sinks for the request. The IOD is a common structure that will be passed between Client API and Bitfile Server, Bitfile Server and Storage Server, Storage Server and Mover, as well as Physical Volume Library and Mover.

In the general case of an HPSS client data transfer request, the Client API will send an IOD which describes the client data address(es) for the transfer, as well as the piece(s) of the HPSS file requested. The HPSS components will perform a series of mappings on the HPSS side, until the actual data source or sink location (for a client read or write request, respectively) is determined. The Mover(s) will then use the client addressing information to perform the data transfer.

For release 3, a capability is provided for the mover to reply with listen port addressing information as an intermediate reply. Also, a flag value is provided to indicate a Mover-to-Mover protocol, which provides transport selection and flow control between movers.

3.2 I/O Reply Purpose

The I/O Reply (IOR) is used to return the state of a request at a particular moment. If the request has completed (due to correct completion or an unrecoverable error), the IOR will contain the final completion status of the request.

An IOR will be returned at the completion of a request from each HPSS component that received an IOD with the request. An IOR will also be returned as the result of a request to query the status of a request, from each HPSS component that received an IOD with the initial request.

3.3 I/O Descriptor Components

The I/O Descriptor consists of these major parts:

- Request Description
- Source Descriptor List
- Sink Descriptor List

The Request Description contains information describing the request to be performed. This information includes the function (e.g., read, write, set position), any flags and/or subfunction information required to completely define the operation, and any other information required by the operation which is not a description of the data source or sink.

The Source Descriptor List contains information which describes the source of a data transfer.

The Sink Descriptor List contains information which describes the sink of a data transfer.

3.4 I/O Reply Components

The I/O Reply consists of these major parts:

- Request State
- Source State List
- Sink State List

The Request State contains information describing the status of the request as a whole. This information will indicate whether the request is in progress, completed, waiting on a resource, encountered an unrecoverable error, etc.

The Source State List contains information describing the status of each source descriptor involved in the I/O operation. Information will include overall status (e.g., complete, in progress), device positioning information (if applicable), listen addresses (if applicable) and the number of bytes transferred.

The Sink State List contains information describing the status of each sink descriptor involved in the I/O operation. Information will include overall status (e.g., complete, in progress), device positioning information (if applicable), listen addresses (if applicable) and the number of bytes transferred.

3.5 Data Definitions

This section describes key internal data definitions and all externally used data definitions for the IOD and IOR.

3.5.1 I/O Descriptor (IOD) - IOD_t

Description

The IOD is a structure that is used to describe I/O requests. This structure contains parameters to describe the function requested and the sources and sinks for the operation.

Format

The IOD has the following format:

```
typedef struct IOD {
     signed32          RequestID;
     signed32          Function;
     unsigned32        Flags;
     requestspec_t     *ReqSpecInfo;
     signed32          SrcDescLength;
     signed32          SinkDescLength;
     srcsinkdesc_t     *SrcDescList;
     srcsinkdesc_t     *SinkDescList;
} IOD_t;
```

RequestID

This field contains a request identifier used to distinguish requests from a given client.

Function

This field indicates the type of I/O operation being requested. The following values are valid for this field:

IOD_READ	Data is to be transferred to the initiator from the responder.
IOD_WRITE	Data is to be transferred from the initiator to the responder.
IOD_DEVICESPECIFIC	Device specific request (e.g., Write Tapemark).
IOD_GETDEVICEATTR	Query device attributes (Storage Server / Mover only).
IOD_SETDEVICEATTR	Set device attributes (Storage Server / Mover only).
IOD_ABORT	Abort outstanding request (Storage Server / Mover only).

Flags

This field is a bit vector used to alter processing of the request. The possible bits that may be set are:

NO_LABEL_CHECK	Override the default label check processing.
REPLYWHENREADY	Requests that the server ready listen ports and reply with addressing information to those ports after they are established.
HOLD_RESOURCES	For device specific and device attribute requests, indicates that the device is to be held open by the mover task after completion of the request. Otherwise, the device is freed when the request is

completed.

ReqSpecInfo

This structure contains information pertaining to information specific to the type of request.

```
     typedef struct requestspec {
     signed32    SubFunction;
     signed32    Argument;
     signed32    DeviceID;
     u_signed64  Count;
     u_signed64  SelectionFlags;
     struct {
           signed32InfoType;
           union {
                 signed32Reserved;
                 char  DisplayBuffer[16];
                 devdesc_attr_tDeviceAttr;
                 char  VolumeID[16];
                 address_tReplyAddr;
           } ReqInfo_u;
     } ReqInfo_s;
} requestspec_t;
```

SubFunction

This field indicates the device specific function to be performed. The following values are valid for this field:

DEVICE_LOAD	Load a physical volume into a drive.
DEVICE_UNLOAD	Unload a physical volume from a drive.
DEVICE_FLUSH	Ensure data previously written is flushed to the media.
DEVICE_WRITETM	Write tape mark (tape only).
DEVICE_LOADDISPLAY	load message to device's display area.
DEVICE_READLABEL	Read media label.
DEVICE_WRITELABEL	Write media label.
DEVICE_CLEAR	Zero portion of the media (disk only).

Argument

This field contains any additional information that is required to perform the requested operation. This field varies based on request and is used internally.

DeviceID

This field describes the device to which the requested operation is to be performed. This field is also used to specify the request ID for request status queries.

Count

This field contains the number of iterations to perform of the requested operation.

SelectionFlags

This field describes the fields within the DeviceAttr field that are to be updated during a device set attributes request.

ReqInfo_s

Generated by the DCE IDL compiler, this structure contains a typed union used to pass request specific information.

InfoType

This value will indicate what information is included in ReqInfo_u. Valid values include:

INFO_NONE	Information union is unused for this request.
INFO_LOADDISPLAY	Information is a buffer to be output on device display area for DEVICE_LOADDISPLAY requests.
INFO_DEVICEATTR	Information is a device attribute structure for DEVICE_SETDEVICEATTR requests.
INFO_VOLUMEID	Information is a volume label.
INFO_REPLYADDR	Information is a reply network address.

ReqInfo_u

This union contains the specific information required to complete the requested operation. The union elements include:

Reserved

This field is a place holder for requests that do not require any request specific information.

DisplayBuffer

This field contains the character string to be output on the indicated devices display area.

DeviceAttr

This structure is used for the Mover's get device characteristics and set device characteristics. See the Mover design document for further details.

VolumeID

This field contains the volume label to be written during a write label request.

ReplyAddr

This field contains the address to which the mover should reply with listen port addressing information.

SrcDescLength

This field contains the number of items in SrcDescList.

SinkDescLength

This field contains the number of items in SinkDescList.

SrcDescList

This list of structures defines the sources of a data transfer. See the description of the source/sink descriptor, below.

SinkDescList

This list of structures defines the sinks of a data transfer. See the description of the source/sink descriptor, below.

3.5.2 Source/Sink Descriptor - srcsinkdesc_t

Description

The source/sink descriptor contains information describing a contiguous segment of data within a request. The structure contains addressing information, which varies depending on which module is currently handling the request.

Format

The Source/Sink Descriptor has the following format:

```
typedef struct srcsinkdesc {
      unsigned32              Flags;
      u_signed64              Offset;
      u_signed64              Length;
      address_t               SrcSinkAddr;
      struct srcsinkdesc      *Next;
signed32                      ServerDefined;
} srcsinkdesc_t;
```

Flags

This field contains flags which affect the portion of the request defined by this descriptor.

HOLD_RESOURCES	When set, resources are not freed at the completion of the current request. At the Storage Server level, this bit may be used to keep removable media mounted across data access requests; at the Mover level this bit may be used to keep network connections or devices open across requests.

CONTROL_ADDR	When set, use mover protocol to perform data transfer. The address information contained in this source/sink descriptor will be the peer mover listen port addressing information. Note that for striped addresses, this flag must be set in the stripe address entry itself.
XFER_RESPONDER	When set, indicates that the client is requesting to be the responder for the part of the data transfer corresponding to this descriptor.
XFEROPT_IP	When set, indicates that the client is capable of transferring data using TCP/IP.
XFEROPT_SHMEM	When set, indicates that the client is capable of transferring data using a shared memory segment.
USE_PORT_RANGE	When set, indicates that the Mover should use the configured TCP/IP port range when making connections to the client.
LAST_SEG_WRITE	When set, indicates that the write operation is the last that will be written to the tape storage segment. The Tape Storage Server and Mover use this flag to enhance performance by writing tape marks and advancing to the next tape section after transferring the data, rather than doing these steps in a separate tape write operation.

Offset

This field contains the offset within the request (described by the IOD) at which the data described by this source/sink descriptor begins. This field will be used in determining the data tag and coordinating the source and sink lists.

Length

The length of the data, in bytes, described by this source/sink descriptor.

SrcSinkAddr

This field contains addressing information for the data described by this source/sink descriptor. See the description of the address structure, below.

Next

This field contains a pointer to the next descriptor in the source or sink list. This field is necessary for use by the DCE encoding code.

ServerDefined

This field contains a value, which is provided by the client, which will be returned upon completion of the request. The receiving server does not examine this field in any way.

3.5.3 Address Structure - address_t

Description

The address structure contains addressing information that will be included in the source/sink descriptor

to define the sources and sinks of data transfers. The structure contains a union of possible types of addresses; which addressing is used varies depending on the configuration (primarily for network addressing) and which HPSS component is acting on the structure.

Format

The Address Structure has the following format. The IPIAddr address type is no longer supported.

```
typedef struct address {
      signed32Type;
      union {
            netaddress_t            NetAddr;
            ipiaddress_t            IPIAddr;
            piofsaddress_t          PIOFSAddr;
            fileaddress_t           FileAddr;
            ssegaddress_t           SSegAddr;
            vvaddress_t             VVolAddr;
            pvaddress_t             PVolAddr;
            devaddress_t            DevAddr;
            memaddress_t            MemAddr;
            stripeaddress_t         StripeAddr;
            clientfileaddress_t     ClientFileAddr;
            shmaddress_t            ShmAddr;
            lftaddress_t            LFTAddress;
            } Addr_u;
} address_t;
```

Type

This field indicates the type of address contained in Addr_u. Values are:

NET_ADDRESS	TCP/IP
PIOFS_ADDRESS	PIOFS file
FILE_ADDRESS	bitfile
SSEG_ADDRESS	Storage segment
VVOL_ADDRESS	virtual volume
PVOL_ADDRESS	physical volume
DEVICE_ADDRESS	device
STRIPE_ADDRESS	stripe
MEMORY_ADDRESS	memory buffer
CLIENTFILE_ADDRESS	client file
SHM_ADDRESS	shared memory segments
LFT_ADDRESS	local file transfer

NetAddr

This structure contains IP addressing information, and will be primarily used as the addressing information that the Mover must use to connect to the client mover. The structure has the following format:

```
typedef struct netaddress {
      u_signed64  SockTransferID;
      struct {
            unsigned32  addr;
            unsigned16  port;
```

```
            unsigned16  family;
      } SockAddr;
      u_signed64SockOffset;
} netaddress_t;
```

SockTransferID

This field contains the transfer ID which will be used in the data tag when the Mover connects to the client mover for device transfers. It will be used to identify and verify the data being transferred.

SockAddr

This field contains the IP address information as usually represented by the standard sockaddr structure. It identifies the address to which the Mover will connect to perform the data transfer. The fields in this structure directly correspond to the fields in the standard structure and are assumed to be in network byte order.

SockOffset

This field contains the offset to be used in the data tag when communicating with the client mover to perform the data transfer. If the logical offset within the entire request is enough information, this field may not be necessary.

PIOFSAddr

This field contains the information necessary to describe a piece of a logical partition of a PIOFS file. The structure has the following format:

```
typedef struct piofsaddress {
      u_signed64 Offset;
      unsigned32 Flags;
      unsigned32 Perms;
      unsigned32 Vbs;
      unsigned32 Vn;
      unsigned32 Hbs;
      unsigned32 Hn;
      unsigned32 SubFile;
      unsigned32 ChkptFlag;
      char       Name[255];
} piofsaddress_t;
```

Reference *Installing, Managing, and Using the IBM AIX Parallel I/O File System* for details of this structure.

Offset

This field contains the offset within the PIOFS subfile.

Flags

This field contains flags which affect the operation of PIOFS import/export operations.

Perms

This field contains the permissions to be used when opening the PIOFS file.

Vbs

This field contains the PIOFS vertical block size.

Vn

This field contains the number of vertical partitions within the PIOFS subfile.

Hbs

This field contains the PIOFS horizontal block size.

Hn

This field contains the number of horizontal partitions within the PIOFS subfile.

SubFile

This field contains the PIOFS subfile identifier.

ChkptFlag

This field indicates whether PIOFS will checkpoint while performing the import or export.

Name

This field contains the name of the PIOFS file.

FileAddress

This field contains information which describes a contiguous piece of an HPSS bitfile. This information will be sent by the Client API to the Bitfile Server with a client read or write request. The Bitfile Server will map this information into a series of logical segment addresses. This structure has the following format:

```
typedef struct fileaddress {
      hpss_object_handle_t    BitFileHandle;
      u_signed64              BitFileOffset;
} fileaddress_t;
```

BitFileHandle

This field contains the open bitfile handle for which this request applies.

BitFileOffset

This field contains the offset into the bitfile at which the data described by the source/sink descriptor begins.

SSegAddr

This structure contains storage segment addressing information. It will be mapped into virtual volume addressing information. The structure has the following format:

```
typedef struct ssegaddress {
      hpssoid_t   SSegID;
      u_signed64  SSegOffset;
} ssegaddress_t;
```

SSegID

This field contains the storage segment ID.

SSegOffset

This field contains the offset, in bytes, within the storage segment at which the data begins.

VVolAddr

This structure contains virtual volume addressing information. The Storage Server will map this information into physical volume addressing information. The structure has the following format:

```
typedef struct vvoladdress {
      hpssoid_t         VVolID;
      positiondesc_t    VVolPosition;
} vvoladdress_t;
```

VVolID

This field contains the virtual volume ID.

VVolPosition

This field contains the position within the virtual volume at which the data begins. See the Mover design document for further details.

PVolAddr

This structure contains physical volume addressing information. The Storage Server will map this information into device addressing information and pass that on to the appropriate Mover. The structure has the following format:

```
typedef struct pvoladdress {
      char              PVolName[HPSS_PV_NAME_SIZE];
      positiondesc_t    PVolPosition;
} pvoladdress_t;
```

PVolName

This field contains the physical volume name.

PVolPosition

This field contains the position on the physical volume at which the data begins. See the Mover design document for further details.

DevAddr

This structure contains device addressing information. The Mover uses this information to access the requested data. The structure has the following format:

```
typedef struct devaddress {
      unsigned32        Flags;
      signed32          DeviceID;
      signed32          BlockSize;
      signed32          BlocksBetweenTMs;
      char              VolumeID[HPSS_PV_NAME_SIZE];
      positiondesc_t    DevicePosition;
} devaddress_t;
```

Flags

This field contains values which alter the way the device is handled. Valid values include (note that only one of the volume type flags may be specified - i.e., that portion of the Flags field is NOT a bit vector):

MVR_DEV_HPSS_VOL	Indicates that the media loaded on the device is in HPSS format.
MVR_DEV_UNITREE_VOL	Indicates that the media loaded on the device is in UniTree format.
MVR_DEV_VOL_USE_BLK_HDRS	For UniTree formatted media only, indicates that the tape does not include the per block tape headers.

DeviceID

This field contains the device ID.

BlockSize

This field contains the block size, in bytes, to use during write requests.

BlocksBetweenTMs

This field contains the number of blocks to be written between tape marks for this request.

VolumeID

This field contains the volume label for which this request applies. This value is used to check that the expected media is loaded on the requested device.

DevicePosition

This field contains device positioning information describing the location on the device that the data begins. See the Mover design document for further details.

MemAddr

This structure contains information describing a client memory address. The structure has the following format:

```
typedef struct memaddress {
      char  *MemoryPtr;
} memaddress_t;
```

MemoryPtr

This field contains the address of the client's memory buffer.

ClientFileAddr

This structure contains information describing data in a file as represented through the Client API. The structure has the following format:

```
typedef struct clientfileaddress {
      signed32    FileDes;
      u_signed64  FileOffset;
} clientfileaddress_t;
```

FileDes

This field contains the Client API file descriptor, as returned from hpss_Open. See section 2.1.28 for further details.

FileOffset

This field contains the file offset at which the request begins.

ShmAddr

This structure contains information describing a shared memory segment address. The structure has the following format:

```
typedef struct shmaddress {
      unsigned32  Flags;
      unsigned32  ShmID;
      unsigned32  ShmOffset;
} shmaddress_t;
```

Flags

This field contains flags which affect the handling of this address. Valid values include:

SHM_COPY_DATA	indicates that the mover should copy data to or from the shared memory segment (i.e., do not perform device I/O directly from the shared memory segment).

ShmID

This field contains the shared memory segment identifier.

ShmOffset

This field contains the offset within the shared memory segment at which the client data begins

LFTAddress

This structure contains information describing a local file address. The structure has the following format:

```
typedef struct lftaddress {
      u_signed64  Offset;
      unsigned32  Flags;
      unsigned32  Uid;
      unsigned32  Gid;
      unsigned32  CellId;
      char        PathName[HPSS_MAX_PATH_NAME];
} lftaddress_t;
```

Offset

This field contains the offset within the file at which the client data begins.

Flags

This field contains flags which affect the handling of this address.

Currently there are no defined flags for this address type.

Uid

This field contains the client user identifier .

Gid

This field contains the client group identifier.

CellId

This field contains the client DCE Cell identifier.

PathName

This field contains the path name of the file that will be used for the data transfer.

StripeAddr

This structure contains information that describes data striped across a number of units (either devices or network addresses). The structure is intended to allow description of striped data without enumerating each piece of data which makes up the request. The structure has the following format:

```
typedef struct stripeaddress {
      u_signed64              BlockSize;
      u_signed64              StripeWidth;
      signed32                AddrListLength;
      unsigned32              Flags;
      straddress_t            Addr;
      struct stripeaddress_t*Next;
```

```
} stripeaddress_t;
```

BlockSize

This field contains the amount of contiguous data, in bytes, that is written to each member of the stripe group.

StripeWidth

This field contains the number of elements that make up the stripe group. The total amount of data written in one stripe is represented by BlockSize * StripeWidth.

AddrListLength

This field contains the number of addresses contained in the stripe address list. Valid values are 1 (used when passed to a Mover controlling one device in a device stripe) and StripeWidth (used to describe the entire stripe group; usually this will be provided by the initiating end of a transfer to supply the list of available addresses to which the responder must connect to perform the data transfer).

Flags

This field contains values that alter the processing of this address. Valid values include:

XFER_RESPONDER	When set, indicates that the client is requesting to be the responder for the part of the data transfer corresponding to this address.
XFEROPT_IP	When set, indicates that the client is capable of transferring data using TCP/IP.
XFEROPT_SHMEM	When set, indicates that the client is capable of transferring data using a shared memory segment.
USE_PORT_RANGE	When set, indicates that the Mover should use the configured TCP/IP port range when making connections to the client.

Addr

This field contains the addressing information which describes the elements of the stripe group. The IPIAddr address type is no longer supported. The structure has the following format:

```
typedef struct straddress {
      signed32Type;
      union {
            netaddress_t            NetAddr;
            ipiaddress_t            IPIAddr;
            piofsaddress_t          PIOFSAddr;
            fileaddress_t           FileAddr;
            ssegaddress_t           SSegAddr;
            vvaddress_t             VVolAddr;
            pvaddress_t             PVolAddr;
            devaddress_t            DevAddr;
            memaddress_t            MemAddr;
```

```
            clientfileaddress_t     ClientFileAddr;
            shmaddress_t            ShmAddr;
            lftaddress_t            LFTAttress;
      } Addr_u;
} straddress_t;
```

Where each of the fields in the Addr_u union are identical to those described in the address_t structure, above.

Next

This field contains a pointer to the next address in the stripe address list.

3.5.4 I/O Reply (IOR) - IOR_t

Description

The IOR is a structure that is used to describe the state of I/O requests. This structure contains parameters to describe the state of the request as a whole, as well as to describe the state of the individual subtransfers described by source and sink descriptors in the IOD associated with the original request.

Format

The IOR has the following format:

```
typedef struct IOR {
      signed32          RequestID;
      signed32          Flags;
      signed32          Status;
      reqspecreply_t    *ReqSpecReply;
      signed32          SrcReplyLength;
      signed32          SinkReplyLength;
      srcsinkreply_t    *SrcReplyList;
      srcsinkreply_t    *SinkReplyList;
} IOR_t;
```

RequestID

This field contains the value of the RequestID field that was passed with the IOD. This field is used to identify the reply.

Flags

This field is a bit vector used to describe how to interpret the rest of the IOR. Valid values to be used in the vector are:

IOR_COMPLETE	If set, this IOR is the last one associated with the RequestID. Otherwise, there will be additional replies following this IOR.
IOR_ERROR	If set, an error was detected during the processing of the request. The Status field will contain a value that describes the error.
IOR_NOT_PROCESSED	The IOD corresponding to this IOR was not processed.

IOR_GAPINFO_VALID	The request specific reply contains gap (file hole) information.
IOR_FOREIGN_LABEL	The media contains a valid ANSI volume label that was not written by HPSS. This value is for read label requests only.
IOR_NO_LABEL	The media does not contain an 80-byte volume label. This value is for read label requests only.
IOR_NON_ANSI_LABEL	The media contain an 80-byte volume label, but does not meet ANSI specifications. This value is for read label requests only.
IOR_END_OF_SEGMENT	This flag is used by the Tape Storage Server to inform its caller that the tape storage segment referred to by the IOR has reached its end and cannot be written further.

Status

This field contains the status of the request. A value of 0 (zero) indicates that the request was processed or completed without error. Any other value will describe an error condition that occurred during processing of the request.

ReqSpecReply

This structure contains information to describe request specific status of an operation.

```
typedef struct reqspecreply {
      signed32Flags;
      signed32Status;
      signed32CountProcessed;
      signed32ReqListLength;
      struct {
            signed32ReqReplyType;
            union {
                  signed32          Reserved;
                  srcsinkdesc_t     *ListenList;
                  devdesc_attr_t    DeviceAttr;
                  char              VolumeID[HPSS_PV_NAME_SIZE];
                  gapinfo_t         GapInfo;
            } ReqReply_u;
      } ReqReply_s;
} reqspecreply_t;
```

Flags

This field is a bit vector which describes how the rest of the structure is to be interpreted. Valid values to be used in the vector are:

IOR_POSITIONVALID	If set, the Position field in the device attribute structure contains a valid entry. If clear, the contents of the Position field is undefined.
IOR_ENDPOSITION	If set and IOR_POSITIONVALID is also set, the Position field describes the position of the device at the end of the request. If clear and IOR_POSITIONVALID is set, the

	Position field in the device attributes structure describes the position of the device at the beginning of the request.
IOR_ERROR	If set, an error was detected during the processing of the request. The Status field will contain a value that describes the error.
IOR_GAPINFO_VALID	The request specific reply contains gap (file hole) information.
IOR_FOREIGN_LABEL	The media contains a valid ANSI label that was not written by HPSS. This value is for read label requests only.
IOR_NO_LABEL	The media does not contain an 80-byte label. This value is for read label requests only.
IOR_NON_ANSI_LABEL	The media contain an 80-byte label, but does not meet ANSI specifications. This value is for read label requests only.

Status

This field contains the device specific status of the request. A value of 0 (zero) indicates that the request was processed without error (or completed without error). Any other value will describe an error condition that occurred during processing of the request.

CountProcessed

This field contains the number of iterations of the requested operation that have been processed.

ReqListLength

This field contains the length of a list in the ReqReply_u union. This is used for describing the length of the ListenList field.

ReqReply_s

This field contains a typed union used to return request specific status information. This structure is generated by the DCE IDL compiler.

RepReplyType

This field contains indication of the type of information included in the following union. Valid values include:

REPLY_NONE	No further information is returned.
REPLY_LISTENLIST	Listen address information is returned. This will be returned when IOD_REPLYWHENREADY is specified in the requesting IOD.
REPLY_DEVICEATTR	Device attributes are returned.

REPLY_VOLUMEID	Volume label information is returned.
REPLY_GAPINFO	Information about an unwritten hole in a file is returned for a read request.

ReqReply_u

This union contains device specific reply information. The union elements include:

Reserved

This field is a place holder used when no request specific status information is returned.

ListenList

This list of structures contains the listen addresses which the responder has established and to which part of the request each address applies.

DeviceAttr

This structure contains device attribute values. See the Mover design document for a description of this structure.

VolumeID

This structure contains the volume label of the media.

GapInfo

This structure contains information describing an unwritten hole in a file that is subsequently read. This information is returned instead of sending NULL bytes to the requestor. The structure has the following format:

```
typedef struct gapinfo {
   u_signed64  Offset;
   u_signed64  Length;
} gapinfo_t;
```

Offset

This structure contains the offset within the transfer at which the hole starts.

Length

This structure contains the length of the hole.

SrcReplyLength

This field contains the number of replies in SrcReplyList.

SinkReplyLength

This field contains the number of replies in SinkReplyList.

SrcReplyList

This list of structures describes the state of each source of a data transfer. See the description of the source/sink reply, below. There will be one source reply list entry for each source request list entry that was received in the request IOD.

SinkReplyList

This list of structures describes the state of each sink of a data transfer. See the description of the source/sink reply, below. There will be one sink reply list entry for each sink request list entry that was received in the request IOD.

3.5.5 Source/Sink Reply - srcsinkreply_t

Description

The source/sink reply contains information describing how much of each source and sink request list entry has completed and any error that was encountered during the processing of the request.

Format

The Source/Sink Reply has the following format:

```
typedef struct srcsinkreply {
     signed32                Flags;
     signed32                Status;
     u_signed64              BytesMoved;
     positiondesc_t          Position;
     struct srcsinkreply_t   *Next;
} srcsinkreply_t;
```

Flags

This field is a bit vector that describes how the state of the request list entry, as well as how the rest of the structure is to be interpreted. Valid values for the vector are:

IOR_COMPLETE	If set, the processing for this request list entry is complete. If clear, this request list entry is still in progress.
IOR_ERROR	If set, an error was encountered processing this process request list entry and the Status field contains a value that describes that error. If clear, no error has been encountered while processing this request list entry.
POSITIONVALID	If set, the Position field contains valid device positioning information. If clear, the contents of the Position field is undefined.
ENDPOSITION	If set, the contents of the Position field describe the position of the device at the end of the request. If clear, the contents of the Position field describe the position of the device at the start of the request.

Status

This field contains a value that describes the current state of the request list entry. A value of 0 (zero) indicates that no errors have been encountered during the processing of this request. Any other value describes an error encounter during processing.

BytesMoved

This field contains the number of bytes of data for this request list entry that have been successfully transferred.

Position

This field contains device positioning information. This information can be later used to position the device to access the data. See the Mover design document for further details.

Next

This field contains a pointer to the next entry in the source or sink reply list. This field is used by the DCE encoding routines.

Chapter 4 Supplemental Data Transfer Functions

This chapter specifies support APIs to facilitate data transfers. Applications which use **hpss_ReadList** or **hpss_WriteList** are potential users of these functions. Specifically, the following information is provided:

- Application Programming Interface (API)
- Data Definitions
- Configuration and Setup

4.1 API Functions

This section describes all APIs which are provided for use by another HPSS a client external to HPSS. The API interface specification includes the following information:

Name

Synopsis

Description

Parameters

Return values

Error conditions

See also

Notes

4.1.1 Mover Socket (Parallel TCP/IP Data Transfer) Functions

4.1.1.1 mover_socket_send_buffer

Purpose

Send a data buffer using the parallel data transfer protocol.

Synopsis

```
#include "pdata.h"

int
mover_socket_send_buffer(
        int          SD,          /* IN */
        u_signed64   XferID,      /* IN */
        u_signed64   Offset,      /* IN */
        char         *Buffer,     /* IN */
        int          Length,      /* IN */
        char         *Ticket);    * IN */
```

Description

The **mover_socket_send_buffer** routine builds a parallel data header that identifies the data, and sends the header followed by the data in the specified buffer over the specified connection.

Parameters

SD	File descriptor that refers to the TCP/IP connection on which the data is to be sent.
XferID	The transfer identifier to send in the parallel data transfer header.
Offset	The transfer offset at which this data begins.
Buffer	Pointer to the data buffer to be sent.
Length	The amount of data to be sent.
Ticket	Pointer to a security ticket to be sent in the parallel data transfer header.

Return values

If the data is successfully sent, the number of bytes sent is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EINVAL	An input parameter is invalid.
EIO	An unexpected error occurred.
EPIPE	The connection closed while the header or data was sent.

See also

pdata_send_hdr_and_data, mover_socket_send_buffer_timeout, mover_socket_send_buffer_time_size.

Notes

The security ticket is currently unused by the HPSS Mover and clients.

4.1.1.2 *mover_socket_send_buffer_timeout*

Purpose

Send a data buffer using the parallel data transfer protocol, specifying a maximum number of seconds to wait if the connection becomes idle.

Synopsis

```
#include "pdata.h"

int
mover_socket_send_buffer_timeout(
        int          SD,            /* IN */
        u_signed64   XferID,        /* IN */
        u_signed64   Offset,        /* IN */
        char         *Buffer,       /* IN */
        int          Length,        /* IN */
        char         *Ticket,       /* IN */
        int          SecTimeout);   /* IN */
```

Description

The **mover_socket_send_buffer_timeout** routine builds a parallel data header that identifies the data, and sends the header followed by the data in the specified buffer over the specified connection. If the number of seconds specified by *SecTimout* elapses between messages from the peer entity, the routine will return an error indication.

Parameters

SD	File descriptor that refers to the TCP/IP connection on which the data is to be sent.
XferID	The transfer identifier to send in the parallel data transfer header.
Offset	The transfer offset at which this data begins.
Buffer	Pointer to the data buffer to be sent.
Length	The amount of data to be sent.
Ticket	Pointer to a security ticket to be sent in the parallel data transfer header.
SecTimeout	Number of seconds to wait if the connection becomes idle before returning; zero specifies infinite wait.

Return values

If the data is successfully sent, the number of bytes sent is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EINVAL	An input parameter is invalid.
EIO	An unexpected error occurred.
EPIPE	The connection closed while the header or data was sent.

ETIMEDOUT	The specified timeout expired between communication with the peer.

See also

pdata_send_hdr_and_data_timeout_size, mover_socket_send_buffer, mover_socket_send_buffer_timeout_size.

Notes

The security ticket is currently unused by the HPSS Mover and clients.

4.1.1.3 mover_socket_send_buffer_timeout_size

Purpose

Send a data buffer using the parallel data transfer protocol, specifying a maximum number of seconds to wait if the connection becomes idle and the size to be used for the individual write request to the network.

Synopsis

```
#include "pdata.h"

int
mover_socket_send_buffer_timeout_size(
        int          SD,            /* IN */
        u_signed64   XferID,        /* IN */
        u_signed64   Offset,        /* IN */
        char         *Buffer,       /* IN */
        int          Length,        /* IN */
        char         *Ticket,       /* IN */
        int          SecTimeout,    /* IN */
        int          WriteSize);    /* IN */
```

Description

The **mover_socket_send_buffer_timeout_size** routine builds a parallel data header that identifies the data, and sends the header followed by the data in the specified buffer over the specified connection. If the number of seconds specified by *SecTimout* elapses between messages from the peer entity, the routine will return an error indication. The value specified by *WriteSize* is used to indicate how much data should be written to the network with each low-level write request (i.e., write() system call to the socket specified by *SD*).

Parameters

SD	File descriptor that refers to the TCP/IP connection on which the data is to be sent.
XferID	The transfer identifier to send in the parallel data transfer header.
Offset	The transfer offset at which this data begins.
Buffer	Pointer to the data buffer to be sent.
Length	The amount of data to be sent.

Ticket	Pointer to a security ticket to be sent in the parallel data transfer header.
SecTimeout	Number of seconds to wait if the connection becomes idle before returning; zero specifies infinite wait.
WriteSize	Maximum number of bytes to be specified with each network write request.

Return values

If the data is successfully sent, the number of bytes sent is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EINVAL	An input parameter is invalid.
EIO	An unexpected error occurred.
EPIPE	The connection closed while the header or data was sent.
ETIMEDOUT	The specified timeout expired between communication with the peer.

See also

pdata_send_hdr_and_data_timeout_size, mover_socket_send_buffer, mover_socket_send_buffer_timeout.

Notes

The security ticket is currently unused by the HPSS Mover and clients.

4.1.1.4 mover_socket_get_buffer

Purpose

Request data using the parallel data transfer protocol.

Synopsis

```
#include "pdata.h"

int
mover_socket_get_buffer(
        int             SD,             /* IN */
        u_signed64      XferID,         /* IN */
        u_signed64      Offset,         /* IN */
        char            *Buffer,        /* OUT */
        int             Length,         /* IN */
        char            *Ticket);       /* IN */
```

Description

The **mover_socket_get_buffer** routine builds a parallel data header that identifies the data, sends the header over the specified connection, and then attempts to receive a parallel data transfer header and data over that same connection.

Parameters

SD	File descriptor that refers to the TCP/IP connection on which the data is to be requested and received.
XferID	The transfer identifier to send in the parallel data transfer header.
Offset	The transfer offset at which this data begins.
Buffer	Pointer to the data buffer in which the received data will be stored.
Length	The amount of data to be received.
Ticket	Pointer to a security ticket to be sent in the parallel data transfer header.

Return Values

If the data is successfully received, the number of bytes received is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error Conditions

EINVAL	An input parameter is invalid, or data received over the connection is invalid.
EIO	An unexpected error occurred.
EPIPE	The connection closed while the header was being sent.

See also

pdata_send_hdr, **pdata_recv_hdr, mover_socket_get_buffer_timeout**.

Notes

The security ticket is currently unused by the HPSS Mover and clients.

4.1.1.5 mover_socket_get_buffer_timeout

Purpose

Request data using the parallel data transfer protocol, specifying a maximum number of seconds to wait if the connection becomes idle.

Synopsis

```
#include "pdata.h"

int
mover_socket_get_buffer_timeout(
        int          SD,          /* IN */
        u_signed64   XferID,      /* IN */
        u_signed64   Offset,      /* IN */
        char         *Buffer,     /* OUT */
        int          Length,      /* IN */
        char         *Ticket,     /* IN */
```

int *SecTimeout);* /* IN */

Description

The **mover_socket_get_buffer_timeout** routine builds a parallel data header that identifies the data, sends the header over the specified connection, and then attempts to receive a parallel data transfer header and data over that same connection. If the number of seconds specified by *SecTimout* elapses between messages from the peer entity, the routine will return an error indication.

Parameters

SD	File descriptor that refers to the TCP/IP connection on which the data is to be requested and received.
XferID	The transfer identifier to send in the parallel data transfer header.
Offset	The transfer offset at which this data begins.
Buffer	Pointer to the data buffer in which the received data will be stored.
Length	The amount of data to be received.
Ticket	Pointer to a security ticket to be sent in the parallel data transfer header.
SecTimeout	Number of seconds to wait if the connection becomes idle before returning; zero specifies infinite wait.

Return Values

If the data is successfully received, the number of bytes received is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error Conditions

EINVAL	An input parameter is invalid, or data received over the connection is invalid.
EIO	An unexpected error occurred.
EPIPE	The connection closed while the header was being sent.
ETIMEDOUT	The specified timeout expired between communication with the peer.

See also

pdata_send_hdr_timeout, **pdata_recv_hdr_timeout, mover_set_get_buffer**.

Notes

The security ticket is currently unused by the HPSS Mover and clients.

4.1.1.6 mover_socket_recv_data

Purpose

Wait for a parallel data transfer header, and receive the specified data.

Synopsis

```
#include "pdata.h"
int
mover_socket_recv_data(
        int          SD,              /* IN */
        u_signed64   XferID,          /* IN */
        u_signed64   Offset,          /* IN */
        char         *Buffer,         /* OUT */
        int          BufSize,         /* IN */
        int          *BytesRecvd,     /* OUT */
        int          NumOfPackets);   /* IN */
```

Description

The **mover_socket_recv_data** routine waits for an incoming parallel data transfer header over the specified connection and if that header matches the expected transfer identifier and offset, receives data over the same connection.

Parameters

SD	File descriptor that refers to the TCP/IP connection on which the header and data is to be received.
XferID	The transfer identifier to expect in the parallel data transfer header.
Offset	The transfer offset at which this data begins.
Buffer	Pointer to the data buffer in which the received data will be stored.
BufSize	The size of the passed buffer.
BytesRecvd	Place to return the number of bytes of data actually received.
NumOfPackets	Number of incoming parallel data transfer headers to process.

Return values

If the data is successfully received, the number of bytes received is returned. If the connection is closed while receiving the header or data, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EINVAL	An input parameter is invalid, or data received over the connection is invalid.
EIO	An unexpected error occurred.

See also

pdata_recv_hdr, **mover_waitfor_data, mover_socket_recv_data_timeout**.

Notes

None.

4.1.1.7 mover_socket_recv_data_timeout

Purpose

Wait for a parallel data transfer header, and receive the specified data, specifying a maximum number of seconds to wait if the connection becomes idle.

Synopsis

```
#include "pdata.h"
int
mover_socket_recv_data_timeout(
        int          SD,            /* IN */
        u_signed64   XferID,        /* IN */
        u_signed64   Offset,        /* IN */
        char         *Buffer,       /* OUT */
        int          BufSize,       /* IN */
        int          *BytesRecvd,   /* OUT */
        int          NumOfPackets,  /* IN */
        int          SecTimeout);   /* IN */
```

Description

The **mover_socket_recv_data_timeout** routine waits for an incoming parallel data transfer header over the specified connection and if that header matches the expected transfer identifier and offset, receives data over the same connection. If the number of seconds specified by *SecTimout* elapses between messages from the peer entity, the routine will return an error indication.

Parameters

SD	File descriptor that refers to the TCP/IP connection on which the header and data is to be received.
XferID	The transfer identifier to expect in the parallel data transfer header.
Offset	The transfer offset at which this data begins.
Buffer	Pointer to the data buffer in which the received data will be stored.
BufSize	The size of the passed buffer.
BytesRecvd	Place to return the number of bytes of data actually received.
NumOfPackets	Number of incoming parallel data transfer headers to process.
SecTimeout	Number of seconds to wait if the connection becomes idle before returning; zero specifies infinite wait.

Return values

If the data is successfully received, the number of bytes received is returned. If the connection is closed while receiving the header or data, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EINVAL	An input parameter is invalid, or data received over the

	connection is invalid.
EIO	An unexpected error occurred.
ETIMEDOUT	The specified timeout expired between communication with the peer.

See also

pdata_recv_hdr_timeout, **mover_waitfor_data_timeout, mover_socket_recv_data**.

Notes

None.

4.1.1.8 mover_socket_send_requested_data

Purpose

Wait for a parallel data transfer header, and send the requested data.

Synopsis

```
#include "pdata.h"

int
mover_socket_send_requested_data(
        int          SD,              /* IN */
        u_signed64   XferID,          /* IN */
        u_signed64   Offset,          /* IN */
        char         Buffer,          /* IN */
        int          BufSize,         /* IN */
        int          *BytesSent,      /* OUT */
        int          NumOfPackets);   /* IN */
```

Description

The **mover_socket_send_requested_data** routine waits for an incoming parallel data transfer header over the specified connection and if that header matches the expected transfer identifier and offset, sends a parallel data transfer header and the data over the same connection.

Parameters

SD	File descriptor that refers to the TCP/IP connection on which the header is to be received, and the data sent.
XferID	The transfer identifier to expect in the parallel data transfer header.
Offset	The transfer offset at which this data begins.
Buffer	Pointer to the data buffer in which the data to be sent is located.
BufLength	The size of the passed buffer.
BytesRecvd	Place to return the number of bytes of data actually sent.

NumOfPackets	Number of incoming parallel data transfer headers to process.

Return values

If the data is successfully sent, the number of bytes sent is returned. If the connection is closed while receiving the header, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EINVAL	An input parameter is invalid, or the header received over the connection is invalid.
EIO	An unexpected error occurred.
EPIPE	The connection closed while the header was being sent.

See also

pdata_recv_hdr, **pdata_send_hdr_and_data**, **mover_waitfor_requests**.

Notes

None.

4.1.1.9 mover_socket_send_requested_data_timeout

Purpose

Wait for a parallel data transfer header, and send the requested data, specifying a maximum number of seconds to wait if the connection becomes idle.

Synopsis

```
#include "pdata.h"

int
mover_socket_send_requested_data_timeout(
        int          SD,              /* IN */
        u_signed64   XferID,          /* IN */
        u_signed64   Offset,          /* IN */
        char         Buffer,          /* IN */
        int          BufSize,         /* IN */
        int          *BytesSent,      /* OUT */
        int          NumOfPackets,    /* IN */
        int          SecTimeout);     /* IN */
```

Description

The **mover_socket_send_requested_data_timeout** routine waits for an incoming parallel data transfer header over the specified connection and if that header matches the expected transfer identifier and offset, sends a parallel data transfer header and the data over the same connection. If the number of seconds specified by *SecTimout* elapses between messages from the peer entity, the routine will return an error indication.

Parameters

SD	File descriptor that refers to the TCP/IP connection on which the header is to be received, and the data sent.

XferID	The transfer identifier to expect in the parallel data transfer header.
Offset	The transfer offset at which this data begins.
Buffer	Pointer to the data buffer in which the data to be sent is located.
BufLength	The size of the passed buffer.
BytesRecvd	Place to return the number of bytes of data actually sent.
NumOfPackets	Number of incoming parallel data transfer headers to process.
SecTimeout	Number of seconds to wait if the connection becomes idle before returning; zero specifies infinite wait.

Return values

If the data is successfully sent, the number of bytes sent is returned. If the connection is closed while receiving the header, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EINVAL	An input parameter is invalid, or the header received over the connection is invalid.
EIO	An unexpected error occurred.
EPIPE	The connection closed while the header was being sent.
ETIMEDOUT	The specified timeout expired between communication with the peer.

See also

pdata_recv_hdr_timeout, **pdata_send_hdr_and_data_timeout**, **mover_waitfor_requests_timeout, mover_send_requested_data**.

Notes

None.

4.1.1.10 mover_socket_send_requested_data_timeout_size

Purpose

Wait for a parallel data transfer header, and send the requested data, specifying a maximum number of seconds to wait if the connection becomes idle and the size to be used for the individual write request to the network.

Synopsis

```
#include "pdata.h"

int
```

```
mover_socket_send_requested_data_timeout_size(
        int             SD,             /* IN */
        u_signed64      XferID,         /* IN */
        u_signed64      Offset,         /* IN */
        char            Buffer,         /* IN */
        int             BufSize,        /* IN */
        int             *BytesSent,     /* OUT */
        int             NumOfPackets,   /* IN */
        int             SecTimeout,     /* IN */
        int             WriteSize);     /* IN */
```

Description

The **mover_socket_send_requested_data_timeout_size** routine waits for an incoming parallel data transfer header over the specified connection and if that header matches the expected transfer identifier and offset, sends a parallel data transfer header and the data over the same connection. If the number of seconds specified by *SecTimout* elapses between messages from the peer entity, the routine will return an error indication.The value specified by *WriteSize* is used to indicate how much data should be written to the network with each low-level write request (i.e., write() system call to the socket specified by *SD*).

Parameters

SD	File descriptor that refers to the TCP/IP connection on which the header is to be received, and the data sent.
XferID	The transfer identifier to expect in the parallel data transfer header.
Offset	The transfer offset at which this data begins.
Buffer	Pointer to the data buffer in which the data to be sent is located.
BufLength	The size of the passed buffer.
BytesRecvd	Place to return the number of bytes of data actually sent.
NumOfPackets	Number of incoming parallel data transfer headers to process.
SecTimeout	Number of seconds to wait if the connection becomes idle before returning; zero specifies infinite wait.
WriteSize	Maximum number of bytes to be specified with each network write request.

Return values

If the data is successfully sent, the number of bytes sent is returned. If the connection is closed while receiving the header, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EINVAL	An input parameter is invalid, or the header received over the connection is invalid.

EIO	An unexpected error occurred.
EPIPE	The connection closed while the header was being sent.
ETIMEDOUT	The specified timeout expired between communication with the peer.

See also

pdata_recv_hdr_timeout, **pdata_send_hdr_and_data_timeout_size**, **mover_waitfor_requests_timeout, mover_send_requested_data_size**.

Notes

None.

4.1.1.11 mover_waitfor_data

Purpose

Wait for connections on a listen socket, then receive data over those connections using the parallel data transfer protocol.

Synopsis

```
#include "pdata.h"

int
mover_waitfor_data(
        int          SD,            /* IN */
        u_signed64   XferID,        /* IN */
        u_signed64   Offset,        /* IN */
        char         *Buffer,       /* OUT */
        int          BufSize,       /* IN */
        int          *BytesMoved);  /* OUT */
```

Description

The **mover_waitfor_data** routine waits for connections on a listen socket, then processes those connections - waiting on incoming parallel data transfer headers and then receiving the specified data. This routine handles multiple connections and multiple requests per connection. All opened connections are closed before this routine returns.

Parameters

SD	File descriptor that refers to the listen socket.
XferID	The transfer identifier to expect in the parallel data transfer headers.
Offset	The transfer offset at which this data begins.
Buffer	Pointer to the data buffer in which the data will be stored.
BufSize	The size of the passed buffer.
BytesMoved	Place to return the number of bytes of data actually received.

Return values

If the data is successfully received, the number of bytes received is returned. If the connection is closed while receiving the headers or data, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EBUSY	The number of connections opened exceeded the maximum supported.
EINVAL	An input parameter is invalid, or a header received over the connection is invalid.
EIO	An unexpected error occurred.

See also

pdata_recv_hdr, **mover_waitfor_data_timeout**.

Notes

This routine currently supports 32 simultaneous open connections.

4.1.1.12 mover_waitfor_data_timeout

Purpose

Wait for connections on a listen socket, then receive data over those connections using the parallel data transfer protocol, specifying a maximum number of seconds to wait if the connection becomes idle.

Synopsis

```
#include "pdata.h"

int
mover_waitfor_data_timeout(
        int             SD,             /* IN */
        u_signed64      XferID,         /* IN */
        u_signed64      Offset,         /* IN */
        char            *Buffer,        /* OUT */
        int             BufSize,        /* IN */
        int             *BytesMoved,    /* OUT */
        int             SecTimeout);    /* IN */
```

Description

The **mover_socket_data_timeout** routine waits for connections on a listen socket, then processes those connections - waiting on incoming parallel data transfer headers and then receiving the specified data.If the number of seconds specified by *SecTimout* elapses between messages from the peer entity, the routine will return an error indication. This routine handles multiple connections and multiple requests per connection. All opened connections are closed before this routine returns.

Parameters

SD	File descriptor that refers to the listen socket.
XferID	The transfer identifier to expect in the parallel data transfer headers.

Offset	The transfer offset at which this data begins.
Buffer	Pointer to the data buffer in which the data will be stored.
BufSize	The size of the passed buffer.
BytesMoved	Place to return the number of bytes of data actually received.
SecTimeout	Number of seconds to wait if the connection becomes idle before returning; zero specifies infinite wait.

Return values

If the data is successfully received, the number of bytes received is returned. If the connection is closed while receiving the headers or data, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EBUSY	The number of connections opened exceeded the maximum supported.
EINVAL	An input parameter is invalid, or a header received over the connection is invalid.
EIO	An unexpected error occurred.
ETIMEDOUT	The specified timeout expired between communication with the peer.

See also

pdata_recv_hdr_timeout, **mover_waitfor_data**.

Notes

This routine currently supports 32 simultaneous open connections.

4.1.1.13 mover_waitfor_requests

Purpose

Wait for connections on a listen socket, then send data over those connections using the parallel data transfer protocol.

Synopsis

```
#include "pdata.h"

int
mover_waitfor_requests(
        int             SD,             /* IN */
        u_signed64      XferID,         /* IN */
        u_signed64      Offset,         /* IN */
        char            *Buffer,        /* IN */
        int             BufSize,        /* IN */
        int             *BytesMoved);   /* OUT */
```

Description

The **mover_waitfor_requests** routine waits for connections on a listen socket, then processes those connections - waiting on incoming parallel data transfer headers and then sending the specified data. This routine handles multiple connections and multiple requests per connection. All opened connections are closed before this routine returns.

Parameters

SD	File descriptor that refers to the listen socket.
XferID	The transfer identifier to expect in the parallel data transfer headers.
Offset	The transfer offset at which this data begins.
Buffer	Pointer to the data buffer in which the data to be sent is located.
BufSize	The size of the passed buffer.
BytesMoved	Place to return the number of bytes of data actually sent.

Return values

If the data is successfully sent, the number of bytes sent is returned. If the connection is closed while receiving the headers, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EBUSY	The number of connections opened exceeded the maximum supported.
EINVAL	An input parameter is invalid, or a header received over the connection is invalid.
EIO	An unexpected error occurred.
EPIPE	The connection closed while the header was being sent.

See also

pdata_recv_hdr, **pdata_send_hdr_and_data**, **mover_waitfor_requests_timeout**.

Notes

This routine currently supports 32 simultaneous open connections.

4.1.1.14 mover_waitfor_requests_timeout

Purpose

Wait for connections on a listen socket, then send data over those connections using the parallel data transfer protocol, specifying a maximum number of seconds to wait if the connection becomes idle.

Synopsis

#include "pdata.h"

```
int
mover_waitfor_requests_timeout(
        int             SD,             /* IN */
        u_signed64      XferID,         /* IN */
        u_signed64      Offset,         /* IN */
        char            *Buffer,        /* IN */
        int             BufSize,        /* IN */
        int             *BytesMoved,    /* OUT */
        int             SecTimeout);    /* IN */
```

Description

The **mover_waitfor_requests_timeout** routine waits for connections on a listen socket, then processes those connections - waiting on incoming parallel data transfer headers and then sending the specified data. If the number of seconds specified by *SecTimout* elapses between messages from the peer entity, the routine will return an error indication. This routine handles multiple connections and multiple requests per connection. All opened connections are closed before this routine returns.

Parameters

SD	File descriptor that refers to the listen socket.
XferID	The transfer identifier to expect in the parallel data transfer headers.
Offset	The transfer offset at which this data begins.
Buffer	Pointer to the data buffer in which the data to be sent is located.
BufSize	The size of the passed buffer.
BytesMoved	Place to return the number of bytes of data actually sent.
SecTimeout	Number of seconds to wait if the connection becomes idle before returning; zero specifies infinite wait.

Return values

If the data is successfully sent, the number of bytes sent is returned. If the connection is closed while receiving the headers, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EBUSY	The number of connections opened exceeded the maximum supported.
EINVAL	An input parameter is invalid, or a header received over the connection is invalid.
EIO	An unexpected error occurred.
EPIPE	The connection closed while the header was being sent.
ETIMEDOUT	The specified timeout expired between communication with

the peer.

See also

pdata_recv_hdr_timeout, **pdata_send_hdr_and_data_timeout**, **mover_waitfor_requests**.

Notes

This routine currently supports 32 simultaneous open connections.

4.1.2 Mover Protocol APIs

4.1.2.1 mvrprot_recv_initmsg

Purpose

Receive a mover protocol initiator message.

Synopsis

```
#include "mvr_protocol.h"

long
mvrprot_recv_initmsg(
        int                 SockFD,         /* IN */
        initiator_msg_t*    InitPtr);       /* OUT */
```

Description

The **mvrprot_recv_initmsg** routine receives a mover protocol initiator message over the specified socket.

Parameters

SockFD	Descriptor that refers to the open connection over which the message is to be received.
InitPtr	Pointer to the area in which the received message will be stored.

Return Values

If the message is successfully received, HPSS_E_NOERROR is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

HPSS_ECONN	The connection was closed while trying to receive the message.
HPSS_EINVAL	The verification of the checksum sent with the message failed.

See also

mvrprot_send_initmsg.

Notes

None.

4.1.2.2 mvrprot_send_initmsg

Purpose

Send a mover protocol initiator message.

Synopsis

```
#include "mvr_protocol.h"

long
mvrprot_send_initmsg(
        int                 SockFD,         /* IN */
        initiator_msg_t     *InitPtr);      /* IN */
```

Description

The **mvrprot_recv_initmsg** routine sends a mover protocol initiator message over the specified socket.

Parameters

SockFD	Descriptor that refers to the open connection over which the message is to be sent.
InitPtr	Pointer to the message to be sent.

Return values

If the message is successfully sent, HPSS_E_NOERROR is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

HPSS_ECONN	The connection was closed while trying to send the message.

See also

mvrprot_recv_initmsg.

Notes

None.

4.1.2.3 mvrprot_recv_ipaddr

Purpose

Receive a mover protocol initiator TCP/IP address.

Synopsis

```
#include "mvr_protocol.h"

long
mvrprot_recv_ipaddr(
        int                 SockFD,         /* IN */
        initiator_ipaddr_t* InitIpPtr);     /* OUT */
```

Description

The **mvrprot_recv_ipaddr** routine receives a mover protocol TCP/IP address over the specified socket.

Parameters

SockFD	Descriptor that refers to the open connection over which the address is to be received.
InitIpPtr	Pointer to the area in which the received address will be stored.

Return values

If the address is successfully received, HPSS_E_NOERROR is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

HPSS_ECONN	The connection was closed while trying to receive the address.
HPSS_EINVAL	The verification of the checksum sent with the address failed.

See also

mvrprot_send_ipaddr.

Notes

None.

4.1.2.4 mvrprot_send_ipaddr

Purpose

Send a mover protocol initiator TCP/IP address.

Synopsis

```
#include "mvr_protocol.h"

long
mvrprot_send_ipaddr(
        int                 SockFD,          /* IN */
        initiator_ipaddr_t  *InitIpPtr);     /* IN */
```

Description

The **mvrprot_send_ipaddr** routine sends a mover protocol TCP/IP address over the specified socket.

Parameters

SockFD	Descriptor that refers to the open connection over which the address is to be sent.
InitIpPtr	Pointer to the address to be sent.

Return values

If the address is successfully sent, HPSS_E_NOERROR is sent. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

HPSS_ECONN	The connection was closed while trying to send the address.

See also

mvrprot_recv_ipaddr.

Notes

None.

4.1.2.5 mvrprot_recv_shmaddr

Purpose

Receive a mover protocol initiator shared memory address.

Synopsis

```
#include "mvr_protocol.h"

long
mvrprot_recv_shmaddr(
        int                     SockFD,         /* IN */
        initiator_shmaddr_t     *InitShmPtr);   /* OUT */
```

Description

The **mvrprot_recv_shmaddr** routine receives a mover protocol shared memory address over the specified socket.

Parameters

SockFD	Descriptor that refers to the open connection over which the address is to be received.
InitShmPtr	Pointer to the area in which the received address will be stored.

Return values

If the address is successfully received, HPSS_E_NOERROR is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

HPSS_ECONN	The connection was closed while trying to receive the address.
HPSS_EINVAL	The verification of the checksum sent with the address failed.

See also

mvrprot_send_shmaddr.

Notes

None.

4.1.2.6 mvrprot_send_shmaddr

Purpose

Send a mover protocol initiator shared memory address.

Synopsis

```
#include "mvr_protocol.h"

long
mvrprot_send_shmaddr(
        int                     SockFD,         /* IN */
        initiator_shmaddr_t*    InitShmPtr);    /* IN */
```

Description

The **mvrprot_send_shmaddr** routine sends a mover protocol shared memory address over the specified socket.

Parameters

SockFD	Descriptor that refers to the open connection over which the address is to be sent.
InitShmPtr	Pointer to the address to be sent.

Return values

If the address is successfully sent, HPSS_E_NOERROR is sent. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

HPSS_ECONN	The connection was closed while trying to send the address.

See also

mvrprot_recv_shmaddr.

Notes

None.

4.1.2.7 mvrprot_recv_compmsg

Purpose

Receive a mover protocol completion message.

Synopsis

```
#include "mvr_protocol.h"

long
mvrprot_recv_compmsg(
        int                     SockFD,         /* IN */
        completion_msg_t        *CompPtr);      /* OUT */
```

Description

The **mvrprot_recv_compmsg** routine receives a mover protocol completion message over the specified socket.

Parameters

SockFD	Descriptor that refers to the open connection over which the message is to be received.

CompPtr	Pointer to the area in which the received message will be stored.

Return values

If the message is successfully received, HPSS_E_NOERROR is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

HPSS_ECONN	The connection was closed while trying to receive the message.
HPSS_EINVAL	The verification of the checksum sent with the message failed.

See also

mvrprot_send_compmsg.

Notes

None.

4.1.2.8 mvrprot_send_compmsg

Purpose

Send a mover protocol completion message.

Synopsis

```
#include "mvr_protocol.h"

long
mvrprot_send_compmsg(
        int                 SockFD,         /* IN */
        completion_msg_t*   CompPtr);       /* IN */
```

Description

The **mvrprot_recv_compmsg** routine sends a mover protocol completion message over the specified socket.

Parameters

SockFD	Descriptor that refers to the open connection over which the message is to be sent.
CompPtr	Pointer to the message to be sent.

Return values

If the message is successfully sent, HPSS_E_NOERROR is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

HPSS_ECONN	The connection was closed while trying to send the message.

See also

mvrprot_recv_compmsg.

Notes

None.

4.1.3 Mover Protocol Data Structures

4.1.3.1 Mover Protocol Initiator Message Structure - initiator_msg_t

Description

The Mover Protocol Initiator Message Structure contains information used to communicate transfer options during an HPSS data transfer.

Format

The Mover Protocol Initiator Message Structure has the following format:

```
typedef struct initiator_msg {
     u_signed64      Delimiter;
     unsigned32      Flags;
     unsigned32      Type;
     u_signed64      Offset;
     u_signed64      Length;
     u_signed64      BlockSize;
     u_signed64      StripeWidth;
     u_signed64      Stride;
     u_signed64      TotalLength;
     char            SecurityTicket[MVRPROT_SEC_TICKET_LEN];
     u_signed64      CheckSum;
} ;
```

Delimiter

Contains a distinct value to identify the message boundary.

Flags

Contains values that indicate data transfer control options.

Valid values include:

MVRPROT_RESPONDER	Indicates that the sender will be the responder for the current part of the data transfer.
MVPROT_ADDR_FOLLOWS	Indicates that an address message will be sent following the initiator message.
MVRPROT_COMP_REPLY	Indicates that the sender requests a completion message be sent upon completion of the current part of the data transfer.
MVRPROT_HOLD_RESOURCES	Indicates that the peer's connection to the passed address should remain open across requests.

Type

Indicates the transfer mechanism type. Valid values include:

NET_ADDRESS	TCP / IP transfer..
SHM_ADDRESS	Shared memory transfer.

Offset

The current offset within the data transfer.

Length

The length of the current part of the data transfer.

BlockSize

The block size used for striping at the client or device level. This field is currently unused.

StripeWidth

The width of the stripe at the client or device level. This field is currently unused.

Stride

The stride at the client or device level.

This field is currently unused.

TotalLength

The total amount of data, contiguous within the transfer, for which the sender will use the same data transfer mechanism. This value is used to perform passive-side read-ahead optimization.

CheckSum

Contains a computed checksum used to verify correct transfer of this header.

4.1.3.2 Mover Protocol Completion Msg Structure - completion_msg_t

Description
The Mover Protocol Completion Message Structure is used to communicate the completion status of part of an HPSS data transfer, and is sent from the transfer responder to the transfer initiator.

Format
The Mover Protocol Completion Message Structure has the following format:

```
typedef struct completion_msg {
     u_signed64       Delimiter;
     unsigned32       Flags;
     unsigned32       Status;
     u_signed64       BytesMoved;
     char             SecurityTicket[MVRPROT_SEC_TICKET_LEN];
     u_signed64       CheckSum;
} completion_msg_t;
```

Delimiter

Contains a distinct value to identify the message boundary.

Flags

Contains flags describing the result of the transfer. This field is currently unused.

Status

Contains the status of completed part of the transfer, as viewed by the transfer responder.

BytesMoved

Contains the number of bytes successfully moved, contiguous from the start of the current part of the transfer.

SecurityTicket

Provides space to communicate a security ticket. This field is currently unused.

CheckSum

Contains a computed checksum used to verify correct transfer of this header.

4.1.3.3 Mover Protocol TCP/IP Address Structure - initiator_ipaddr_t

Description
The Mover Protocol TCP/IP Address Structure is used to communicate TCP/IP addressing information.

Format
The Mover Protocol TCP/IP Address Structure has the following format:

```
typedef struct initiator_ipaddr {
     u_signed64       Delimiter;
     unsigned32       Flags;
     netaddress_t     IpAddr;
     char             SecurityTicket[MVRPROT_SEC_TICKET_LEN];
     u_signed64       CheckSum;
} initiator_ipaddr_t;
```

Delimiter

Contains a distinct value to identify the message boundary.

Flags

Contains flags specific to the TCP/IP address. This field is currently unused.

IpAddr

Contains the TCP/IP address in network byte order. See the format of the Input / Output Descriptor (IOD) for further details.

SecurityTicket

Provides space to communicate a security ticket. This field is currently unused.

CheckSum

Contains a computed checksum used to verify correct transfer of this header.

4.1.3.4 Mover Protocol Shm Address Structure - initiator_shmaddr_t

Description
The Mover Protocol Shared Memory Address Structure is used to communicate shared memory addressing information.

Format
The Mover Protocol Shared Memory Address Structure has the following format:

```
typedef struct initiator_shmaddr {
     u_signed64       Delimiter;
     unsigned32       Flags;
     shmaddress_t     ShmAddr;
     char             SecurityTicket[MVRPROT_SEC_TICKET_LEN];
     u_signed64       CheckSum;
} initiator_shmaddr_t;
```

Delimiter

Contains a distinct value to identify the message boundary.

Flags

Contains flags specific to the shared memory address. This field is currently unused.

ShmAddr

Contains the shared memory address. See the format of the Input / Output Descriptor (IOD) for further details.

SecurityTicket

Provides space to communicate a security ticket. This field is currently unused.

CheckSum

Contains a computed checksum used to verify correct transfer of this header.

4.1.4 Parallel Data Transfer Functions

4.1.4.1 pdata_recv_hdr

Purpose

Receive a parallel data transfer header.

Synopsis

```
#include "pdata.h"

int
pdata_recv_hdr(
        int             SocketDescriptor,       /* IN */
        pdata_hdr_t     *PdataHeaderPtr);       /* OUT */
```

Description

The **pdata_recv_hdr** function receives a parallel data transfer header on the connection referenced by *SocketDescriptor*, and places the received header into the structure pointed to by *PdataHeaderPtr*.

Parameters

SocketDescriptor	File descriptor referring to the open TCP/IP connection over which the header is to be received.
PdataHeaderPtr	Pointer to the area where the incoming header will be stored.

Return Values

If a parallel data transfer header is successfully received, the size of the header in bytes is returned. If the connection is closed while waiting for the incoming header, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error Conditions

EBADF	*SocketDescriptor* does not refer to an open connection.
EINVAL	Header checksum or delimiter is invalid.

See also

mover_socket_recv_data, **mover_waitfor_data, pdata_recv_hdr_timeout**.

Notes

None.

4.1.4.2 pdata_recv_hdr_timeout

Purpose

Receive a parallel data transfer header, specifying a maximum number of seconds to wait if the connection becomes idle.

Synopsis

```
#include "pdata.h"
```

```
int
pdata_recv_hdr_timeout(
        int             SocketDescriptor,       /* IN */
        pdata_hdr_t     *PdataHeaderPtr,        /* OUT */
        int             SecTimeout);            /* IN */
```

Description

The **pdata_recv_hdr_timeout** function receives a parallel data transfer header on the connection referenced by *SocketDescriptor*, and places the received header into the structure pointed to by *PdataHeaderPtr*. If the number of seconds specified by *SecTimout* elapses between messages from the peer entity, the routine will return an error indication.

Parameters

SocketDescriptor	File descriptor referring to the open TCP/IP connection over which the header is to be received.
PdataHeaderPtr	Pointer to the area where the incoming header will be stored.
SecTimeout	Number of seconds to wait if the connection becomes idle before returning; zero specifies infinite wait.

Return Values

If a parallel data transfer header is successfully received, the size of the header in bytes is returned. If the connection is closed while waiting for the incoming header, a value of zero is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error Conditions

EBADF	*SocketDescriptor* does not refer to an open connection.
EINVAL	Header checksum or delimiter is invalid.
ETIMEDOUT	The specified timeout expired between communication with the peer.

See also

mover_socket_recv_data_timeout, **mover_waitfor_data_timeout, pdata_recv_hdr**.

Notes

None.

4.1.4.3 pdata_send_hdr

Purpose

Send a parallel data transfer header.

Synopsis

```
#include "pdata.h"

int
pdata_send_hdr(
        int             SocketDescriptor,       /* IN */
```

```
        pdata_hdr_t*    PdataHeaderPtr);        /* IN */
```

Description

The **pdata_send_hdr** function sends a paralleled data transfer header, pointed to by *PdataHeaderPtr*, on the connection referenced by *SocketDescriptor*.

Parameters

SocketDescriptor	File descriptor referring to the open TCP/IP connection over which the header is to be sent.
PdataHeaderPtr	Pointer to the header to be sent.

Return values

If the parallel data transfer header is successfully sent, the size of the header in bytes is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EBADF	*SocketDescriptor* does not refer to an open connection.
EPIPE	The connection closed while attempting to send the header.

See also

pdata_send_hdr_and_data, **mover_socket_send_buffer**.

Notes

None.

4.1.4.4 pdata_send_hdr_timeout

Purpose

Send a parallel data transfer header, specifying a maximum number of seconds to wait if the connection becomes idle.

Synopsis

```
#include "pdata.h"

int
pdata_send_hdr_timeout(
        int             SocketDescriptor,       /* IN */
        pdata_hdr_t*    PdataHeaderPtr,         /* IN */
        int             SecTimeout);            /* IN */
```

Description

The **pdata_send_hdr_timeout** function sends a paralleled data transfer header, pointed to by *PdataHeaderPtr*, on the connection referenced by *SocketDescriptor*. If the number of seconds specified by *SecTimout* elapses between messages from the peer entity, the routine will return an error indication.

Parameters

SocketDescriptor	File descriptor referring to the open TCP/IP connection over which the header is to be sent.

PdataHeaderPtr	Pointer to the header to be sent.
SecTimeout	Number of seconds to wait if the connection becomes idle before returning; zero specifies infinite wait.

Return values

If the parallel data transfer header is successfully sent, the size of the header in bytes is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EBADF	*SocketDescriptor* does not refer to an open connection.
EPIPE	The connection closed while attempting to send the header.
ETIMEDOUT	The specified timeout expired between communication with the peer.

See also

pdata_send_hdr_and_data_timeout, **mover_socket_send_buffer_timeout**.

Notes

None.

4.1.4.5 pdata_send_hdr_and_data

Purpose

Send a parallel data transfer header, followed by the data described by that header.

Synopsis

```
#include "pdata.h"

int
pdata_send_hdr_and_data(
        int             SocketDescriptor,       /* IN */
        pdata_hdr_t     *PdataHeaderPtr,        /* IN */
        char            *DataBuffer,            /* IN */
        int             DataLength);            /* IN */
```

Description

The **pdata_send_hdr_and_data** function sends a parallel data transfer header, pointed to by *PdataHeaderPtr*, and data, from the buffer referenced by *DataBuffer*, on the connection referenced by *SocketDescriptor*.

Parameters

SocketDescriptor	File descriptor referring to the open TCP/IP connection over which the header is to be sent.
PdataHeaderPtr	Pointer to the header to be sent.
DataBuffer	Pointer to the data to be sent.

DataLength	Amount of data, in bytes, to be sent (excluding the header).

Return values

If the parallel data transfer header and the data buffer are successfully sent, the amount of data sent in bytes is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EBADF	*SocketDescriptor* does not refer to an open connection.
EINVAL	The *DataLength* parameter does not match the length in the parallel data transfer header.
EPIPE	The connection closed while attempting to send the header or data.

See also

pdata_send_hdr, **mover_socket_send_buffer**.

Notes

None.

4.1.4.6 pdata_send_hdr_and_data_timeout

Purpose

Send a parallel data transfer header, followed by the data described by that header, specifying a maximum number of seconds to wait if the connection becomes idle.

Synopsis

```
#include "pdata.h"

int
pdata_send_hdr_and_data_timeout(
        int         SocketDescriptor,       /* IN */
        pdata_hdr_t *PdataHeaderPtr,        /* IN */
        char        *DataBuffer,            /* IN */
        int         DataLength,             /* IN */
        int         SecTimeout);            /* IN */
```

Description

The **pdata_send_hdr_and_data_timeout** function sends a parallel data transfer header, pointed to by *PdataHeaderPtr*, and data, from the buffer referenced by *DataBuffer*, on the connection referenced by *SocketDescriptor*. If the number of seconds specified by *SecTimout* elapses between messages from the peer entity, the routine will return an error indication.

Parameters

SocketDescriptor	File descriptor referring to the open TCP/IP connection over which the header is to be sent.
PdataHeaderPtr	Pointer to the header to be sent.
DataBuffer	Pointer to the data to be sent.

DataLength	Amount of data, in bytes, to be sent (excluding the header).
SecTimeout	Number of seconds to wait if the connection becomes idle before returning; zero specifies infinite wait.

Return values

If the parallel data transfer header and the data buffer are successfully sent, the amount of data sent in bytes is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EBADF	*SocketDescriptor* does not refer to an open connection.
EINVAL	The *DataLength* parameter does not match the length in the parallel data transfer header.
EPIPE	The connection closed while attempting to send the header or data.
ETIMEDOUT	The specified timeout expired between communication with the peer.

See also

pdata_send_hdr_timeout, **mover_socket_send_buffer_timeout**.

Notes

None.

4.1.4.7 pdata_send_hdr_and_data_timeout_size

Purpose

Send a parallel data transfer header, followed by the data described by that header, specifying a maximum number of seconds to wait if the connection becomes idle and the size to be used for the individual write request to the network.

Synopsis

```
#include "pdata.h"

int
pdata_send_hdr_and_data_timeout_size(
        int             SocketDescriptor,       /* IN */
        pdata_hdr_t     *PdataHeaderPtr,        /* IN */
        char            *DataBuffer,            /* IN */
        int             DataLength,             /* IN */
        int             SecTimeout.             /* IN */
        int             WriteSize);             /* IN */
```

Description

The **pdata_send_hdr_and_data_timeout_size** function sends a parallel data transfer header, pointed to by *PdataHeaderPtr*, and data, from the buffer referenced by *DataBuffer*, on the connection referenced by *SocketDescriptor*. If the number of seconds specified by *SecTimout* elapses between messages from the peer entity, the routine will return an error indication. The value specified by

WriteSize is used to indicate how much data should be written to the network with each low-level write request (i.e., write() system call to the socket specified by *SD*).

Parameters

SocketDescriptor	File descriptor referring to the open TCP/IP connection over which the header is to be sent.
PdataHeaderPtr	Pointer to the header to be sent.
DataBuffer	Pointer to the data to be sent.
DataLength	Amount of data, in bytes, to be sent (excluding the header).
SecTimeout	Number of seconds to wait if the connection becomes idle before returning; zero specifies infinite wait.
WriteSize	Maximum number of bytes to be specified with each network write request.

Return values

If the parallel data transfer header and the data buffer are successfully sent, the amount of data sent in bytes is returned. Otherwise, a negative value is returned, the absolute value of which is equal to an errno value defined below.

Error conditions

EBADF	*SocketDescriptor* does not refer to an open connection.
EINVAL	The *DataLength* parameter does not match the length in the parallel data transfer header.
EPIPE	The connection closed while attempting to send the header or data.
ETIMEDOUT	The specified timeout expired between communication with the peer.

See also

pdata_send_hdr_timeout, **mover_socket_send_buffer_timeout_size**.

Notes

None.

4.1.5 Parallel Data Transfer Data Definitions

4.1.5.1 Parallel Data Transfer Header - pdata_hdr_t

Description

The Parallel Data Transfer Header contains the information necessary to exactly identify a contiguous piece of a parallel data transfer. This header is used to request and/or precede data sent via TCP/IP.

Format

The Parallel Data Transfer Header has the following format:

```
typedef struct pdata_hdr{
     u_signed64       PdataDelimiter;
     u_signed64       XferID;
     u_signed64       Offset;
     u_signed64       Length;
     char             SecurityTicket[8];
     u_signed64       CheckSum;
} pdata_hdr_t;
```

PdataDelimiter

Contains a distinct value to identify the message boundary.

XferID

The transfer identifier for the current data transfer.

Offset

The offset within the data transfer at which the associated piece of data begins.

Length

The length of the associated piece of data.

SecurityTicket

Provides space to communicate a security ticket. This field is currently unused.

CheckSum

Contains a computed checksum used to verify correct transfer of this header.

4.1.6 Network Options Functions

4.1.6.1 netopt_FindEntry

Purpose

Returns an HPSS network option entry for the specified IP address.

Synopsis

```
#include "hpss_netopt.h"

long
netopt_FindEntry(
        long            NetAddr,                /* IN */
        netopt_entry_t  **RetTableEntryPtr);    /* OUT */
```

Description

The **netopt_FindEntry** function searches the HPSS network option entries for an entry that matches the IP address specified by *NetAddr*, and returns a pointer to the entry in the value pointed to by *RetTableEntryPtr*.

Parameters

NetAddr	The IP address of the machine or network for which a match is requested in the HPSS network options table.
RetTableEntryPtr	Pointer to the area where the HPSS network option entry pointer will be returned.

Return Values

If an HPSS network option entry that corresponds to the specified IP address is found, then a valu of zero is returned and a pointer to the entry is returned in the area specified by *RetTableEntryPtr*. Otherwise, a negative value is returned which describes the error, as defined below.

Error Conditions

HPSS_ENOMEM	Could not allocate memory for the network options table.
HPSS_ENOENT	The HPSS network options configuration file does not exist or no entry could be found that corresponds to the specified IP address.
HPSS_ESYSTEM	An operating system service failed.

See also

netopt_GetWriteSize.

Notes

None.

4.1.6.2 netopt_GetWriteSize

Purpose

Returns the size in bytes to be used for writes to the TCP/IP connection referred to by either a

socket descriptor or IP address.

Synopsis

```
#include "hpss_netopt.h"

int
netopt_GetWriteSize(
        int         SocketDescriptor,       /* IN */
        long        NetAddr);               /* IN */
```

Description

The **netopt_GetWriteSize** function returns the configured size, in bytes, to be used for writing data to the connection referred to by either *SocketDescriptor* or *NetAddr*. The function searches the HPSS network option entries for an entry that matches the IP address specified by *NetAddr*, if non-zero and otherwise based on the local address corresponding to the socket referred to by *SocketDescriptor* - if an entry is found and contains a non-zero network write size, that value is returned; otherwise the environment variable **HPSS_TCP_WRITESIZE** is interrogated - if it is set and contains a non-zero value, that value is returned, otherwise zero is returned.

Parameters

NetAddr	The IP address of the machine or network for which a match is requested in the HPSS network options table.
RetTableEntryPtr	Pointer to the area where the HPSS network option entry pointer will be returned.

Return Values

The size, in bytes, that is configured for the specified network address is returned, if any values has been specified. Otherwise a value of zero is returned.

Error Conditions

None.

See also

netopt_FindEntry.

Notes

None.

4.1.7 Network Options Data Definitions

4.1.7.1 Network Options Entry - netopt_entry_t

Description

The Network Options Entry structure contains the configuration information for a particular network (IP) address. This information is used to allow different networking options to be utilized for different HPSS Mover and/or HPSS client machines and networks

Format

The Network Options Entry structure has the following format:

```
typedef struct netopt_entry {
     unsigned long    IPAddr;
     unsigned long    NetMask;
     unsigned long    RFC1323;
     unsigned long    SendSpace;
     unsigned long    RecvSpace;
     unsigned long    WriteSize;
     unsigned long    TCPNodelay;
     unsigned long    Reserved2;
} netopt_entry_t;
```

IpAddr

The IP address for this entry.

NetMask

The network mask to be applied to the incoming address to determine whether this entry applies to that address.

RFC1323

Indicates whether RFC 1323 (large TCP window size support) should be enabled for this address. A value of zero indicates that RFC 1323 support should be disabled; a non-zero value indicates that RFC 1323 support should be enabled.

SendSpace

The value to be used for the socket send buffer size.

RecvSpace

The value to be used for the socket receive buffer size.

WriteSize

The maximum number of bytes that should be written to a network connection corresponding to this entry with a single I/O request.

TCPNodelay

Indicates whether the algorithm employed to try and coalesce small writes to a TCP connection should be disabled. A value of zero indicates that the algorigthm should not be disabled; a non-zero value indicates that the algorithm should be disabled.

Reserved2

This field is currently unused.

Chapter 5 Math Library

This chapter specifies the HPSS 64-bit arithmetic programming interface. Specifically, the following information is provided:

Application Programming Interfaces (APIs)

Data Definitions

5.1 API Interfaces

This section describes all API interfaces which are provided for use by another HPSS subsystem or by a client external to HPSS. The API interface specification includes the following information:

Name

Synopsis

Description

Parameters

Return values

Error conditions

See also

Notes

5.1.1 add64m

Purpose

Add two unsigned 64-bit integers.

Synopsis

```
#include "u_signed64.h"

u_signed64 add64m(
        u_signed64          ll1,          /* IN */
        u_signed64          ll2);         /* IN */
```

Description

The *add64m* macro is called to add two unsigned 64-bit integers.

Parameters

ll1	Specifies the first unsigned 64-bit integer.
ll2	Specifies the second unsigned 64-bit integer.

Return values

Upon completion, an unsigned 64-bit integer is returned.

Error conditions

None.

See also

add64_3m.

Notes

None.

5.1.2 add64_3m

Purpose

Add two unsigned 64-bit integers and store the result in a separate integer field.

Synopsis

```
#include "u_signed64.h"

void  add64m(
        u_signed64          ll1,          /* OUT */
        u_signed64          ll2,          /* IN */
        u_signed64          ll3);         /* IN */
```

Description

The *add64_3m* macro is called to add two unsigned 64-bit integers and store the result into a third unsigned 64-bit integer.

Parameters

ll1	Specifies the result of the addition of the 2 unsigned 64-bit operands.
ll2	Specifies the first unsigned 64-bit integer addition operand.
ll3	Specifies an unsigned 64-bit integer to add to the first operand.

Return values

None. The result of the addition is stored in the first parameter.

Error conditions

None.

See also

add64m.

Notes

None.

5.1.3 and64m

Purpose

Find the bitwise AND of two unsigned 64-bit values.

Synopsis

```
#include "u_signed64.h"

u_signed64 and64m (
        u_signed64      ll1,        /* IN */
        u_signed64      ll2);       /* IN */
```

Description

The *and64m* macro is called to perform an AND operation of two unsigned 64-bit integer values.

Parameters

ll1	Specifies the first unsigned 64-bit integer.
ll2	Specifies the second unsigned 64-bit integer.

Return values

Upon completion, an unsigned 64-bit integer is returned.

Error conditions

None.

See also

not64m, **or64m**.

Notes

None.

5.1.4 bld64m

Purpose

Build an unsigned 64-bit integer from two unsigned 32-bit integers.

Synopsis

```
#include "u_signed64.h"

u_signed64 bld64m(
        unsigned32          l1,          /* IN */
        unsigned32          l2);         /* IN */
```

Description

The *bld64m* macro is called to construct an unsigned 64-bit integer from 2 unsigned 32-bit integers.

Parameters

l1	Specifies the unsigned 32-bit integer which will occupy the high order portion of the unsigned 64-bit integer.
l2	Specifies the unsigned 32-bit integer which will occupy the low order portion of the unsigned 64-bit integer.

Return values

Upon completion, an unsigned 64-bit integer is returned.

Error conditions

None.

See also

cast64m.

Notes

none.

5.1.5 cast64m

Purpose

Cast an unsigned 32-bit integer into an unsigned 64-bit integer.

Synopsis

```
#include "u_signed64.h"

u_signed64 cast64m(
        unsigned32          l1);         /* IN */
```

Description

The *cast64m* macro is called to cast an unsigned 32-bit integer into an unsigned 64-bit integer.

Parameters

l1 Specifies an unsigned 32-bit integer to be cast into an unsigned 64-bit integer.

Return values

Upon completion, an unsigned 64-bit integer is returned.

Error conditions

None.

See also

cast32m.

Notes

None.

5.1.6 div64m

Purpose

Divide an unsigned 64-bit integer by an unsigned 32-bit integer.

Synopsis

```
#include "u_signed64.h"

u_signed64 div64m(
        u_signed64      ll1,        /* IN */
        unsigned32      l1);        /* IN */
```

Description

The *div64m* macro is called to divide an unsigned 64-bit value by an unsigned 32-bit value.

Parameters

ll1 Specifies an unsigned 64-bit integer numerator.

l1 Specifies an unsigned 32-bit integer divisor.

Return values

Upon completion, an unsigned 64-bit integer is returned.

Error conditions

None.

See also

div2x64m, **div_cl64m**, **div_2xcl64m**, **mod64m**, **mod2x64m**.

Notes

None.

5.1.7 div2x64m

Purpose

Divide an unsigned 64-bit integer by an unsigned 64-bit integer.

Synopsis

```
#include "u_signed64.h"

u_signed64 div64m(
        u_signed64      ll1,        /* IN */
        u_signed64      ll2);       /* IN */
```

Description

The *div64m* macro is called to divide an unsigned 64-bit value by an unsigned 64-bit value.

Parameters

ll1	Specifies an unsigned 64-bit integer numerator.
ll2	Specifies an unsigned 64-bit integer divisor.

Return values

Upon completion, an unsigned 64-bit integer is returned.

Error conditions

None.

See also

div64m, **div2x64m**, **div_cl64m**, **mod64m**, **mod2x64m**.

Notes

None.

5.1.8 div_cl64m

Purpose

Divide an unsigned 64-bit integer by an unsigned 32-bit integer and return the integer ceiling of the result.

Synopsis

```
#include "u_signed64.h"

u_signed64 div_cl64m(
        u_signed64      ll1,        /* IN */
        unsigned32      l1);        /* IN */
```

Description

The *div_cl64m* macro is called to return the integer ceiling resulting from the division of an unsigned 64-bit value by an unsigned 32-bit value.

Parameters

ll1	Specifies an unsigned 64-bit integer numerator.
l1	Specifies an unsigned 32-bit integer numerator.

Return values

Upon completion, an unsigned 64-bit integer is returned.

Error conditions

None.

See also

div64m, **div2x64m**, **div_2x64m**, **mod64m**.

Notes

None.

5.1.9 div_2xcl64m

Purpose

Divide an unsigned 64-bit integer by an unsigned 64-bit integer and return the integer ceiling of the result.

Synopsis

```
#include "u_signed64.h"

u_signed64 div_cl64m(
        u_signed64          ll1,            /* IN */
        u_signed64          ll2);           /* IN */
```

Description

The *div_cl64m* macro is called to return the integer ceiling resulting from the division of an unsigned 64-bit value by an unsigned 64-bit value.

Parameters

ll1	Specifies an unsigned 64-bit integer numerator.
ll2	Specifies an unsigned 64-bit integer numerator.

Return values

Upon completion, an unsigned 64-bit integer is returned.

Error conditions

None.

See also

div64m, **div2x64m**, **div_cl64m**, **mod64m**, **mod2x4m**.

Notes

None.

5.1.10 eqz64m

Purpose

Determine if an unsigned 64-bit integer is zero.

Synopsis

```
#include "u_signed64.h"

int eqz64m (
        u_signed64          ll1);          /* IN */
```

Description

The *eqz64m* macro is called to check if an unsigned 64-bit integer equals zero.

Parameters

ll1	Specifies the unsigned 64-bit integer.

Return values

1 is returned if ll1 equals zero. Otherwise 0 is returned.

Error conditions

None.

See also

eq64m, **ge64m**, **gt64m**, **le64m**, **lt64m**, **neqz64m**, **neq64m**.

Notes

None.

5.1.11 eq64m

Purpose

Compare two unsigned 64-bit integers for equality.

Synopsis

```
#include "u_signed64.h"

int eq64m (
        u_signed64          ll1,           /* IN */
        u_signed64          ll2);          /* IN */
```

Description

The *eq64m* macro is called to check if two unsigned 64-bit integer values are equal.

Parameters

ll1	Specifies the first unsigned 64-bit integer.
ll2	Specifies the second unsigned 64-bit integer.

Return values

1 is returned if ll1 = ll2. Otherwise 0 is returned.

Error conditions

None.

See also

eqz64m, **ge64m**, **gt64m**, **le64m**, **lt64m**, **neqz64m**, **neq64m**.

Notes

None.

5.1.12 ge64m

Purpose

Perform greater than or equal check between two unsigned 64-bit integers.

Synopsis

```
#include "u_signed64.h"

int ge64m(
        u_signed64      ll1,        /* IN */
        u_signed64      ll2);       /* IN */
```

Description

The *ge64m* macro is called to determine if the first unsigned 64-bit integer value is greater than or equal to the second unsigned 64-bit integer.

Parameters

ll1	Specifies the first unsigned 64-bit integer.
ll2	Specifies the second unsigned 64-bit integer.

Return values

1 is returned if ll1 >=ll2. Otherwise 0 is returned.

Error conditions

None.

See also

eqz64m, **eq64m**, **gt64m**, **le64m**, **lt64m**, **neqz64m**, **neq64m**.

Notes

None.

5.1.13 gt64m

Purpose

Perform greater than check between two unsigned 64-bit integers.

Synopsis

```
#include "u_signed64.h"

int gt64m(
        u_signed64      ll1,        /* IN */
        u_signed64      ll2);       /* IN */
```

Description

The *gt64m* macro is called to determine if the first unsigned 64-bit integer value is greater than the second unsigned 64-bit integer.

Parameters

ll1	Specifies the first unsigned 64-bit integer.
ll2	Specifies the second unsigned 64-bit integer.

Return values

1 is returned if ll1 > ll2. Otherwise 0 is returned.

Error conditions

None.

See also

eqz64m, **eq64m**, **ge64m**, **le64m**, **lt64m**, **neq64m**.

Notes

None.

5.1.14 high32m

Purpose

Extract the high order 32-bits from an unsigned 64-bit integer.

Synopsis

```
#include "u_signed64.h"

unsigned32 high32m(
        u_signed64          ll1);          /* IN */
```

Description

The *high32m* macro is called to extract an unsigned 32-bit integer from the high order 32-bits of an unsigned 64-bit integer.

Parameters

ll1	Specifies the unsigned 64-bit integer.

Return values

Upon completion, an unsigned 32-bit integer is returned.

Error conditions

None.

See also

low32m.

Notes

None.

5.1.15 le64m

Purpose

Perform less than or equal check between two unsigned 64-bit integers.

Synopsis

```
#include "u_signed64.h"

int le64m(
        u_signed64          ll1,          /* IN */
        u_signed64          ll2);         /* IN */
```

Description

The *le64m* macro is called to determine if the first unsigned 64-bit integer value is less than or equal to the second unsigned 64-bit integer.

Parameters

ll1	Specifies the first unsigned 64-bit integer.
ll2	Specifies the second unsigned 64-bit integer.

Return values

1 is returned if ll1 <= ll2. Otherwise 0 is returned.

Error conditions

None.

See also

eqz64m, **eq64m**, **ge64m**, **gt64m**, **lt64m**, **neq64m**.

Notes

None.

5.1.16 low32m

Purpose

Extract the low order 32-bits from an unsigned 64-bit integer.

Synopsis

```
#include "u_signed64.h"

unsigned32 low32m(
        u_signed64          ll1);         /* IN */
```

Description

The *low32m* macro is called to extract an unsigned 32-bit integer from the low order 32-bits of an unsigned 64-bit integer.

Parameters

ll1 Specifies the unsigned 64-bit integer.

Return values

Upon completion, a 32-bit unsigned integer is returned.

Error conditions

None.

See also

high32m.

Notes

None.

5.1.17 lt64m

Purpose

Perform less than check between two unsigned 64-bit integers.

Synopsis

```
#include "u_signed64.h"

int lt64m(
        u_signed64      ll1,        /* IN */
        u_signed64      ll2);       /* IN */
```

Description

The *lt64m* macro is called to determine if the first unsigned 64-bit integer value is less than the second unsigned 64-bit integer.

Parameters

ll1 Specifies the first unsigned 64-bit integer.

ll2 Specifies the second unsigned 64-bit integer.

Return values

1 is returned if ll1 < ll2. Otherwise 0 is returned.

Error conditions

None.

See also

eqz64m, **eq64m**, **ge64m**, **gt64m**, **le64m**, **neq64m**.

Notes

None.

5.1.18 mod64m

Purpose

Determine the remainder on division of an unsigned 64-bit integer by an unsigned 32-bit integer.

Synopsis

```
#include "u_signed64.h"

u_signed64 mod64m(
        u_signed64      ll1,        /* IN */
        unsigned32      l1);        /* IN */
```

Description

The *mod64m* macro is called to perform a modulus of an unsigned 64-bit integer value by an unsigned 32-bit integer value.

Parameters

ll1	Specifies the unsigned 64-bit integer to divide.
l1	Specifies the unsigned 32-bit integer divisor.

Return values

Upon completion, an unsigned 64-bit integer is returned.

Error conditions

None.

See also

div64m, **div2x64m**, **div_cl64m**, **div_2xcl64m**, **mod2x64m**.

Notes

None.

5.1.19 mod2x64m

Purpose

Determine the remainder on division of an unsigned 64-bit integer by an unsigned 64-bit integer.

Synopsis

```
#include "u_signed64.h"

u_signed64 mod64m(
        u_signed64      ll1,        /* IN */
        u_signed64      ll2);       /* IN */
```

Description

The *mod64m* macro is called to perform a modulus of an unsigned 64-bit integer value by an unsigned 64-bit integer value.

Parameters

ll1	Specifies the unsigned 64-bit integer to modulus.
ll2	Specifies the unsigned 64-bit integer modulus value.

Return values

Upon completion, an unsigned 64-bit integer is returned.

Error conditions

None.

See also

div64m, **div2x64**, **div_cl64m**, **div_2xcl64m**, **mod64m**.

Notes

None.

5.1.20 mul64m

Purpose

Multiply an unsigned 64-bit integer by an unsigned 32-bit integer.

Synopsis

```
#include "u_signed64.h"

u_signed64 mul64m(
        u_signed64      ll1,        /* IN */
        unsigned32      l1);        /* IN */
```

Description

The *mul64m* macro is called to multiply an unsigned 64-bit integer value by an unsigned 32-bit integer value.

Parameters

ll1	Specifies the unsigned 64-bit integer multiplier.
l1	Specifies the unsigned 32-bit integer multiplier.

Return values

Upon completion, an unsigned 64-bit integer is returned.

Error conditions

None.

See also

None.

Notes

None.

5.1.21 neqz64m

Purpose

Determine if an unsigned 64-bit integer is nonzero.

Synopsis

```
#include "u_signed64.h"

int neqz64m(
        u_signed64          ll1);          /* IN */
```

Description

The *neqz64m* macro is called to check if an unsigned 64-bit integer is nonzero.

Parameters

ll1	Specifies the unsigned 64-bit integer.

Return values

1 is returned if ll1 != 0. Otherwise 0 is returned.

Error conditions

None.

See also

eqz64m, **eq64m**, **ge64m**, **gt64m**, **le64m**, **lt64m**, **neq64m**.

Notes

None.

5.1.22 not64m

Purpose

Perform a bitwise NOT of an unsigned 64-bit integer.

Synopsis

```
#include "u_signed64.h"

u_signed64 not64m(
        u_signed64          ll1);          /* IN */
```

Description

The *not64m* macro is called to perform a bitwise NOT of an unsigned 64-bit integer value.

Parameters

ll1	Specifies an unsigned 64-bit integer.

Return values

Upon completion, an unsigned 64-bit integer is returned.

Error conditions

None.

See also

and64m, **or64m**.

Notes

None.

5.1.23 or64m

Purpose

Find the bitwise OR of two unsigned 64-bit values.

Synopsis

```
#include "u_signed64.h"

or64m (
        u_signed64          ll1,          /* IN */
        u_signed64          ll2);         /* IN */
```

Description

The *or64m* macro is called to perform an OR operation of two unsigned 64-bit integer values.

Parameters

ll1	Specifies the first unsigned 64-bit integer.
ll2	Specifies the second unsigned 64-bit integer.

Return values

Upon completion, an unsigned 64-bit integer is returned.

Error conditions

None.

See also

and64m, **not64m**.

Notes

None.

5.1.24 shl64m

Purpose

Shift an unsigned 64-bit integer left by an unsigned 32-bit unsigned integer amount.

Synopsis

```
#include "u_signed64.h"

u_signed64 shl64m(
        u_signed64          ll1,          /* IN */
        unsigned32          l1);          /* IN */
```

Description

The *shl64m* macro is called to shift an unsigned 64-bit integer left by an unsigned 32-bit integer count.

Parameters

ll1 Specifies an unsigned 64-bit integer to be shifted.

l1 Specifies an unsigned 32-bit integer shift count.

Return values

Upon completion, an unsigned 64-bit integer is returned.

Error conditions

None.

See also

shr64m.

Notes

None.

5.1.25 shr64m

Purpose

Shift an unsigned 64-bit integer right by an unsigned 32-bit integer amount.

Synopsis

```
#include "u_signed64.h"

u_signed64 shr64m(
        u_signed64      ll1,        /* IN */
        unsigned32      l1);        /* IN */
```

Description

The *shr64m* macro is called to shift an unsigned 64-bit integer right by an unsigned 32-bit integer count.

Parameters

ll1 Specifies an unsigned 64-bit integer to be shifted.

l1 Specifies an unsigned 32-bit integer shift count.

Return values

Upon completion, an unsigned 64-bit integer is returned.

Error conditions

None.

See also

shl64m.

Notes

None.

5.1.26 sub64m

Purpose

Subtract two unsigned 64-bit integers.

Synopsis

```
#include "u_signed64.h"

u_signed64 sub64m(
        u_signed64      ll1,        /* IN */
        u_signed64      ll2);       /* IN */
```

Description

The *sub64m* macro is called to subtract two unsigned 64-bit integer values.

Parameters

ll1	Specifies an unsigned 64-bit integer.
ll2	Specifies an unsigned 64-bit integer to subtract from the first operand.

Return values

Upon completion, an unsigned 64-bit integer is returned.

Error conditions

None.

See also

sub64_3m.

Notes

None.

5.1.27 sub64_3m

Purpose

Subtract two unsigned 64-bit integers and store the result in a separate integer field.

Synopsis

```
#include "u_signed64.h"

void  sub64m(
        u_signed64      ll1,        /* OUT */
        u_signed64      ll2,        /* IN */
        u_signed64      ll3);       /* IN */
```

Description

The *sub64_3m* macro is called to add two unsigned 64-bit integers and store the result into a third unsigned 64-bit integer.

Parameters

ll1	Specifies the result of the subtraction of the 2 unsigned 64-bit operands.
ll2	Specifies the first unsigned 64-bit integer subtraction operand.
ll3	Specifies an unsigned 64-bit integer to subtract from the first operand.

Return values

None. The result of the subtraction is stored in the first parameter.

Error conditions

None.

See also

sub64m.

Notes

None.

5.2 Data Definitions

This section describes key internal data definitions and all externally used data definitions which are provided by this subsystem. A data definition may be represented by constructs such as data structures and constants. For each data definition, a description, format (including parameter descriptions), and clients which access the data definition are provided.

5.2.1 u_signed64

Description

u_signed64 is the type for an unsigned 64-bit integer.

Format

For big-endian platforms, the u_signed64 type is defined in the u_signed64.h file as follows:

```
typedef struct{
     unsigned long h;
     unsigned long l;
} u_signed64;
```

For little-endian platforms, the u_signed64 type is defined in the u_signed64.h file as follows:

```
typedef struct{
```

```
     unsigned long l;
     unsigned long h;
} u_signed64;
```

5.2.2 unsigned32

Description
unsigned32 is the type for an unsigned 32-bit integer.

Format
The unsigned32 type is defined in the u_signed64.h file as follows:

```
typedef unsigned long unsigned32;
```

Chapter 6 MPI-IO API Functions

This chapter specifies the MPI-IO application programming interface. Specifically, the following information is provided:

- Application Programming Interfaces
- C++ Language Bindings
- Fortran Language Bindings
- Data Definitions
- Troubleshooting

Appendix F provides programming examples for MPI-IO.

6.1 Application Programming Interfaces

This section describes all APIs that are provided for use by a client application. The definitions of these APIs follow the MPI-2 Standard. Implementation-defined aspects of the standard are noted in the descriptions of the individual functions.

Each API is designated as being collective or noncollective. Collective functions must be called by all the processes that participated in opening a file (e.g., for **MPI_File_set_view**, all the processes in the communicator group specified in the *comm* argument to **MPI_File_open** must call the function). Otherwise, the application may eventually deadlock. Note that the communicator group specified in opening a file need not include all the processes running in a parallel program.

Processes that are members of the communicator group that opens a given file are called *participating processes* for that file. Participating processes must issue corresponding collective calls in the same order to avoid deadlock. For example, if a set of processes opens two files, the processes must do so in the same order.

Among collective calls, the descriptions further distinguish between synchronizing and nonsynchronizing calls. For synchronizing calls, no process will return from its call until all participating processes have made their corresponding call. Nonsynchronizing calls may return on

individual processes before the other participating processes have issued the call. The designation of a call as collective or noncollective is specified in the MPI-2 standard. Whether or not a collective call synchronizes is a characteristic of the implementation.

Synchronizing operations should not be confused with blocking operations. MPI-2 defines both blocking and nonblocking versions of its read and write interfaces. Blocking versions do not return until the requesting process has completed its data transfer and the buffer can safely be read (in the case of read operations) or reused (in the case of writes). Nonblocking calls may return before the transfer is complete so that computation may proceed in parallel with I/O. The program must later test for completion of the transfer using **MPI_Test** or **MPI_Wait**.

The APIs are designed to work in a multithreaded application. Multiple threads can safely access different open file handles. However, as with MPI collective communications, collective access to the same file handle from multiple threads is not guaranteed to be safe. Although this implementation attempts to detect such errors, it is not able to do so in every case. Accessing the same file from different threads using different file handles (obtained from multiple calls to **MPI_File_open**) is safe.

The MPI-IO API and the HPSS Client API can access all the same files, but an MPI-IO file handle returned by **MPI_File_open** cannot be used as a file descriptor in any HPSS calls, and HPSS API file descriptors cannot be used in MPI-IO calls.

The APIs provided to client applications are divided into the following subsections:

- File Manipulation
- File Access
- File Interoperability
- File Consistency
- Error Handling
- File Hints

Each API specification includes the following information:

Name

Synopsis

Description

Parameters

Return Values

Error Conditions

See Also

Notes

There are a number of errors that may be returned from any MPI-IO API call. Rather than repeating them in each API specification, the common errors are given here:

MPI_ERR_ENOCONNECT	Unable to connect to HPSS file system; HPSS may be down.
MPI_ERR_EPERM	The user's credentials could not be established; DCE may be down.
MPI_ERR_ETIMEDOUT	An operation timed out or received a communication error, and the operation could not be successfully retried; HPSS or DCE may be down.
MPI_ERR_FILE_IN_USE	The file is currently being used by another process.
MPI_ERR_INTERN	An internal MPI-IO error occurred.
MPI_ERR_IO	An internal HPSS error occurred.
MPI_ERR_NOT_INITIALIZED	MPI (MPI-IO) is not initialized.

The following errors can occur with any collective call:

MPI_ERR_DUP_CLIENT	Duplicate collective operation for a client process.
MPI_ERR_EXPECTED_ATOMICITY	Collective call to **MPI_File_set_atomicity** expected.
MPI_ERR_EXPECTED_CLOSE	Collective call to **MPI_File_close** expected.
MPI_ERR_EXPECTED_OPEN	Collective call to **MPI_File_open** expected.
MPI_ERR_EXPECTED_SEEK	Collective call to **MPI_File_seek_shared** expected.
MPI_ERR_EXPECTED_SIZE	Collective call to **MPI_File_set_size** or **MPI_File_preallocate** expected.
MPI_ERR_EXPECTED_SYNC	Collective call to **MPI_File_sync** expected.
MPI_ERR_EXPECTED_VIEW	Collective call to **MPI_File_set_view** expected.
MPI_ERR_MEMBER	Client is not a member of the communicator group for an open file.

See 6.1.5 "Error Handling" on the effect on error returns from setting file error handlers.

6.1.1 File Manipulation

The APIs described in this section are used to open (including create), close, and delete files, and to set or get characteristics of the file that control data layout and access within the file. The characteristics of an open file include its path name within the file system on which it is stored, its set of participating processes, its access mode and permissions, and its size. These characteristics are specified or determined at the time that the file is opened. Each process that participates in a file open is also allowed to specify a file view.

The following definitions will clarify these concepts:

An MPI-IO *file* is an ordered collection of typed data items. MPI-IO supports random or sequential access to any integral set of these items. A file is opened collectively by a group of processes. A *file handle* is an opaque object created by **MPI_File_open** and freed by **MPI_File_close**. All operations on an open file reference the file through the file handle.

A *view* defines the current set of data visible and accessible from an open file as an ordered set of etypes. Each process has its own view of the file, defined by: a displacement, an etype, and a filetype. The pattern described by a filetype is repeated, beginning at the displacement, to define the view.

A file *displacement* is an absolute byte position relative to the beginning of a file. The displacement defines the byte location where a view begins.

An *etype* is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes.

A *filetype* is the basis for partitioning a file among processes and defines a template for accessing the file. A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. Note that HPSS limits the number of holes that a file can contain (approximately 1000). This limit may preclude some noncollective writes for some filetype choices, even though complementary writes by other processes will fill the holes. In this case, collective writes should be used to avoid the file fragmentation.

An *offset* is a position in the file relative to the current view, expressed as a count of etypes. Holes in the view's filetype are skipped when calculating this position. Offset 0 is the location of the first etype visible in the view.

The *size* of a file is measured in bytes from the beginning of the file. A newly created file has a size of zero bytes. Using the size as an absolute displacement gives the position of the byte immediately following the last byte in the file. For any given view, the *end of file* is the offset of the first etype accessible in the current view starting after the last byte in the file. The end of file for a given view is its *end of view*.

A *file pointer* is an implicit offset maintained by MPI-IO. An *individual file pointer* is a file pointer that is local to each process that opened a file. A *shared file pointer* is a file pointer that is shared by the group of processes that opened a file.

6.1.1.1 MPI_File_open

Purpose

Optionally create and open an MPI-IO file.

Synopsis

```
#include <mpio.h>

int
MPI_File_open(
        MPI_Comm      comm,          /* IN  */
        char *        filename,      /* IN  */
        int           amode,         /* IN  */
        MPI_Info      info,          /* IN  */
        MPI_File *    fh);           /* OUT */
```

Description

MPI_File_open opens the file identified by *filename* on all processes in the *comm* communicator (which may be MPI_COMM_SELF). **MPI_File_open** is a collective routine: all processes in the communicator must provide the same value for *amode* and for *filename* (i.e., *filename*s on all processes must be the same text). Values for *info* may vary, depending on the MPI_Info keys specified (see 6.1.6, "File Hints"). *comm* must be an intracommunicator. The file handle returned in *fh* can subsequently be used to access the file until the file is closed using **MPI_File_close**.

In general, the *amode* flags have the same semantics as their POSIX counterparts. Exactly one of MPI_MODE_RDONLY, MPI_MODE_RDWR, or MPI_MODE_WRONLY must be specified. It is erroneous to specify MPI_MODE_CREATE or MPI_MODE_EXCL in conjunction with MPI_MODE_RDONLY; it is erroneous to specify MPI_MODE_SEQUENTIAL together with MPI_MODE_RDWR. The MPI_MODE_UNIQUE_OPEN mode allows an implementation to optimize access by eliminating the overhead of file locking; it is erroneous to open a file in this mode if the file can be concurrently opened elsewhere. The MPI_MODE_SEQUENTIAL mode allows an implementation to optimize access to sequential devices; it is erroneous to attempt nonsequential access to a file that has been opened in this mode.

Initially, all processes view the file as a linear byte stream, and each process views data in its own native data representation. The file view can be changed using **MPI_File_set_view**. Files are opened by default using nonatomic file consistency semantics; more stringent atomic mode consistency semantics can be set using **MPI_File_set_atomicity**.

Parameters

comm — MPI communicator specifying which processes will access the file.

filename — HPSS path name of the file (length must be less than or equal to HPSS_MAX_PATH_NAME).

amode — Access mode of the file; it is a bit vector OR of one or more of the following flags:

MPI_MODE_RDONLY - open file for reading only.

MPI_MODE_RDWR - open file for reading and writing.

MPI_MODE_WRONLY - open file for writing only.

MPI_MODE_CREATE - create file if it does not already exist.

MPI_MODE_EXCL - error if creating file that already exists.

MPI_MODE_DELETE_ON_CLOSE - delete file when file is closed.

MPI_MODE_UNIQUE_OPEN - file will not be concurrently opened elsewhere. Enables automatic file caching.

MPI_MODE_SEQUENTIAL - file will only be accessed sequentially.

MPI_MODE_APPEND - set initial position of all file pointers to end of file.

info Handle for opaque data structure containing hints regarding file access patterns and file system specifics. The constant MPI_INFO_NULL can be used when no hints need to be specified. See 6.1.6 for more information.

fh Returned file handle.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*filename* or *fh* argument is NULL or *comm* argument is MPI_COMM_NULL.
MPI_ERR_ACCESS	Access permission is denied to the file or to a component of the *filename* path prefix.
MPI_ERR_AMODE	Invalid *amode* argument.
MPI_ERR_BAD_FILE	Invalid *filename* argument (empty or too long).
MPI_ERR_COMM	Invalid *comm* argument.
MPI_ERR_EISDIR	The specified *filename* is a directory.
MPI_ERR_EMFILE	No more file descriptors available.
MPI_ERR_ENFILE	Too many open files in the system.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_ENOTDIR	A component of the *filename* path is not a directory.
MPI_ERR_FILE_EXISTS	MPI_MODE_CREATE and MPI_MODE_EXCL are specified and the file already exists.
MPI_ERR_HINTS	Invalid *info* argument.

MPI_ERR_NO_SPACE	Insufficient free space on the HPSS file system to create, write, or preallocate a file.
MPI_ERR_NO_SUCH_FILE	MPI_MODE_CREATE is not set and the named file does not exist or a component of the named file path does not exist.
MPI_ERR_NOT_SAME	*amode* or *filename* arguments or *info* argument key values do not match across processes.

See Also

MPI_File_close, **MPI_File_set_view**, **MPI_File_set_info**.

Notes

Collective; synchronizing.

6.1.1.2 MPI_File_close

Purpose

Close an MPI-IO file.

Synopsis

```
#include <mpio.h>

int
MPI_File_close(
        MPI_File *      fh          /* IN/OUT */
);
```

Description

MPI_File_close closes the file associated with *fh*. If the file was opened with access mode MPI_MODE_DELETE_ON_CLOSE, the file is deleted. The file handle *fh* is set to MPI_FILE_NULL.

The user is responsible for ensuring that all outstanding requests associated with *fh* have completed before calling **MPI_File_close**.

Parameters

fh	File handle of file to be closed.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*fh* argument is NULL.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_FILE	Invalid file handle.

MPI_ERR_PENDING_RW	There are uncompleted read or write operations pending for *fh*.

See Also

MPI_File_open, MPI_File_delete.

Notes

Collective; synchronizing.

6.1.1.3 MPI_File_delete

Purpose

Delete an MPI-IO file.

Synopsis

```
#include <mpio.h>

int
MPI_File_delete(
        char *      filename    /* IN */
        MPI_Info    info        /* IN */
);
```

Description

MPI_File_delete removes the file identified by *filename*. The *info* argument can be used to provide hints regarding file system specifics; the constant MPI_INFO_NULL can be used when no hints need to be specified.

Parameters

filename	HPSS path name of the file to delete.
info	File hints.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*filename* argument is NULL.
MPI_ERR_ACCESS	Access permission is denied to the file or to a component of the *filename* path prefix.
MPI_ERR_BAD_FILE	Invalid *filename* (empty or too long).
MPI_ERR_EISDIR	The specified *filename* is a directory.
MPI_ERR_ENOMEM	Unable to allocate program space.

MPI_ERR_ENOTDIR	A component of the *filename* path is not a directory.
MPI_ERR_NO_SUCH_FILE	Named file does not exist.

See Also

MPI_File_open, MPI_File_close.

Notes

Noncollective.

Currently, the implementation ignores any hints in **MPI_File_delete**.

6.1.1.4 MPI_File_set_size

Purpose

Change the size of an open file.

Synopsis

```
#include <mpio.h>

int
MPI_File_set_size(
        MPI_File      fh,          /* IN/OUT */
        MPI_Offset    size         /* IN     */
);
```

Description

MPI_File_set_size resizes the file associated with the file handle *fh*. *size* is measured in bytes from the beginning of the file. All participating processes must specify the same value for *size*. Regions of the file that have been previously written are unaffected. If the new *size* is smaller than the current file size, the file is truncated. If the new *size* is greater than the current file size, the file size becomes *size*; the values of data in the new regions in the file are undefined, and until written, it is not guaranteed that the file space for the new regions is allocated. Use **MPI_File_preallocate** to force the file space to be reserved.

MPI_File_set_size does not affect the individual or shared file pointers.

It is erroneous to call **MPI_File_set_size** while uncompleted nonblocking or split collective calls are pending for *fh*.

Parameters

fh	File handle.
size	New size of the file.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NOT_SAME	*size* arguments do not match across processes.
MPI_ERR_OFFSET	Negative *size* specified.
MPI_ERR_PENDING_RW	There are uncompleted read or write operations pending for *fh*.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL.

See Also

MPI_File_get_size, MPI_File_preallocate.

Notes

Collective; synchronizing.

6.1.1.5 MPI_File_preallocate

Purpose

Change the allocated size of an open file.

Synopsis

```
#include <mpio.h>

int
MPI_File_preallocate(
        MPI_File      fh,          /* IN / OUT */
        MPI_Offset    size         /* IN     */
);
```

Description

MPI_File_preallocate ensures that storage space is allocated for the first *size* bytes of the file associated with *fh*. All participating processes must specify the same value for *size*. Regions of the file that have been previously written are unaffected. The values of data in the new regions of the file are undefined. If *size* is less than or equal to the current file size, the file size is unchanged. If *size* is greater than the current file size, the file size increases to *size*.

MPI_File_preallocate does not affect the individual or shared file pointers.

It is erroneous to call **MPI_File_preallocate** while uncompleted nonblocking or split collective calls are pending for *fh*.

Parameters

fh	File handle.

size	Allocated size to ensure for the file.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NO_SPACE	Insufficient free space on the HPSS file system to create, write, or preallocate a file.
MPI_ERR_NOT_SAME	*size* arguments do not match across processes.
MPI_ERR_OFFSET	Negative *size* specified.
MPI_ERR_PENDING_RW	There are uncompleted read or write operations pending for *fh*.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL.

See Also

MPI_File_get_size, MPI_File_set_size.

Notes

Collective; synchronizing.

6.1.1.6 MPI_File_get_size

Purpose

Get the current size of an open file.

Synopsis

```
#include <mpio.h>

int
MPI_File_get_size(
        MPI_File        fh,         /* IN  */
        MPI_Offset *    size        /* OUT */
);
```

Description

MPI_File_get_size returns in *size* the current size in bytes of the file associated with the file handle *fh*.

Parameters

fh	File handle.

size	Returned size of the file.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*size* argument is NULL.
MPI_ERR_FILE	Invalid file handle.

See Also

MPI_File_set_size, MPI_File_preallocate.

Notes

Noncollective.

6.1.1.7 MPI_File_get_group

Purpose

Get the communicator group associated with an open file.

Synopsis

```
#include <mpio.h>

int
MPI_File_get_group(
        MPI_File      fh,          /* IN  */
        MPI_Group *   group        /* OUT */
);
```

Description

MPI_File_get_group returns in *group* a duplicate of the group of the MPI communicator used to open the file associated with *fh*. The user is responsible for freeing *group*.

Parameters

fh	File handle.
group	Returned communicator group.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*group* argument is NULL.
MPI_ERR_FILE	Invalid file handle.

See Also

MPI_File_open.

Notes

Noncollective.

6.1.1.8 MPI_File_get_amode

Purpose

Get the mode associated with an open file.

Synopsis

```
#include <mpio.h>

int
MPI_File_get_amode(
        MPI_File    fh,         /* IN  */
        int *       amode       /* OUT */
);
```

Description

MPI_File_get_amode returns in *amode* the mode associated with the open file *fh*.

Parameters

fh	File handle.
amode	Returned file mode.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*amode* argument is NULL.
MPI_ERR_FILE	Invalid file handle.

I am assuming we can just get comm from local ft. Otherwise, add an MPI_ERR_ENOMEM from sending a message.

See Also

MPI_File_open.

Notes

Noncollective.

6.1.1.9 MPI_File_set_info

Purpose

Set new hints for an open file.

Synopsis

```
#include <mpio.h>
```

```
int
MPI_File_set_info(
        MPI_File        fh,         /* IN / OUT */
        MPI_Info        info        /* IN     */
);
```

Description

MPI_File_set_info updates the file hints associated with the open file *fh*. The *info* object may be different on each process, but any *info* key values that are required to be the same on all processes (see 6.1.6 "File Hints") must appear in each process's *info* object.

Parameters

fh	File handle.
info	File hints regarding file access patterns and file system specifics.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NOT_SAME	*info* argument key values do not match across processes.

See Also

MPI_File_get_info, MPI_File_open, MPI_File_set_view.

Notes

Collective; nonsynchronizing.

Currently the implementation does not allow changing any of the file hints on an open file.

6.1.1.10 MPI_File_get_info

Purpose

Get the file hints associated with an open file.

Synopsis

```
#include <mpio.h>

int
MPI_File_get_info(
        MPI_File        fh,                 /* IN  */
        MPI_Info *      info_used /* OUT */
);
```

Description

MPI_File_get_info returns in *info_used* a handle for a new info object containing the current setting of all hints associated with the open file *fh*. The user is responsible for freeing *info_used* via **MPI_Info_free**.

Parameters

fh	File handle.
info_used	Returned handle to the current file hints.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*info_used* argument is NULL.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_FILE	Invalid file handle.

See Also

MPI_File_open, MPI_File_set_info, MPI_File_set_view.

Notes

Noncollective.

6.1.1.11 MPI_File_set_view

Purpose

Change the view of the data in an open file.

Synopsis

```
#include <mpio.h>

int
MPI_File_set_view(
        MPI_File        fh,         /* IN / OUT */
        MPI_Offset      disp,       /* IN    */
        MPI_Datatype    etype,      /* IN    */
        MPI_Datatype    filetype,   /* IN    */
        char *          datarep,    /* IN    */
        MPI_Info        info        /* IN    */
);
```

Description

MPI_File_set_view changes the process's view of the data in the open file *fh*. The start of the view is set to *disp* (absolute offset in bytes from the beginning of the file); the type of data is set to *etype*; the distribution of data to processes is set to *filetype*; the representation of data in the file is set to *datarep*; the *info* argument specifies file access patterns or file system specifics to direct optimization. Each participating process must specify the same *datarep* and the same extents of *etype* in the file data representation, but *disp*, *filetype*, and *info* may vary. The individual file pointers and the shared

file pointer are reset to zero.

The datatypes for *etype* and *filetype* must be committed. An *etype* is the unit of data access and positioning; offsets are expressed as a count of *etype*s; file pointers point to the beginning of *etype*s. All *etype* typemap displacements must be nonnegative and monotonically nondecreasing. The *filetype* must be composed of *etypes*. In addition, the extent of any hole in the *filetype* must be a multiple of the *etype*'s extent. The *filetype* typemap displacements need not be distinct, but they cannot be negative and must be monotonically nondecreasing. If the file is opened for writing, neither the *etype* nor the *filetype* is permitted to contain overlapping regions. It is erroneous to use absolute addresses in the construction of *etype* and *filetype*.

If MPI_MODE_SEQUENTIAL was specified when the file was opened, the special displacement MPI_DISPLACEMENT_CURRENT must be passed in *disp*. It is erroneous to use the shared file pointer data access routines unless identical values for *disp* and *filetype* are given (i.e., all processes use the same file view).

It is erroneous to call **MPI_File_set_view** while uncompleted nonblocking or split collective calls are pending for *fh*.

Parameters

fh	File handle.
disp	Byte location of file offset zero.
etype	MPI_Datatype file unit.
filetype	MPI_Datatype that describes process's accessible *etype*s.
datarep	Data representation to be used for file data. See 6.1.3. "File Interoperability"
info	Handle for opaque data structure containing information regarding file access patterns and file system specifics.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*datarep* argument is NULL or empty, or *etype* or *filetype* argument is MPI_DATATYPE_NULL.
MPI_ERR_DISPLACEMENT	Invalid *disp*.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_ETYPE	Invalid *etype*.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_FILETYPE	Invalid *filetype*.

MPI_ERR_HINTS	Invalid *info*.
MPI_ERR_NOT_SAME	*datarep* or *etype* arguments or *info* argument key values do not match across processes.
MPI_ERR_PENDING_RW	There are uncompleted read or write operations pending for *fh*.
MPI_ERR_UNSUPPORTED_DATAREP	Invalid *datarep*.

See Also

MPI_File_open, MPI_File_get_view, MPI_File_get_type_extent, MPI_Register_datarep.

Notes

Collective, synchronizing.

6.1.1.12 MPI_File_get_view

Purpose

Get the current view of the data in an open file.

Synopsis

```
#include <mpio.h>

int
MPI_File_get_view(
        MPI_File          fh,          /* IN  */
        MPI_Offset*       disp,        /* OUT */
        MPI_Datatype *    etype,       /* OUT */
        MPI_Datatype *    filetype,    /* OUT */
        char *            datarep      /* OUT */
);
```

Description

MPI_File_get_view returns the process's current view of the data in the open file *fh*. The current value of the displacement is returned in *disp*. The *etype* and *filetype* returned are new datatypes with typemaps equal to the typemaps of the current etype and filetype of the view, respectively; the user is responsible for freeing derived datatypes that are returned. The *etype* and *filetype* returned are both committed. The data representation is returned in *datarep*; the user is responsible for ensuring that *datarep* is large enough to hold the data representation string (i.e., allows up to MPI_MAX_DATAREP_STRING characters).

Parameters

fh	File handle.
disp	The returned displacement.
etype	The returned *etype*.
filetype	The returned *filetype*.

datarep	The returned data representation.

Return Values

Upon successful completion, returns MPI_SUCCESS. Otherwise, it returns one of the error conditions below.

Error Conditions

MPI_ERR_ARG	*disp*, *etype*, *filetype*, or *datarep* argument is NULL.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_FILE	Invalid file handle.

See Also

MPI_File_open, MPI_File_set_view.

Notes

Noncollective.

6.1.2 File Access

The APIs in this section are the MPI-IO data access routines. Data is moved between files and processes by calling read and write routines. Read routines move data from a file into memory. Write routines move data from memory into a file. The file is designated by a file handle, *fh*. The location of the file data is specified by an *offset*, which is always in etype units relative the the current view. The data in memory is specified by a triple: *buf*, *count*, and *datatype*. Upon completion, the amount of data accessed by the calling process is returned in a *status*.

A data access routine attempts to transfer *count* data items of type *datatype* between the user's buffer *buf* and the file. The layout of the data in the user's buffer is described by *datatype*, which must be a committed datatype. The type signature of *datatype* must match the type signature of some number of contiguous copies of the etype of the current view.

For a blocking I/O call, the *status* is returned directly; otherwise, it is returned from a call to **MPI_Wait** or **MPI_Test** (or one of their variants). The number of *datatype* entries and predefined elements actually read can be extracted from the *status*, using the routines MPI_Get_count or MPI_Get_elements, respectively.

To detect the end of file, an application can detect when the amount of data read is less than what was requested.

For nonblocking calls, the MPI_Error field of the returned *status* will contain error information, if the return value of the MPI completion call is MPI_ERR_IN_STATUS. For blocking calls, the error information is just the return value of the function.

MPI-2 defines special versions of collective I/O operations that are called split-collective. These operations are split by having an initiating function to begin the I/O and a completing function to end the I/O. MPI-2 allows these operations to be implemented as either blocking or nonblocking. In this implementation, the initiating function is nonblocking, and the completing function is blocking. Hence, the status information is managed the same as for nonblocking calls.

It is erroneous to specify a datatype for reading that contains overlapping regions. If MPI_MODE_SEQUENTIAL was specified when the file was opened, it is erroneous to invoke a read or write routine that uses an explicit offset or an offset specified by an individual file pointer.

Each data access call can be further characterized by its method of positioning, synchronization, and coordination.

Positioning

For routines that use explicit offsets as an argument, that argument value explicitly specifies the current offset in etypes to be used. No file pointer is used or updated.

For routines that use individual file pointers, MPI-IO maintains one individual file pointer per process per file handle. The current value of this pointer implicitly specifies the current offset in etypes to be used by these data access routines. After an individual file pointer operation is initiated, the individual file pointer is updated to point to the next etype after the last one that will be accessed, relative to the current process' file view. Only the individual file pointers are updated; the shared file pointer is not used or updated.

For routines that use shared file pointers, MPI-IO maintains exactly one shared file pointer per file handle. The current value of this pointer implicitly specifies the current offset in etypes to be used by these data access routines. After a shared file pointer operation is initiated, the shared file pointer is updated to point to the next etype after the last one that will be accessed, relative to the current file view. It is erroneous to use the shared file pointer routines if not all processes use the same file view. The effect of multiple calls to shared file pointer routines is defined to behave as if the calls were serialized; for noncollective routines, the serialization ordering is not deterministic. Only the shared file pointer is updated; the individual file pointers are not used or updated.

Synchronism

A blocking I/O call will not return until the I/O request is completed.

A nonblocking I/O call initiates an I/O operation but does not wait for it to complete. A separate request complete call (**MPI_Wait**, **MPI_Test**, or any of their variants) is needed to complete the I/O request. It is erroneous to access the local buffer of a nonblocking data access operation, or to use that buffer as the source or target of other communications, between the initiation and completion of the operation.

Split-collective calls are implemented essentially as nonblocking calls, with the **MPI_File_***R/W***_begin** call initiating the I/O operation, and the **MPI_File_***R/W***_end** call completing the operation. The **MPI_File_***R/W***_end** call will block until the I/O has completed.

Coordination

Each data access is either collective or noncollective. For collective calls, all processes that participated in the file open must participate in the data access.

The completion of a noncollective call depends only on the activity of the calling process. However, the completion of a collective call may depend on the activity of the other processes participating in the collective call.

In general, collective calls may perform much better than their noncollective counterparts. In particular, collective calls where the file views match the striping on the HPSS file system and the

data is distributed across client processes (user buffer datatypes) in uniformly striped chunks will achieve the best performance.

6.1.2.1 MPI_File_read_at

Purpose

Read a file at an explicit offset, noncollectively.

Synopsis

```
#include <mpio.h>

int
MPI_File_read_at(
        MPI_File        fh,         /* IN  */
        MPI_Offset      offset,     /* IN  */
        void *          buf,        /* OUT */
        int             count,      /* IN  */
        MPI_Datatype    datatype,   /* IN  */
        MPI_Status *    status      /* OUT */
);
```

Description

MPI_File_read_at attempts to read from the file associated with *fh* at the specified *offset* for a total number of *count* data items having *datatype* type into the user's buffer *buf*. The *offset* is in etype units relative to the current view. The data is read from those parts of the file specified by the current view. Data is stored into *buf* according to the pattern specified by *datatype*. The number of datatype elements actually read is returned in *status*. All other fields of *status* are undefined.

Parameters

fh	File handle.
offset	File offset in etypes at which to begin reading.
buf	User's memory buffer where data should be stored.
count	Number of *datatype*-sized chunks of data to read.
datatype	MPI datatype specifying data layout in *buf*.
status	Returned MPI_Status object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* or *status* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.

MPI_ERR_EXTENT_CONVERSION
: Error in user-defined extent function.

MPI_ERR_FILE
: Invalid file handle.

MPI_ERR_READ_CONVERSION
: Error in user-defined read conversion function.

MPI_ERR_UNSUPPORTED_OPERATION
: Operation not allowed with MPI_MODE_SEQUENTIAL.

See Also

MPI_File_read_at_all, MPI_File_iread_at, MPI_File_read_at_all_begin.

Notes

Noncollective.

6.1.2.2 MPI_File_read_at_all

Purpose

Read from a file at explicit offsets, collectively.

Synopsis

```
#include <mpio.h>

int
MPI_File_read_at_all(
        MPI_File        fh,         /* IN  */
        MPI_Offset      offset,     /* IN  */
        void *          buf,        /* OUT */
        int             count,      /* IN  */
        MPI_Datatype    datatype,   /* IN  */
        MPI_Status *    status      /* OUT */
);
```

Description

MPI_File_read_at_all attempts to read collectively from the file associated with *fh*. Each process reads at a specified *offset* for a total number of *count* data items having *datatype* type into the user's buffer *buf*. The *offset* is in etype units relative to each process's current view. The data is read from those parts of the file specified by each process's current view. Data is stored into *buf* according to the pattern specified by *datatype*.

Each process may pass different argument values for *offset*, *datatype* and *count*. After all the processes have issued their respective calls, each process attempts to read.

For each process, the number of datatype elements actually read is returned in *status*. All other fields of *status* are undefined.

Parameters

fh
: File handle.

offset	File offset in etypes at which to begin reading.
buf	User's memory buffer where data should be stored.
count	Number of *datatype*-sized chunks of data to read.
datatype	MPI datatype specifying data layout in *buf*.
status	Returned MPI_Status object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* or *status* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_PENDING_RW	A collective operation is already pending for this file.
MPI_ERR_READ_CONVERSION	Error in user-defined read conversion function.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL.

See Also

MPI_File_read_at, MPI_File_read_at_all_begin.

Notes

Collective; synchronizing.

6.1.2.3 MPI_File_write_at

Purpose

Write to a file at an explicit offset, noncollectively.

Synopsis

#include <mpio.h>

```
int
MPI_File_write_at(
        MPI_File          fh,          /* IN / OUT */
        MPI_Offset        offset,      /* IN    */
        void *            buf,         /* IN    */
        int               count,       /* IN    */
        MPI_Datatype      datatype,    /* IN    */
        MPI_Status *      status       /* OUT   */
);
```

Description

MPI_File_write_at attempts to write into the file associated with *fh* at the specifiec *offset* for a total number of *count* data items having *datatype* type from the user's buffer *buf*. The *offset* is in etype units relative to the current view. The data is written into those parts of the file specified by the current view. Data is read from *buf* according to the pattern specified by *datatype*. The number of datatype elements actually written is returned in *status*. All other fields of *status* are undefined.

Parameters

fh	File handle.
offset	File offset in etypes at which to begin writing.
buf	User's memory buffer where data should be read.
count	Number of *datatype*-sized chunks of data to write.
datatype	MPI datatype specifying how data is laid out in *buf*.
status	Returned MPI_Status object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* or *status* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NO_SPACE	Insufficient free space on the HPSS file system to create, write, or preallocate a file.

MPI_ERR_TOO_FRAGMENTED Write operation resulted in excessive file fragmentation.

MPI_ERR_UNSUPPORTED_OPERATION
Operation not allowed with MPI_MODE_SEQUENTIAL.

MPI_ERR_WRITE_CONVERSION Error in user-defined write conversion function.

See Also

MPI_File_write_at_all, MPI_File_iwrite_at, MPI_File_write_at_all_begin.

Notes

Noncollective.

6.1.2.4 MPI_File_write_at_all

Purpose

Write to a file at explicit offsets, collectively.

Synopsis

```
#include <mpio.h>

int MPI_File_write_at_all(
        MPI_File        fh,         /* IN / OUT */
        MPI_Offset      offset,     /* IN    */
        void *          buf,        /* IN    */
        int             count,      /* IN    */
        MPI_Datatype    datatype,   /* IN    */
        MPI_Status *    status      /* OUT   */
);
```

Description

MPI_File_write_at_all attempts to write collectively into the file associated with *fh*. Each process writes at a specified *offset* for a total number of *count* data items having *datatype* type from the user's buffer *buf*. The *offset* is in etype units relative to each process's current view. The data is written into those parts of the file specified by each process's current view. Data is read from *buf* according to the pattern specified by *datatype*.

Each process may pass different argument values for *offset*, *datatype* and *count*. After all the processes have issued their respective calls, each process attempts to write.

For each process, the number of datatype elements actually written is returned in *status*. All other fields of *status* are undefined.

Parameters

fh File handle.

offset File offset in etypes at which to begin writing.

buf User's memory buffer where data should be read.

count Number of *datatype*-sized chunks of data to write.

datatype	MPI datatype specifying how data is laid out in *buf*.
status	Returned MPI_Status object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* or *status* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NO_SPACE	Insufficient free space on the HPSS file system to create, write, or preallocate a file.
MPI_ERR_PENDING_RW	A collective operation is already pending for this file.
MPI_ERR_TOO_FRAGMENTED	Write operation resulted in excessive file fragmentation.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL.
MPI_ERR_WRITE_CONVERSION	Error in user-defined write conversion function.

See Also

MPI_File_write_at, MPI_FIle_iwrite_at, MPI_File_write_at_all_begin.

Notes

Collective; synchronizing.

6.1.2.5 MPI_File_iread_at

Purpose

Read a file at an explicit offset, noncollectively and without blocking.

Synopsis

```
#include <mpio.h>

int
MPI_File_iread_at(
```

```
        MPI_File          fh,          /* IN  */
        MPI_Offset        offset,      /* IN  */
        void *            buf,         /* OUT */
        int               count,       /* IN  */
        MPI_Datatype      datatype,    /* IN  */
        MPI_Request *     request      /* OUT */
);
```

Description

MPI_File_iread_at initiates a read from the file associated with *fh* at the specified *offset* for a total number of *count* data items having *datatype* type into the user's buffer *buf*. The *offset* is in etype units relative to the current view. The data is read from those parts of the file specified by the current view. Data is stored into *buf* according to the pattern specified by *datatype*.

A request handle *request* is returned, which can be used later as the argument to **MPI_Test** or **MPI_Wait** (or any of their variants) to query the status of the read or wait for its completion. The number of *datatype* elements actually read or error information regarding the completion of the read is returned through the *status* argument to **MPI_Test** or **MPI_Wait**.

Parameters

fh	File handle.
offset	File offset in etypes at which to begin reading.
buf	User's memory buffer where data should be stored.
count	Number of datatype-sized chunks of data to read.
datatype	MPI datatype specifying data layout in *buf*.
request	Returned request handle.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* or *request* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_READ_CONVERSION	

Error in user-defined read conversion function.

MPI_ERR_UNSUPPORTED_OPERATION
Operation not allowed with MPI_MODE_SEQUENTIAL.

See Also

MPI_File_read_at, MPI_File_read_at_all_begin.

Notes

Noncollective.

6.1.2.6 MPI_File_iwrite_at

Purpose

Write to a file at an explicit offset, noncollectively and without blocking.

Synopsis

```
#include <mpio.h>

int
MPI_File_iwrite_at(
        MPI_File          fh,          /* IN / OUT */
        MPI_Offset        offset,      /* IN    */
        void *            buf,         /* IN    */
        int               count,       /* IN    */
        MPI_Datatype      datatype,    /* IN    */
        MPI_Request *     request      /* OUT   */
);
```

Description

MPI_File_iwrite_at initiates a write into the file associated with *fh* at the *offset* position for a total number of *count* data items having *datatype* type from the user's buffer *buf*. The *offset* is in etype units relative to the current view. The data is written into those parts of the file specified by the current view. Data is read from *buf* according to the pattern specified by *datatype*.

A request handle *request* is returned, which can be used later as the argument to **MPI_Test** or **MPI_Wait** (or any of their variants) to query the status of the write or wait for its completion. The number of datatype elements actually written or error information regarding the completion of the write is returned through the *status* argument to **MPI_Test** or **MPI_Wait**.

Parameters

fh File handle.

offset File offset in etypes at which to begin writing.

buf User's memory buffer where data should be read.

count Number of *datatype*-sized chunks of data to write.

datatype MPI datatype specifying how data is laid out in *buf*.

request	Returned request handle.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* or *request* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NO_SPACE	Insufficient free space on the HPSS file system to create, write, or preallocate a file.
MPI_ERR_TOO_FRAGMENTED	Write operation resulted in excessive file fragmentation.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL.
MPI_ERR_WRITE_CONVERSION	Error in user-defined write conversion function.

See Also

MPI_File_write_at, MPI_File_write_at_all_begin.

Notes

Noncollective.

6.1.2.7 MPI_File_read

Purpose

Read a file at the offset specified by the file's individual file pointer, noncollectively.

Synopsis

```
#include <mpio.h>

int
MPI_File_read(
        MPI_File        fh,         /* IN/OUT */
        void *          buf,        /* OUT    */
        int             count,      /* IN     */
        MPI_Datatype    datatype,   /* IN     */
```

```
        MPI_Status *        status        /* OUT   */
);
```

Description

MPI_File_read attempts to read from the file associated with *fh* at the offset specified by the current individual file pointer position for a total number of *count* data items having *datatype* type into the user's buffer *buf*. The offset is in etype units relative to the current view. The data is read from those parts of the file specified by the current view. Data is stored into *buf* according to the pattern specified by *datatype*. The number of datatype elements actually read is returned in *status*. All other fields of *status* are undefined.

The individual file pointer is incremented by the amount of data requested (not necessarily the amount actually read), provided the read request was initiated without error.

Parameters

fh	File handle.
buf	User's memory buffer where data should be stored.
count	Number of *datatype*-sized chunks of data to read.
datatype	MPI datatype specifying data layout in *buf*.
status	Returned MPI_Status object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* or *status* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_READ_CONVERSION	Error in user-defined read conversion function.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL.

See Also

MPI_File_read_all, MPI_File_iread, MPI_File_read_all_begin.

Notes

Noncollective.

6.1.2.8 MPI_File_read_all

Purpose

Read a file at the offsets specified by the file's individual file pointers, collectively.

Synopsis

```
#include <mpio.h>

int
MPI_File_read_all(
        MPI_File        fh,         /* IN/OUT */
        void *          buf,        /* OUT    */
        int             count,      /* IN     */
        MPI_Datatype    datatype,   /* IN     */
        MPI_Status *    status      /* OUT    */
);
```

Description

MPI_File_read_all attempts to read collectively from the file associated with *fh*. Each process reads at the offset specified by its current individual file pointer position for a total number of *count* data items having *datatype* type into the user's buffer *buf*. The offset is in etype units relative to each process's current view. The data is read from those parts of the file specified by each process's current view. Data is stored into *buf* according to the pattern specified by *datatype*.

Each process may pass different argument values for *datatype* and *count*. After all the processes have issued their respective calls, each process attempts to read.

Each individual file pointer is incremented by the amount of data requested (not necessarily the amount actually read), provided the read request was initiated without error.

For each process, the number of datatype elements actually read is returned in *status*. All other fields of *status* are undefined.

Parameters

fh	File handle.
buf	User's memory buffer where data should be stored.
count	Number of *datatype*-sized chunks of data to read.
datatype	MPI datatype specifying data layout in *buf*.
status	Returned MPI_Status object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* or *status* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_PENDING_RW	A collective operation is already pending for this file.
MPI_ERR_READ_CONVERSION	Error in user-defined read conversion function.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL.

See Also

MPI_File_read, MPI_File_iread, MPI_File_read_all_begin.

Notes

Collective; synchronizing.

6.1.2.9 MPI_File_write

Purpose

Write a file at the offset specified by the file's individual file pointer, noncollectively.

Synopsis

```
#include <mpio.h>

int
MPI_File_write(
        MPI_File        fh,         /* IN / OUT */
        void *          buf,        /* IN    */
        int             count,      /* IN    */
        MPI_Datatype    datatype,   /* IN    */
        MPI_Status *    status      /* OUT   */
);
```

Description

MPI_File_write attempts to write into the file associated with *fh* at the offset specified by the current individual file pointer position for a total number of *count* data items having *datatype* type from the user's buffer *buf*. The offset is in etype units relative to the current view. The data is

written into those parts of the file specified by the current view. Data is read from *buf* according to the pattern specified by *datatype*. The number of datatype elements actually written is returned in *status*. All other fields of *status* are undefined.

The individual file pointer is incremented by the amount of data requested (not necessarily the amount actually written), provided the write request was initiated without error.

Parameters

fh	File handle.
buf	User's memory buffer where data should be read.
count	Number of *datatype*-sized chunks of data to write.
datatype	MPI datatype specifying how data is laid out in *buf*.
status	Returned MPI_Status object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* or *status* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NO_SPACE	Insufficient free space on the HPSS file system to create, write, or preallocate a file.
MPI_ERR_TOO_FRAGMENTED	Write operation resulted in excessive file fragmentation.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL.
MPI_ERR_WRITE_CONVERSION	Error in user-defined write conversion function.

See Also

MPI_File_write_all, MPI_File_iwrite, MPI_File_write_all_begin.

Notes

Noncollective.

6.1.2.10 MPI_File_write_all

Purpose

Write a file at the offsets specified by the file's individual file pointers, collectively.

Synopsis

```
#include <mpio.h>

int
MPI_File_write_all(
        MPI_File        fh,         /* IN / OUT */
        void *          buf,        /* IN    */
        int             count,      /* IN    */
        MPI_Datatype    datatype,   /* IN    */
        MPI_Status *    status      /* OUT   */
);
```

Description

MPI_File_write_all attempts to write collectively into the file associated with *fh*. Each process writes at the offset specified by its current individual file pointer position for a total number of *count* data items having *datatype* type from the user's buffer *buf*. The offset is in etype units relative to each process's current view. The data is written into those parts of the file specified by each process's current view. Data is read from *buf* according to the pattern specified by *datatype*.

Each process may pass different argument values for *datatype* and *count*. After all the processes have issued their respective calls, each process attempts to write.

Each individual file pointer is incremented by the amount of data requested (not necessarily the amount actually written), provided the write request was initiated without error.

For each process, the number of datatype elements actually written is returned in *status*. All other fields of *status* are undefined.

Parameters

fh	File handle.
buf	User's memory buffer where data should be read.
count	Number of *datatype*-sized chunks of data to write.
datatype	MPI datatype specifying how data is laid out in *buf*.
status	Returned MPI_Status object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* or *status* argument is NULL.

MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NO_SPACE	Insufficient free space on the HPSS file system to create, write, or preallocate a file.
MPI_ERR_PENDING_RW	A collective operation is already pending for this file.
MPI_ERR_TOO_FRAGMENTED	Write operation resulted in excessive file fragmentation.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL.
MPI_ERR_WRITE_CONVERSION	Error in user-defined write conversion function.

See Also

MPI_File_write, MPI_File_iwrite, MPI_File_write_all_begin.

Notes

Collective; synchronizing.

6.1.2.11 MPI_File_iread

Purpose

Read a file at the offset specified by the file's individual file pointer, noncollectively and without blocking.

Synopsis

```
#include <mpio.h>

int
MPI_File_iread(
        MPI_File        fh,         /* IN/OUT */
        void *          buf,        /* OUT    */
        int             count,      /* IN     */
        MPI_Datatype    datatype,   /* IN     */
        MPI_Request *   request     /* OUT    */
);
```

Description

MPI_File_iread initiates a read from the file associated with *fh* at the offset specified by the current individual file pointer position for a total number of *count* data items having *datatype* type into the user's buffer *buf*. The offset is in etype units relative to the current view. The data is read from

those parts of the file specified by the current view. Data is stored into *buf* according to the pattern specified by *datatype*.

The individual file pointer is incremented by the amount of data requested (not necessarily the amount actually read), provided the read request was initiated without error.

A request handle *request* is returned, which can be used later as the argument to **MPI_Test** or **MPI_Wait** (or any of their variants) to query the status of the read or wait for its completion. The number of datatype elements actually read or error information regarding the completion of the read is returned through the *status* argument to **MPI_Test** or **MPI_Wait**.

Parameters

fh	File handle.
buf	User's memory buffer where data should be stored.
count	Number of *datatype*-sized chunks of data to read.
datatype	MPI datatype specifying data layout in *buf*.
request	Returned request handle.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* or *request* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_READ_CONVERSION	Error in user-defined read conversion function.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL.

See Also

MPI_File_read, MPI_File_read_all_begin.

Notes

Noncollective.

6.1.2.12 MPI_File_iwrite

Purpose

Write a file at the offset specified by the file's individual file pointer, noncollectively and without blocking.

Synopsis

```
#include <mpio.h>

int
MPI_File_iwrite(
        MPI_File            fh,             /* IN / OUT */
        void *              buf,            /* IN    */
        int                 count,          /* IN    */
        MPI_Datatype        datatype,       /* IN    */
        MPI_Request *       request         /* OUT   */
);
```

Description

MPI_File_iwrite initiates a write into the file associated with *fh* at the offset specified by the current individual file pointer position for a total number of *count* data items having *datatype* type from the user's buffer *buf*. The offset is in etype units relative to the current view. The data is written into those parts of the file specified by the current view. Data is read from *buf* according to the pattern specified by *datatype*.

The individual file pointer is incremented by the amount of data requested (not necessarily the amount actually written), provided the write request was initiated without error.

A request handle *request* is returned, which can be used later as the argument to **MPI_Test** or **MPI_Wait** (or any of their variants) to query the status of the write or wait for its completion. The number of datatype elements actually written or error information regarding the completion of the write is returned through the *status* argument to **MPI_Test** or **MPI_Wait**.

Parameters

fh	File handle.
buf	User's buffer where data should be read.
count	Number of *datatype*-sized chunks of data to write.
datatype	MPI datatype specifying data layout in *buf*.
request	Returned request handle.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* or *request* argument is NULL.

MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NO_SPACE	Insufficient free space on the HPSS file system to create, write, or preallocate a file.
MPI_ERR_TOO_FRAGMENTED	Write operation resulted in excessive file fragmentation.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL.
MPI_ERR_WRITE_CONVERSION	Error in user-defined write conversion function.

See Also

MPI_File_write, MPI_File_write_all_begin.

Notes

Noncollective.

6.1.2.13 MPI_File_seek

Purpose

Set an individual file pointer of a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_seek(
        MPI_File      fh,          /* IN/OUT */
        MPI_Offset    offset,      /* IN     */
        int           whence       /* IN     */
);
```

Description

MPI_File_seek updates the individual file pointer associated with *fh* according to *whence* and *offset* for one process. The *offset* can be negative, which allows seeking backwards.

It is erroneous to seek to a negative position in the view.

The end of the file is defined to be the position of the next elementary data item, relative to the current view, following the last whole elementary data item accessible. When seeking relative to end of file, users should be aware that the MPI-2 definition of this differs from the usual definition

of EOF for sequentially-accessed files. Each process may have a different view for a given open file, depending on their filetypes. A file may contain data located beyond an end of file for one process that is visible only to other processes.

Parameters

fh	File handle.
offset	Number of etypes to add or subtract from the position specified by *whence*.
whence	Flag specifying one of three locations in the file: MPI_SEEK_SET - the pointer is set to *offset*. MPI_SEEK_CUR - the pointer is set to the current pointer position plus *offset*. MPI_SEEK_END - the pointer is set to the end of file plus *offset*.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_FILE	Invalid file handle.
MPI_ERR_OFFSET	Invalid *offset* argument.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL.
MPI_ERR_WHENCE	Invalid *whence* argument.

See Also

MPI_File_get_position, MPI_File_get_byte_offset, MPI_File_set_view.

Notes

Noncollective.

6.1.2.14 MPI_File_get_position

Purpose

Get the current position of an individual file pointer.

Synopsis

```
#include <mpio.h>

int
MPI_File_get_position(
        MPI_File        fh,             /* IN  */
        MPI_Offset *    offset          /* OUT */
```

```
);
```

Description

MPI_File_get_position returns in *offset* the current position of the individual file pointer associated with *fh*, where the offset is returned in etype units relative to the current view.

Parameters

fh	File handle.
offset	Returned *offset* in etypes.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*offset* argument is NULL.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL.

See Also

MPI_File_seek, MPI_File_set_view, MPI_File_get_byte_offset.

Notes

Noncollective.

6.1.2.15 MPI_File_get_byte_offset

Purpose

Get the current position of an individual file pointer.

Synopsis

```
#include <mpio.h>

int
MPI_File_get_byte_offset(
        MPI_File        fh,         /* IN  */
        MPI_Offset      offset,     /* IN  */
        MPI_Offset *    disp        /* OUT */
);
```

Description

MPI_File_get_byte_offset converts a view-relative *offset* for the file associated with *fh* into an absolute byte position, returned in *disp*.

Parameters

fh	File handle.

offset	Offset in etypes to convert.
disp	Returned offset in bytes.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*disp* argument is NULL.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL.

See Also

MPI_File_seek, MPI_File_set_view, MPI_File_get_position.

Notes

Noncollective.

6.1.2.16 MPI_File_read_shared

Purpose

Read a file at the offset specified by the file's shared file pointer, noncollectively.

Synopsis

```
#include <mpio.h>

int
MPI_File_read_shared(
        MPI_File        fh,         /* IN / OUT */
        void *          buf,        /* OUT  */
        int             count,      /* IN   */
        MPI_Datatype    datatype,   /* IN   */
        MPI_Status *    status      /* OUT  */
);
```

Description

MPI_File_read_shared attempts to read from the file associated with *fh* at the offset specified by the current shared file pointer position for a total number of *count* data items having *datatype* type into the user's buffer *buf*. The offset is in etype units relative to the current view. The data is read from those parts of the file specified by the current view. Data is stored into *buf* according to the pattern specified by *datatype*. The number of datatype elements actually read is returned in *status*. All other fields of *status* are undefined.

The shared file pointer is incremented by the amount of data requested (not necessarily the amount actually read), provided the read request was initiated without error.

It is erroneous to call this routine if not all processes use the same file view.

Parameters

fh	File handle.
buf	User's memory buffer where data should be stored.
count	Number of *datatype*-sized chunks of data to read.
datatype	MPI datatype specifying data layout in *buf*.
status	Returned MPI_Status object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* or *status* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NOT_SAME_VIEW	Shared file pointer operation requires all processes have the same view.
MPI_ERR_READ_CONVERSION	Error in user-defined read conversion function.

See Also

MPI_File_iread_shared, MPI_File_read_ordered, MPI_File_read_ordered_begin.

Notes

Noncollective.

6.1.2.17 MPI_File_write_shared

Purpose

Write a file at the offset specified by the file's shared file pointer, noncollectively.

Synopsis

```
#include <mpio.h>

int
```

```
MPI_File_write_shared(
        MPI_File        fh,         /* IN / OUT */
        void *          buf,        /* IN    */
        int             count,      /* IN    */
        MPI_Datatype    datatype,   /* IN    */
        MPI_Status *    status      /* OUT   */
);
```

Description

MPI_File_write_shared attempts to write into the file associated with *fh* at the offset specified by the current shared file pointer position for a total number of *count* data items having *datatype* type from the user's buffer *buf*. The offset is in etype units relative to the current view. The data is written into those parts of the file specified by the current view. Data is read from *buf* according to the pattern specified by *datatype*. The number of datatype elements actually written is returned in *status*. All other fields of *status* are undefined.

The shared file pointer is incremented by the amount of data requested (not necessarily the amount actually written), provided the write request was initiated without error.

It is erroneous to call this routine if not all processes use the same file view.

Parameters

fh	File handle.
buf	User's memory buffer where data should be read.
count	Number of *datatype*-sized chunks of data to write.
datatype	MPI datatype specifying data layout in *buf*.
status	Returned MPI_Status object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* or *status* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NO_SPACE	Insufficient free space on the HPSS file system to create, write,

	or preallocate a file.
MPI_ERR_NOT_SAME_VIEW	Shared file pointer operation requires all processes have the same view.
MPI_ERR_TOO_FRAGMENTED	Write operation resulted in excessive file fragmentation.
MPI_ERR_WRITE_CONVERSION	Error in user-defined write conversion function.

See Also

MPI_File_Iwrite_shared, MPI_File_write_ordered, MPI_File_write_ordered_begin.

Notes

Noncollective.

6.1.2.18 MPI_File_iread_shared

Purpose

Read a file at the offset specified by the file's shared file pointer, noncollectively and without blocking.

Synopsis

```
#include <mpio.h>

int
MPI_File_iread_shared(
        MPI_File          fh,          /* IN / OUT */
        void *            buf,         /* OUT    */
        int               count,       /* IN     */
        MPI_Datatype      datatype,    /* IN     */
        MPI_Request *     request      /* OUT    */
);
```

Description

MPI_File_iread_shared initiates a read from the file associated with *fh* at the offset specified by the current shared file pointer position for a total number of *count* data items having *datatype* type into the user's buffer *buf*. The offset is in etype units relative to the current view. The data is read from those parts of the file specified by the current view. Data is stored into *buf* according to the pattern specified by *datatype*

The shared file pointer is incremented by the amount of data requested (not necessarily the amount actually read), provided the read request was initiated without error.

A request handle *request* is returned, which can be used later as the argument to **MPI_Test** or **MPI_Wait** (or any of their variants) to query the status of the read or wait for its completion. The number of datatype elements actually read or error information regarding the completion of the read is returned through the *status* argument to **MPI_Test** or **MPI_Wait**.

It is erroneous to call this routine if not all processes use the same file view.

Parameters

fh	File handle.
buf	User's memory buffer where data should be stored.
count	Number of *datatype*-sized chunks of data to read.
datatype	MPI datatype specifying data layout in *buf*.
request	Returned request handle.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* or *request* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NOT_SAME_VIEW	Shared file pointer operation requires all processes have the same view.
MPI_ERR_READ_CONVERSION	Error in user-defined read conversion function.

See Also

MPI_File_read_shared, MPI_File_read_ordered, MPI_File_read_ordered_begin.

Notes

Noncollective.

6.1.2.19 MPI_File_iwrite_shared

Purpose

Write a file at the offset specified by the file's shared file pointer, noncollectively and without blocking.

Synopsis

```
#include <mpio.h>

int
```

```
MPI_File_iwrite_shared(
        MPI_File            fh,             /* IN / OUT */
        void *              buf,            /* IN    */
        int                 count,          /* IN    */
        MPI_Datatype        datatype,       /* IN    */
        MPI_Request *       request         /* OUT   */
);
```

Description

MPI_File_iwrite_shared initiates a write into the file associated with *fh* at the offset specified by the current shared file pointer position for a total number of *count* data items having *datatype* type from the user's buffer *buf*. The offset is in etype units relative to the current view. The data is written into those parts of the file specified by the current view. Data is read from *buf* according to the pattern specified by *datatype*.

The shared file pointer is incremented by the amount of data requested (not necessarily the amount actually written), provided the write request was initiated without error.

A request handle *request* is returned, which can be used later as the argument to **MPI_Test** or **MPI_Wait** (or any of their variants) to query the status of the write or wait for its completion. The number of datatype elements actually written or error information regarding the completion of the write is returned through the *status* argument to **MPI_Test** or **MPI_Wait**.

It is erroneous to call this routine if not all processes use the same file view.

Parameters

fh	File handle.
buf	User's memory buffer where data should be read.
count	Number of *datatype*-sized chunks of data to write.
datatype	MPI datatype specifying data layout in *buf*.
request	Returned request handle.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* or *request* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.

MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NO_SPACE	Insufficient free space on the HPSS file system to create, write, or preallocate a file.
MPI_ERR_NOT_SAME_VIEW	Shared file pointer operation requires all processes have the same view.
MPI_ERR_TOO_FRAGMENTED	Write operation resulted in excessive file fragmentation.
MPI_ERR_WRITE_CONVERSION	Error in user-defined write conversion function.

See Also

MPI_File_write_shared, MPI_File_write_ordered, MPI_File_write_ordered_begin.

Notes

Noncollective.

6.1.2.20 MPI_File_read_ordered

Purpose

Read a file at the offset specified by the file's shared file pointer, collectively.

Synopsis

```
#include <mpio.h>

int
MPI_File_read_ordered(
        MPI_File        fh,         /* IN/OUT */
        void *          buf,        /* OUT    */
        int             count,      /* IN     */
        MPI_Datatype    datatype,   /* IN     */
        MPI_Status *    status      /* OUT    */
);
```

Description

MPI_File_read_ordered attempts to read collectively from the file associated with *fh*. For each process, the offset in the file at which data is read is the position at which the shared file pointer would be after all processes whose ranks within the group are less than that of this process have read their data. Each process reads for a total number of *count* data items having *datatype* type into the user's buffer *buf*. The data is read from those parts of the file specified by the current view. Data is stored into *buf* according to the pattern specified by *datatype*.

Each process may pass different argument values for *datatype* and *count*. After all the processes have issued their respective calls, each process attempts to read.

The shared file pointer is updated by the amounts of data requested by all processes of the group. For each process, the number of datatype elements actually read is returned in *status*.

It is erroneous to call this routine if not all processes use the same file view.

Parameters

fh	File handle.
buf	User's memory buffer where data should be stored.
count	Number of *datatype*-sized chunks of data to read.
datatype	MPI datatype specifying data layout in *buf*.
status	Returned MPI_Status object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* or *status* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NOT_SAME_VIEW	Shared file pointer operation requires all processes have the same view.
MPI_ERR_PENDING_RW	A collective operation is already pending for this file.
MPI_ERR_READ_CONVERSION	Error in user-defined read conversion function.

See Also

MPI_File_read_shared, MPI_File_iread_shared, MPI_File_read_ordered_begin.

Notes

Collective; synchronizing.

6.1.2.21 MPI_File_write_ordered

Purpose

Write a file at the offset specified by the file's shared file pointer, collectively.

Synopsis

#include <mpio.h>

```
int
MPI_File_write_ordered(
        MPI_File        fh,             /* IN/OUT */
        void *          buf,            /* IN     */
        int             count,          /* IN     */
        MPI_Datatype datatype,          /* IN     */
        MPI_Status *status              /* OUT    */
);
```

Description

MPI_File_write_ordered attempts to write collectively into the file associated with *fh*. For each process, the location in the file at which data is written is the position at which the shared file pointer would be after all processes whose ranks within the group are less than that of this process have written their data. Each process writes for a total number of *count* data items having *datatype* type from the user's buffer *buf*. The data is written into those parts of the file specified by the current view. Data is read from *buf* according to the pattern specified by *datatype*.

Each process may pass different argument values for *datatype* and *count*. After all the processes have issued their respective calls, each process attempts to write.

The shared file pointer is updated by the amounts of data requested by all processes of the group. For each process, the number of datatype elements actually written is returned in *status*.

It is erroneous to call this routine if not all processes use the same file view.

Parameters

fh	File handle.
buf	User's memory buffer where data should be read.
count	Number of *datatype*-sized chunks of data to write.
datatype	MPI datatype specifying data layout in *buf*.
status	Returned MPI_Status object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* or *status* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	

	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NO_SPACE	Insufficient free space on the HPSS file system to create, write, or preallocate a file.
MPI_ERR_NOT_SAME_VIEW	Shared file pointer operation requires all processes have the same view.
MPI_ERR_PENDING_RW	A collective operation is already pending for this file.
MPI_ERR_TOO_FRAGMENTED	Write operation resulted in excessive file fragmentation.
MPI_ERR_WRITE_CONVERSION	Error in user-defined write conversion function.

See Also

MPI_File_write_shared, MPI_File_iwrite_shared, MPI_File_write_ordered_begin.

Notes

Collective; synchronizing.

6.1.2.22 MPI_File_seek_shared

Purpose

Set the current shared file pointer of a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_seek_shared(
        MPI_File    fh          /* IN / OUT */
        int         offset,     /* IN    */
        MPI_Whence  whence      /* IN    */
);
```

Description

MPI_File_seek_shared updates the shared file pointer associated with *fh* according to *whence* and *offset* for all processes in the file's communicator group. All participating processes must provide the same values for *offset* and *whence*. The *offset* can be negative, which allows seeking backwards. It is erroneous to seek to a negative position in the view.

The end of the file is defined to be the position of the next elementary data item, relative to the current view, following the last whole elementary data item accessible. Since shared file pointers are well-defined only when all the processes use the same view, this position will be the same for all processes.

It is erroneous to call this routine if not all processes use the same file view.

Parameters

fh	File handle.
offset	Number of etypes to add or subtract from the location specified by whence.
whence	Flag specifying one of three locations in the file: MPI_SEEK_SET - the pointer is set to *offset*. MPI_SEEK_CUR - the pointer is set to the current pointer position plus *offset*. MPI_SEEK_END - the pointer is set to the end of file plus *offset*.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NOT_SAME	*offset* and *whence* arguments do not match across processes.
MPI_ERR_NOT_SAME_VIEW	Shared file pointer operation requires all processes have the same view.
MPI_ERR_OFFSET	Invalid *offset* argument.
MPI_ERR_WHENCE	Invalid *whence* argument.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL..

See Also

MPI_File_get_position_shared.

Notes

Collective; synchronizing.

6.1.2.23 MPI_File_get_position_shared

Purpose

Get the current position of a shared file pointer.

Synopsis

```
#include <mpio.h>

int
MPI_File_get_position_shared(
```

```
        MPI_File        fh,             /* IN  */
        MPI_Offset *    offset          /* OUT */
);
```

Description

MPI_File_get_position_shared returns in *offset* the current position of the shared file pointer associated with *fh*, where the offset is returned in etype units relative to the current view.

It is erroneous to call this routine if not all processes use the same file view.

Parameters

fh	File handle.
offset	Returned offset in etypes.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*offset* argument is NULL.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NOT_SAME_VIEW	Shared file pointer operation requires all processes have the same view.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL.

See Also

MPI_File_seek_shared, MPI_File_set_view, MPI_File_get_byte_offset.

Notes

Noncollective.

6.1.2.24 MPI_File_read_at_all_begin

Purpose

Inititiate a split-collective, explicit offset read of a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_read_at_all_begin(
        MPI_File        fh,             /* IN  */
        MPI_Offset      offset,         /* IN  */
        void *          buf,            /* OUT */
        int             count,          /* IN  */
        MPI_Datatype    datatype        /* IN  */
```

);

Description

MPI_File_read_at_all_begin initiates a split-collective read from the file associated with *fh*. Each process reads at a specified *offset* for a total number of *count* data items having *datatype* type into the user's buffer *buf*. The *offset* is in etype units relative to each process's current view. The data is read from those parts of the file specified by each process's current view. Data is stored into *buf* according to the pattern specified by *datatype*.

Each process may pass different argument values for *offset*, *datatype*, and *count*. After all the processes have issued their respective calls, each process attempts to read.

For each process, the number of datatype elements actually read is returned by **MPI_File_read_at_all_end**.

Parameters

fh	File handle.
offset	File offset in etypes at which to begin reading.
buf	User's memory buffer where data should be stored.
count	Number of *datatype*-sized chunks of data to read.
datatype	MPI datatype specifying data layout in *buf*.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_PENDING_RW	A collective operation is already pending for this *fh*.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL.

See Also

MPI_File_read_at, MPI_File_read_at_all_end.

Notes

Collective; synchronizing.

6.1.2.25 MPI_File_read_at_all_end

Purpose

Complete a split-collective, explicit offset read of a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_read_at_all_end(
        MPI_File        fh,             /* IN  */
        void *          buf,            /* OUT */
        MPI_Status *    status          /* OUT */
);
```

Description

MPI_File_read_at_all_end completes a split-collective read initiated by **MPI_File_read_at_all_begin**.

The amount of data actually read is returned to each process in *status*.

Parameters

fh	File handle.
buf	User's memory buffer where data should be stored.
status	Returned MPI_Status object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*buf* or *status* argument is NULL.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NOT_SAME	No matching begin operation for this *fh*.
MPI_ERR_READ_CONVERSION	Error in user-defined read conversion function.

See Also

MPI_File_read_at_all_begin.

Notes

Collective; nonsynchronizing.

6.1.2.26 MPI_File_write_at_all_begin

Purpose

Initiate a split-collective, explicit offset write to a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_write_at_all_begin(
        MPI_File        fh,         /* IN / OUT */
        MPI_Offset      offset,     /* IN    */
        void *          buf,        /* IN    */
        int             count,      /* IN    */
        MPI_Datatype    datatype    /* IN    */
);
```

Description

MPI_File_write_at_all_begin initiates a split-collective write into the file associated with *fh*. Each process writes at a specified *offset* for a total number of *count* data items having *datatype* type from the user's buffer *buf*. The *offset* is in etype units relative to each process's current view. The data is written to those parts of the file specified by each process's current view. Data is read from *buf* according to the pattern specified by *datatype*.

Each process may pass different argument values for *offset*, *datatype*, and *count*. After all the processes have issued their respective calls, each process attempts to write.

For each process, the number of datatype elements actually written is returned by **MPI_File_write_at_all_end**.

Parameters

fh	File handle.
offset	File offset in etypes at which to begin writing.
buf	User's memory buffer where data should be read.
count	Number of *datatype*-sized chunks of data to write.
datatype	MPI datatype specifying data layout in *buf*.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.

MPI_ERR_ENOMEM — Unable to allocate program space.

MPI_ERR_EXTENT_CONVERSION — Error in user-defined extent function.

MPI_ERR_FILE — Invalid file handle.

MPI_ERR_PENDING_RW — A collective operation is already pending for this *fh*.

MPI_ERR_UNSUPPORTED_OPERATION — Operation not allowed with MPI_MODE_SEQUENTIAL.

See Also

MPI_File_write_at, MPI_File_write_at_all_end.

Notes

Collective; synchronizing.

6.1.2.27 MPI_File_write_at_all_end

Purpose

Complete a split-collective, explicit offset write to a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_write_at_all_end(
        MPI_File        fh,             /* IN/OUT */
        void *          buf,            /* IN     */
        MPI_Status *    status          /* OUT    */
);
```

Description

MPI_File_write_at_all_end completes a split-collective write initiated by **MPI_File_write_at_all_begin**.

The amount of data actually written is returned to each process in *status*.

Parameters

fh — File handle.

buf — User's memory buffer where data should be read.

status — Returned MPI_Status object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*buf* or *status* argument is NULL.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NO_SPACE	Insufficient free space on the HPSS file system to create, write, or preallocate a file.
MPI_ERR_NOT_SAME	No matching begin operation for this *fh*.
MPI_ERR_TOO_FRAGMENTED	Write operation resulted in excessive file fragmentation.
MPI_ERR_WRITE_CONVERSION	Error in user-defined write conversion function.

See Also

MPI_File_write_at_all_begin.

Notes

Collective; nonsynchronizing.

6.1.2.28 MPI_File_read_all_begin

Purpose

Initiate a split-collective, individual file pointer read of a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_read_all_begin(
        MPI_File        fh,         /* IN/OUT */
        void *          buf,        /* OUT    */
        int             count,      /* IN     */
        MPI_Datatype    datatype    /* IN     */
);
```

Description

MPI_File_read_all_begin initiates a split-collective read from the file associated with *fh*. Each process reads at the offset specified by its current individual file pointer position for a total number of *count* data items having *datatype* type into the user's buffer *buf*. The offset is in etype units relative to each process's current view. The data is read from those parts of the file specified by each process's current view. Data is stored into *buf* according to the pattern specified by *datatype*.

Each process may pass different argument values for *datatype* and *count*. After all the processes have issued their respective calls, each process attempts to read.

Each individual file pointer is incremented by the amount of data requested (not necessarily the amount actually read), provided the read request was initiated without error.

For each process, the number of datatype elements actually read is returned by

MPI_File_read_all_end.

Parameters

fh	File handle.
buf	User's memory buffer where data should be stored.
count	Number of *datatype*-sized chunks of data to read.
datatype	MPI datatype specifying data layout in *buf*.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_PENDING_RW	A collective operation is already pending for this *fh*.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL.

See Also

MPI_File_read, MPI_File_read_all_end.

Notes

Collective; synchronizing.

6.1.2.29 MPI_File_read_all_end

Purpose

Complete a split-collective, individual file pointer read of a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_read_all_end(
```

```
        MPI_File        fh,             /* IN/OUT */
        void *          buf,            /* OUT    */
        MPI_Status *    status    /* OUT    */
);
```

Description

MPI_File_read_all_end completes a split-collective read initiated by **MPI_File_read_all_begin**.

The amount of data actually read is returned to each process in *status*.

Parameters

fh	File handle.
buf	User's memory buffer where data should be stored.
status	Returned MPI_Status object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*buf* or *status* argument is NULL.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NOT_SAME	No matching begin operation for this *fh*.
MPI_ERR_READ_CONVERSION	Error in user-defined read conversion function.

See Also

MPI_File_read_all_begin.

Notes

Collective; nonsynchronizing.

6.1.2.30 MPI_File_write_all_begin

Purpose

Initiate a split-collective, individual file pointer write to a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_write_all_begin(
        MPI_File        fh,             /* IN/OUT */
        void *          buf,            /* IN     */
        int             count,          /* IN     */
```

```
        MPI_Datatype        datatype        /* IN    */
);
```

Description

MPI_File_write_all_begin initiates a split-collective write into the file associated with *fh*. Each process writes at the offset specified by its current individual file pointer position for a total number of *count* data items having *datatype* type from the user's buffer *buf*. The offset is in etype units relative to each process's current view. The data is written into those parts of the file specified by each process's current view. Data is read from *buf* according to the pattern specified by *datatype*.

Each process may pass different argument values for *datatype* and *count*. After all the processes have issued their respective calls, each process attempts to write.

Each individual file pointer is incremented by the amount of data requested (not necessarily the amount actually written), provided the write request was initiated without error.

For each process, the number of datatype elements actually written is returned by **MPI_File_write_all_end**.

Parameters

fh	File handle.
buf	User's memory buffer where data should be read.
count	Number of *datatype*-sized chunks of data to write.
datatype	MPI datatype specifying data layout in *buf*.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_PENDING_RW	A collective operation is already pending for this *fh*.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL.

See Also

MPI_File_write, MPI_File_write_all, MPI_File_write_all_end.

Notes

Collective; synchronizing.

6.1.2.31 MPI_File_write_all_end

Purpose

Complete a split-collective, individual file pointer write to a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_write_all_end(
        MPI_File        fh,             /* IN / OUT */
        void *          buf,            /* IN    */
        MPI_Status *    status          /* OUT   */
);
```

Description

MPI_File_write_all_end completes a split-collective write initiated by **MPI_File_write_all_begin**.

The amount of data actually written is returned to each process in *status*.

Parameters

fh	File handle.
buf	User's memory buffer where data should be read.
status	Returned MPI_Status object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*buf* or *status* argument is NULL.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NO_SPACE	Insufficient free space on the HPSS file system to create, write, or preallocate a file.
MPI_ERR_NOT_SAME	No matching begin operation for this *fh*.
MPI_ERR_TOO_FRAGMENTED	Write operation resulted in excessive file fragmentation.

MPI_ERR_WRITE_CONVERSION Error in user-defined write conversion function.

See Also

MPI_File_write_all_begin.

Notes

Collective; nonsynchronizing.

6.1.2.32 MPI_File_read_ordered_begin

Purpose

Initiate a split-collective, shared file pointer read of a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_read_ordered_begin(
        MPI_File        fh,         /* IN/OUT */
        void *          buf,        /* OUT    */
        int             count,      /* IN     */
        MPI_Datatype    datatype    /* IN     */
);
```

Description

MPI_File_read_ordered_begin initiates a split-collective read from the file associated with *fh*. For each process, the offset in the file at which data is read is the position at which the shared file pointer would be after all processes whose ranks within the group are less than that of this process have read their data. Each process reads for a total number of *count* data items having *datatype* type into the user's buffer *buf*. The data is read from those parts of the file specified by the current view. Data is stored into *buf* according to the pattern specified by *datatype*.

Each process may pass different argument values for *datatype* and *count*. After all the processes have issued their respective calls, each process attempts to read.

The shared file pointer is updated by the amounts of data requested by all processes of the group.

For each process, the number of datatype elements actually written is returned by **MPI_File_read_ordered_end**.

It is erroneous to call this routine if not all processes use the same file view.

Parameters

fh	File handle.
buf	User's memory buffer where data should be stored.
count	Number of *datatype*-sized chunks of data to read.

datatype	MPI datatype specifying data layout in *buf*.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NOT_SAME_VIEW	Shared file pointer operation requires all processes have the same view.
MPI_ERR_PENDING_RW	A collective operation is already pending for this *fh*.

See Also

MPI_File_read_ordered, MPI_File_read_ordered_end.

Notes

Collective; synchronizing.

6.1.2.33 MPI_File_read_ordered_end

Purpose

Complete a split-collective, shared file pointer read of a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_read_ordered_end(
        MPI_File      fh,          /* IN/OUT */
        void *        buf,         /* OUT    */
        MPI_Status *  status       /* OUT    */
);
```

Description

MPI_File_read_ordered_end completes a split-collective read initiated by **MPI_File_read_ordered_begin**.

The amount of data actually read is returned to each process in *status*.

Parameters

fh	File handle.
buf	User's memory buffer where data should be read.
status	Returned MPI_Status object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*buf* or *status* argument is NULL.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NOT_SAME	No matching begin operation for this *fh*.
MPI_ERR_READ_CONVERSION	Error in user-defined read conversion function.

See Also

MPI_File_read_ordered_begin.

Notes

Collective; nonsynchronizing.

6.1.2.34 MPI_File_write_ordered_begin

Purpose

Initiate a split-collective, shared file pointer write to a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_write_ordered_begin(
        MPI_File        fh,         /* IN/OUT */
        void *          buf,        /* IN     */
        int             count,      /* IN     */
        MPI_Datatype    datatype    /* IN     */
);
```

Description

MPI_File_write_ordered_begin initiates a split-collective write into the file associated with *fh*. For each process, the location in the file at which data is written is the position at which the shared file pointer would be after all processes whose ranks within the group are less than that of this process

have written their data. Each process writes for a total number of *count* data items having *datatype* type from the user's buffer *buf*. The data is written into those parts of the file specified by the current view. Data is read from *buf* according to the pattern specified by *datatype*.

Each process may pass different argument values for *datatype* and *count*. After all the processes have issued their respective calls, each process attempts to write.

The shared file pointer is updated by the amounts of data requested by all processes of the group.

For each process, the number of datatype elements actually written is returned by **MPI_File_write_ordered_end**.

It is erroneous to call this routine if not all processes use the same file view.

Parameters

fh	File handle.
buf	User's memory buffer where data should be read.
count	Number of *datatype*-sized chunks of data to write.
datatype	MPI datatype specifying data layout in *buf*.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ACCESS	Access permission is denied to the file.
MPI_ERR_ARG	*buf* argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer *datatype*.
MPI_ERR_COUNT	Invalid *count* argument.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NOT_SAME_VIEW	Shared file pointer operation requires all processes have the same view.
MPI_ERR_PENDING_RW	A collective operation is already pending for this *fh*.

See Also

MPI_File_write_ordered, MPI_File_write_ordered_end.

Notes

Collective; synchronizing.

6.1.2.35 MPI_File_write_ordered_end

Purpose

Complete a split-collective, shared file pointer write to a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_write_ordered_end(
        MPI_File        fh,             /* IN / OUT */
        void *          buf,            /* IN     */
        MPI_Status *    status          /* OUT    */
);
```

Description

MPI_File_write_ordered_end completes a split-collective write initiated by **MPI_File_write_ordered_begin**.

The amount of data actually written is returned to each process in *status*.

Parameters

fh	File handle.
buf	User's memory buffer where data should be read.
status	Returned MPI_Status object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*buf* or *status* argument is NULL.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NO_SPACE	Insufficient free space on the HPSS file system to create, write, or preallocate a file.
MPI_ERR_NOT_SAME	No matching begin operation for this *fh*.
MPI_ERR_TOO_FRAGMENTED	Write operation resulted in excessive file fragmentation.
MPI_ERR_WRITE_CONVERSION	Error in user-defined write conversion function.

See Also

MPI_File_write_ordered_begin.

Notes

Collective; nonsynchronizing.

6.1.3 File Interoperability

File interoperability is the ability to read the information previously written to a file. For homogeneous environments, applications achieve interoperability by using the host's native data representation in files. For heterogeneous environments, applications must agree on a data representation for files that will allow file sharing.

MPI-IO extends file interoperability by allowing applications to associate a user-defined data representation or any of three predefined data representations with an open file, to be used when reading or writing the file. Note that the data representation is a property of an open file; it is not stored in the file. The application specifies a data representation as an argument to **MPI_File_set_view**.

The three predefined data representations supported are:

"native" This file representation is the same as that used in memory, so no conversions are needed. This is the default data representation used.

"internal" This file representation is currently the same as "native" for this implementation.

"external32" This file representation converts all native/internal data to a canonical representation for each MPI predefined type. In this representation, all floating point values are big-endian IEEE format; all integral values are two's complement big-endian format; and all data is byte-aligned, regardless of type. The byte sizes of data in this representation are:

MPI_PACKED1

MPI_BYTE1

MPI_CHAR1

MPI_UNSIGNED_CHAR1

MPI_SIGNED_CHAR1

MPI_WCHAR2

MPI_SHORT2

MPI_UNSIGNED_SHORT2

MPI_INT4

MPI_UNSIGNED4

MPI_LONG4

MPI_UNSIGNED_LONG4

MPI_FLOAT4

MPI_DOUBLE8

MPI_LONG_DOUBLE16

MPI_CHARACTER1

MPI_LOGICAL4

MPI_INTEGER4

MPI_REAL4

MPI_DOUBLE_PRECISION8

MPI_COMPLEX2*4

MPI_DOUBLE_COMPLEX2*8

User-defined data representations and external32 data representations require extra buffering of the data being transferred to and from a file, which will impact performance of the system.

User-defined data representations require that the user provide conversion and extent callback functions so that MPI-IO can perform the necessary conversions when reading and writing the file. MPI_Register_datarep is used to associate the callback functions with a data representation identifier.

The conversion callback functions have the following syntactic specification:

```
typedef int
MPI_Datarep_conversion_function(
     void *           userbuf,
     MPI_Datatype     datatype,
     int              count,
     void *           filebuf,
     MPI_Offset       position,
     void *           extra_state
);
```

MPI-IO will automatically invoke the associated conversion functions for read and write operations on a file with a user-defined data representation. Each callback function must convert between file data representation and native data representation. MPI-IO allocates space for a *filebuf*, and invokes the appropriate callback to fill or empty the *filebuf* as many times as necessary to complete the read or write, transferring data to or from the *userbuf*. Each callback routine must be able to convert *count* data items (predefined MPI types) starting at the data item at *position* in the *datatype*. If the size of *datatype* is less than the size of the *count* data items, the conversion function must treat *datatype* as being contiguously tiled over the *userbuf*. The *extra_state* argument can be used to maintain state information between invocations of the callback routines.

If an application defines and uses its own data representation, it must also take care in constructing etypes and filetypes for portability and scalability. MPI-IO will use the extent callback function for predefined types in order to scale all portable etypes and filetypes; nonportable etypes and filetypes are not scaled. The application may use **MPI_File_get_type_extent** in calculating file offsets, such as for seeks. The extent callback function has the following syntactic specification:

```
typedef int
MPI_Datarep_extent_function(
     MPI_Datatype     datatype,
     MPI_Aint *       file_extent,
```

```
    void *            extra_state
);
```

The extent callback function returns in *file_extent* the number of bytes needed to store *datatype* in the file. The *extra_state* argument can be used to maintain state information between invocations of the callback routines. MPI-IO will only invoke this callback function for predefined datatypes employed by the user.

6.1.3.1 MPI_File_get_type_extent

Purpose

Get the extent of a datatype in the data representation mode specified for a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_get_type_extent(
        MPI_File        fh,             /* IN  */
        MPI_Datatype    datatype,       /* IN  */
        MPI_Aint *      extent          /* OUT */
);
```

Description

MPI_File_get_type_extent returns in *extent* the extent of *datatype* in the data representation of the current file view for the file *fh*.

Parameters

fh	File handle.
datatype	MPI datatype for which to determine extent.
extent	Returned extent.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*extent* argument is NULL or *datatype* argument is MPI_DATATYPE_NULL.
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.

See Also

MPI_File_set_view, MPI_Register_datarep.

Notes

Noncollective.

6.1.3.2 MPI_Register_datarep

Purpose

Register a data representation identifer.

Synopsis

```
#include <mpio.h>

int
MPI_Register_datarep(
        char *                                  datarep,                /* IN */
        MPI_Datarep_conversion_function *       read_conversion_fn,     /* IN */
        MPI_Datarep_conversion_function *       write_conversion_fn,    /* IN */
        MPI_Datarep_extent_function *           dtype_file_extent_fn,   /* IN */
        void *                                  extra_state             /* IN */
);
```

Description

MPI_Register_datarep associates *read_conversion_fn*, *write_conversion_fn*, and *dtype_file_extent_fn* with the data representation identifier *datarep*. *datarep* can then be used as an argument to **MPI_File_set_view**. This is a local operation and only registers *datarep* for the calling MPI process.

The length of a data representation string is limited to the value MPI_MAX_DATAREP_STRING.

Parameters

datarep	String literal to be used to identify the data representation.
read_conversion_fn	Callback function to convert data read from a file to the native representation.
write_conversion_fn	Callback function to convert data written to a file from the native representation.
dtype_file_extent_fn	Callback function to determine the extent of a predefined type in *datarep*.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	Invalid *datarep* argument.
MPI_ERR_DUP_DATAREP	*datarep* duplicates a registered or predefined data representation identifier.
MPI_ERR_ENOMEM	Unable to allocate program space.

See Also

MPI_File_set_view, MPI_File_get_type_extent.

Notes

Noncollective.

6.1.4 File Consistency

Consistency semantics define the outcome of multiple accesses to a single file. MPI-2 describes

three levels of consistency: sequential consistency among all accesses using a single file handle; sequential consistency among all accesses using file handles created from a single collective open with atomic mode enabled, and user-imposed consistency among accesses other than the above. Sequential consistency means the behavior of a set of operations will be as if the operations were performed in some serial order consistent with program order; each access appears atomic, although the exact ordering of accesses is unspecified. User-imposed consistency may be enforced using program order, calls to **MPI_File_sync**, and file atomicity modes.

In MPI-IO, HPSS semantics guarantee atomic sequential consistency for accesses using a single file handle and for accesses using the file handles created from a single collective open, regardless of whether or not atomic mode is enabled. When using file handles created with multiple opens of the same file, the application must impose its own consistency strategies.

Furthermore, for data transfers where the file view dictates that the transfer will be fragmented over the file (i.e., where there will be holes in the file between data that is accessed), HPSS constraints may require breaking a single logical transfer into multiple physical transfers. That is, HPSS imposes a limit on the number of chunks allowed per transfer; if this limit is exceeded, MPI-IO will divide the transfer into smaller transfers. In this case, the use of atomic mode is recommended to guarantee sequential consistency as required.

6.1.4.1 MPI_File_set_atomicity

Purpose

Set the atomicity mode for data access operations on an open file.

Synopsis

```
#include <mpio.h>

int
MPI_File_set_atomicity(
        MPI_File*f*     *h*,            /* IN / OUT */
        int             *flag*          /* IN    */
);
```

Description

MPI_File_set_atomicity sets the atomicity mode for the file *fh* as specified in *flag*. All participating processes must specify the same value for *flag*. If *flag* is true, atomic mode is set; if *flag* is false, nonatomic mode is set.

Parameters

fh	File handle.
flag	Specifies how the system should handle overlapping write requests from different processes. Possible values are: **true** (nonzero) and **false** (zero).

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NOT_SAME	*flag* arguments do not match across processes.

See Also

MPI_File_get_atomicity.

Notes

Collective; synchronizing.

6.1.4.2 MPI_File_get_atomicity

Purpose

Get the current atomicity mode for data accesses on a file.

Synopsis

```
#include <mpio.h>
```

```
int
MPI_File_get_atomicity(
        MPI_File        fh,             /* IN  */
        int *           flag            /* OUT */
);
```

Description

MPI_File_get_atomicity returns the atomicity mode setting for the file designated by *fh*.

Parameters

fh	File handle.
flag	Returned atomicity mode.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*flag* argument is NULL.
MPI_ERR_FILE	Invalid file handle.

See Also

MPI_File_set_atomicity.

Notes

Noncollective.

6.1.4.3 MPI_File_sync

Purpose

Synchronize pending data accesses to a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_sync(
        MPI_File        fh          /* IN/OUT */
);
```

Description

MPI_File_sync causes all previous writes to *fh* by the calling process to be transferred to the storage device. This may be necessary to ensure sequential consistency in certain cases.

For this implementation, **MPI_File_sync** is equivalent to an MPI_Barrier for the group of processes that participated in a file open. HPSS semantics guarantee that writes are always transferred to the storage device.

It is erroneous to call **MPI_File_sync** while uncompleted nonblocking or split collective calls are pending for *fh*.

Parameters

fh	File handle to synchronize.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_PENDING_RW	There are uncompleted read or write operations pending for *fh*.

See Also

None.

Notes

Collective; synchronizing.

6.1.5 Error Handling

By default in MPI all communication errors are fatal. I/O errors are usually less catastrophic, so MPI-2 provides additional error facilities for I/O.

Like communicators, each file handle has an error handler associated with it. The predefined error handler for all file handles is MPI_ERRORS_RETURN. An application may override this by defining its own error handler using **MPI_File_create_errhandler** and **MPI_File_set_errhandler**.

For those APIs for which no file handle is associated (e.g., **MPI_File_delete** or **MPI_File_open** before a file handle has been successfully assigned), there is a default file error handler. This default file error handler is initially MPI_ERRORS_RETURN. The default error handler may be changed by invoking **MPI_File_set_errhandler** with MPI_FILE_NULL as the file handle argument.

The MPI-2 standard defines I/O error *classes*, and implementation-specific error *codes* may be added by an MPI implementation. MPI_Error_class translates error codes into classes so that programs can handle errors portably while allowing them to use implementation-specific information about the error where available. MPI_Error_string can be used to translate an error code into an error message.

MPI-2 adds the following error classes to MPI:

MPI_ERR_FILE	Invalid file handle.
MPI_ERR_NOT_SAME	Collective argument not the same on all processes, or collective routine called in a different order by different processes.
MPI_ERR_AMODE	Error related to the *amode* passed to **MPI_File_open**.

MPI_ERR_UNSUPPORTED_DATAREP
: Unsupported *datarep* passed to **MPI_File_set_view**.

MPI_ERR_UNSUPPORTED_OPERATION
: Unsupported operation, such as seeking on a file which supports sequential access only.

MPI_ERR_NO_SUCH_FILE
: File does not exist.

MPI_ERR_FILE_EXISTS
: File exists.

MPI_ERR_BAD_FILE
: Invalid file name.

MPI_ERR_ACCESS
: Permission denied.

MPI_ERR_NO_SPACE
: Not enough space.

MPI_ERR_QUOTA
: Quota exceeded.

MPI_ERR_READ_ONLY
: Read-only file or file system.

MPI_ERR_FILE_IN_USE
: File operation could not be completed, as the file is currently being used by another process.

MPI_ERR_DUP_DATAREP
: Conversion functions could not be registered because a data representation identifier that was already defined was passed to MPI_Register_datarep.

MPI_ERR_CONVERSION
: An error occurred in a user-supplied data conversion function.

MPI_ERR_IO
: Other I/O error.

MPI_ERR_INFO_KEY
: Invalid key length.

MPI_ERR_INFO_VALUE
: Invalid value length.

MPI_ERR_INFO_NOKEY
: Key not defined.

This MPI-IO implementation adds the following error codes to MPI:

MPI_ERR_EPERM
: Invalid credentials for user.

MPI_ERR_ENOMEM
: Could not allocate core space.

MPI_ERR_ENOTDIR
: Pathname element is not a directory.

MPI_ERR_EISDIR
: Filename is a directory.

MPI_ERR_ENFILE
: Too many open files in system.

MPI_ERR_EMFILE
: File table overflow.

MPI_ERR_ETIMEDOUT
: Operation timed out.

MPI_ERR_ENOCONNECT	Unable to connect to file system.
MPI_ERR_DISPLACEMENT	Invalid displacement.
MPI_ERR_OFFSET	Invalid offset.
MPI_ERR_WHENCE	Invalid seek *whence* argument.
MPI_ERR_HINTS	Invalid hints.
MPI_ERR_FILETYPE	Invalid filetype.
MPI_ERR_ETYPE	Invalid etype.
MPI_ERR_BUFTYPE	Invalid datatype for user buffer.
MPI_ERR_MEMBER	Client process not in file communicator.
MPI_ERR_EXPECTED_OPEN	Expected open operation.
MPI_ERR_EXPECTED_CLOSE	Expected close operation.
MPI_ERR_EXPECTED_SIZE	Expected set_size operation.
MPI_ERR_EXPECTED_VIEW	Expected set_view operation.
MPI_ERR_EXPECTED_SEEK	Expected seek_shared operation.
MPI_ERR_EXPECTED_ATOMICITY	Expected set_atomicity operation.
MPI_ERR_EXPECTED_SYNC	Expected sync operation.
MPI_ERR_DUP_CLIENT	Duplicate client in collective operation.
MPI_ERR_PENDING_RW	Pending RW operation on file.
MPI_ERR_NOT_SAME_VIEW	Shared file pointer requires same view.
MPI_ERR_TOO_FRAGMENTED	File is too fragmented.
MPI_ERR_EXTENT_CONVERSION	Error in datarep extent function.
MPI_ERR_READ_CONVERSION	Error in datarep read conversion.
MPI_ERR_WRITE_CONVERSION	Error in datarep write conversion.
MPI_ERR_NOT_INITIALIZED	MPI-IO is not initialized.
MPI_ERR_ALREADY_INITIALIZED	

MPI-IO is already initialized.

Each error returned by an MPI-IO API falls into a specific class and may be further distinguished by an error code distinct from its MPI error class. MPI-IO returns errors in the MPI-2 error I/O classes as well as in the MPI-1 error classes MPI_ERR_ARG, MPI_ERR_TYPE, MPI_ERR_COUNT, and MPI_ERR_RANK.

Nonblocking APIs (including initiating split-collective APIs) can return errors in two ways: some errors can be detected early and are returned directly as the return value of the API. In this case, no request object (MPI_REQUEST_NULL) is returned and the program should not call **MPI_Test** or **MPI_Wait** (or the completing split-collective API). Other errors are not detected until after the nonblocking API has returned. In this case, the completing function (e.g., **MPI_Wait**) will return a value of MPI_ERR_IN_STATUS, and the error will be reported in the MPI_Status object that is returned.

6.1.5.1 MPI_File_create_errhandler

Purpose

Create an error handler to associate with a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_create_errhandler(
        MPI_File_errhandler_fn *    function,       /* IN  */
        MPI_Errhandler *            errhandler      /* OUT */
);
```

Description

MPI_File_create_errhandler creates a file error handler object from a function having the following type signature:

```
typedef void
MPI_File_errhandler_fn(
        MPI_File *,
        int *,
        ...
);
```

Parameters

function	User-defined error handling function.
errhandler	Returned error handler

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*function* or *errhandler* argument is NULL.
MPI_ERR_ENOMEM	Unable to allocate program space.

See Also

MPI_File_set_errhandler.

Notes

Noncollective.

6.1.5.2 MPI_File_set_errhandler

Purpose

Set the error handler associated with a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_set_errhandler(
        MPI_File            fh,             /* IN / OUT */
        MPI_Errhandler      errhandler      /* IN     */
);
```

Description

MPI_File_set_errhandler associates a new error handler with a file. The error handler must be either a predefined error handler, or a handler created by a call to **MPI_File_create_errhandler**. If the file is MPI_FILE_NULL, the default file error handler is assigned to be *errhandler*.

Parameters

fh	File handle.
errhandler	File error handler to associate with *fh*.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	Invalid *errhandler* argument.
MPI_ERR_FILE	Invalid file handle.

See Also

MPI_File_create_errhandler, MPI_File_get_errhandler.

Notes

Noncollective.

6.1.5.3 MPI_File_get_errhandler

Purpose

Get the error handler associated with a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_get_errhandler(
        MPI_File            fh,             /* IN  */
        MPI_Errhandler *    errhandler      /* OUT */
);
```

Description

MPI_File_get_errhandler returns the error handler currently associated with a file. If the file

handle is MPI_FILE_NULL, the current default file error handler is returned.

Parameters

fh	File handle.
errhandler	Returned error handler.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*errhandler* argument is NULL.
MPI_ERR_FILE	Invalid file handle.

See Also

MPI_File_set_errhandler.

Notes

Noncollective.

6.1.5.4 MPI_File_call_errhandler

Purpose

Invokes the error handler associated with a file.

Synopsis

```
#include <mpio.h>

int
MPI_File_call_errhandler(
        MPI_File    fh,          /* IN */
        int         errorcode    /* IN */
);
```

Description

MPI_File_call_errhandler invokes the error handler currently associated with the file designated by *fh* with the *errorcode* supplied. If *fh* is MPI_FILE_NULL or *errorcode* is MPI_ERR_FILE, the default file error handler is invoked.

Parameters

fh	File handle.
errorcode	Error code to send to error handler.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, the return value depends on the error handler invoked, which may not return at all.

Error Conditions

MPI_ERR_ARG	*fh* is invalid.

See Also

MPI_File_set_errhandler.

Notes

Noncollective.

6.1.6 File Hints

MPI-2 provides a mechanism to supply hints to an implementation at file open time for the purpose of communicating known or desired characteristics of the file, such as access style, file permissions, and striping parameters. These hints are given to the open command in the form of an opaque MPI data object of type MPI_Info. This object contains some number of (key, value) pairs where key and value are both strings.

There are several keys reserved by the MPI-2 standard for use as file hints. An implementation in not required to support all the reserved keys, and it is allowed to add others. The following keys are currently supported in MPI-IO:

"access_style" Specifies the manner in which a file will be accessed until the file is closed. The value is one of the following: **"read_once"**, **"write_once"**, **"read_mostly"**, **"write_mostly"**, **"sequential"**, **"reverse_sequential"**, **"random"**, or **"collective_only"**. Currently, only **"collective_only"** is used to optimize accesses. All processes must specify the same value for this key.

"cache_buffer_space" Specifies the total space to use per file per participating process for file caching. If caching is enabled for a file, this amount of cache space will be allocated and divided into as many cache slices as possible, where each slice is the natural size of contiguously accessible data for each process' view of the file. The value string must contain a positive integer, and must be greater than or equal to 4096, the smallest cache space allowed.

"conversion_buffer_size" Specifies the size of the buffer to use for translating between an internal representation to a noninternal data representation in a file. The value string must contain a positive integer. All processes must specify the same value for this key.

"filename" Specifies the file name used when a file was opened. This key is only supported in **MPI_File_get_info**.

"file_perm" Specifies the file permissions to use for file creation, which is only useful when passed to MPI_File_open with an *amode* of MPI_MODE_CREATE. The value must contain a valid HPSS permission specification for the Mode argument to **hpss_Chmod** (e.g., "0744"). All processes must specify the same value for this key.

"nb_proc" Specifies the number of parallel processes that will typically be assigned to run programs that access the file. The value string contains a positive integer. All processes must specify the same value for this key.

"striping_factor" Specifies the number of I/O devices across which the file should be striped. The value string contains a positive integer. All processes must specify the same value for this key.

"striping_unit" Specifies the suggested striping unit to be used for this file, in bytes. The value string must contain an integer greater than or equal to 1. The HPSS class of service closest to the requested level of striping will be used. All processes must specify the same value for this key.

"hpss_cos" Specifies the suggested HPSS COS to be used for this file. The value string must contain a valid class of service identifier (a positive integer) for the HPSS host environment. All processes must specify the same value for this key.

"hpss_sclasstype" Specifies the suggested HPSS type of storage class to be used for this file. The value string must be either "TAPE" or "DISK". All processes must specify the same value for this key.

"hpss_max_file_size" Specifies the maximum file size anticipated for the file. The value string must contain a positive integer. All processes must specify the same value for this key.

"hpss_min_file_size" Specifies the minimum file size anticipated for the file. The value string must contain a positive integer. All processes must specify the same value for this key.

"hpss_access_size" Specifies the anticipated access size for the file, per processor. The value string must contain a positive integer. All processes must specify the same value for this key.

Each open file has an associated set of hints, which can be represented in an MPI_Info object whose (key,value) pairs indicate the hints given to the open command that were honored by the open. The hints may be retrieved using **MPI_File_get_info**. The hints that this implementation currently supports are only useful at the time the file is opened. Hence, the hints arguments to **MPI_File_set_view**, **MPI_File_set_info** and **MPI_File_delete** are currently ignored.

MPI-IO hints are translated to HPSS hints at open and set view time. Unlike HPSS, MPI-IO does not assign priorities to hints, so all hints given through the MPI-IO interface have equal priority.

Applications can use **MPI_Info_create** and **MPI_Info_set** to construct the file hints to provide to **MPI_File_open**. Applications can use **MPI_Info_get** to retrieve hints from the **MPI_Info** object returned by **MPI_File_get_info**. **MPI_Info_free** may be used to free the MPI_Info objects returned by **MPI_File_get_info** or **MPI_Info_create**.

6.1.6.1 MPI_Info_create

Purpose

Create an MPI_Info object.

Synopsis

```
#include <mpio.h>

int
MPI_Info_create(
        MPI_Info *      info        /* OUT */
);
```

Description

MPI_Info_create creates a new MPI_Info object. The newly created object contains no key/value pairs.

Parameters

info	Returned created *info* object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*info* argument is NULL.
MPI_ERR_ENOMEM	Unable to allocate program space.

See Also

MPI_Info_free, MPI_Info_set, MPI_Info_get.

Notes

Noncollective.

6.1.6.2 MPI_Info_set

Purpose

Add a key/value pair to an MPI_Info object.

Synopsis

```
#include <mpio.h>

int
MPI_Info_set(
        MPI_Info        info,           /* IN/OUT */
        char *          key,            /* IN     */
        char *          value           /* IN     */
);
```

Description

MPI_Info_set adds the (*key*, *value*) pair to *info*, or overrides the previous value paired with key, if *key* is already defined in *info*.

Parameters

info	MPI_Info object to update.
key	Key to add or update.
value	Value for the associated *key*.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*info* is MPI_INFO_NULL, or *key* or *value* is NULL.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_INFO_KEY	*key* length exceeds MPI_MAX_INFO_KEY.
MPI_ERR_INFO_VALUE	*value* length exceeds MPI_MAX_INFO_VALUE.

See Also

MPI_Info_create, MPI_Info_delete, MPI_Info_get.

Notes

Noncollective.

6.1.6.3 MPI_Info_delete

Purpose

Delete a key/value pair from an MPI_Info object.

Synopsis

```
#include <mpio.h>

int
MPI_Info_delete(
        MPI_Info        info,       /* IN/OUT */
        char *          key         /* IN     */
);
```

Description

MPI_Info_delete deletes the key/value pair matching *key* from *info*.

Parameters

info	MPI_Info object to update.
key	Key to delete.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*key* argument is NULL or *info* argument is MPI_INFO_NULL.
MPI_ERR_INFO_KEY	*key* length exceeds MPI_MAX_INFO_KEY.
MPI_ERR_INFO_NOKEY	*key* is not defined in *info*.

See Also

MPI_Info_set, MPI_Info_get.

Notes

Noncollective.

6.1.6.4 MPI_Info_get

Purpose

Retrieve the value of a key / value pair from an MPI_Info object.

Synopsis

```
#include <mpio.h>

int
MPI_Info_get(
        MPI_Info    info,       /* IN  */
        char *      key,        /* IN  */
        int         valuelen,   /* IN  */
        char *      value,      /* OUT */
        int *       flag        /* OUT */
);
```

Description

MPI_Info_get retrieves the value associated with *key* from the given *info*. If *key* is defined in *info*, it sets *flag* to **true** and returns the value in *value*. Otherwise, it sets *flag* to **false** and leaves *value* unchanged. *valuelen* is the number of characters available in *value*. If *valuelen* is less than the string length of the value, the returned *value* is truncated.

Parameters

info	MPI_Info object to examine.
key	Key to retrieve.
valuelen	Maximum number of characters available in *value*.
value	Returned value associated with *key*.
flag	Indicator of whether or not *key* is defined in *info*.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*key*, *value* or *flag* argument is NULL or *info* argument is MPI_INFO_NULL.
MPI_ERR_INFO_KEY	*key* length exceeds MPI_MAX_INFO_KEY.]

See Also

MPI_Info_set, MPI_Info_delete, MPI_Info_get_valuelen.

Notes

Noncollective.

6.1.6.5 MPI_Info_get_valuelen

Purpose

Get the length of the value of a key / value pair in an MPI_Info object.

Synopsis

```
#include <mpio.h>

int
MPI_Info_get_valuelen(
        MPI_Info    info,       /* IN  */
        char *      key,        /* IN  */
        int *       valuelen,   /* OUT */
        int *       flag        /* OUT */
);
```

Description

MPI_Info_get_valuelen retrieves the length of the value associated with *key* from *info*. If *key* is defined in *info*, *valuelen* is set to the length of its associated value and *flag* is set to **true**. If *key* is not defined in *info*, *valuelen* is unchanged and *flag* is set to **false**.

Parameters

info	MPI_Info object to examine.
key	Key to retrieve.
valuelen	Returned length of the associated value for *key*.
flag	Returned indicator of whether *key* was defined in *info*.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*key*, *valuelen*, or *flag* argument is NULL or *info* argument is

	MPI_INFO_NULL.
MPI_ERR_INFO_KEY	*key* length exceeds MPI_MAX_INFO_KEY.

See Also

MPI_Info_free, MPI_Info_set, MPI_Info_get.

Notes

Noncollective.

6.1.6.6 MPI_Info_get_nkeys

Purpose

Get the number of keys defined in an MPI_Info object.

Synopsis

```
#include <mpio.h>

int
MPI_Info_get_nkeys(
        MPI_Info    info,       /* IN  */
        int *       nkeys       /* OUT */
);
```

Description

MPI_Info_get_nkeys returns in *nkeys* the number of keys currently defined in *info*.

Parameters

info	MPI_Info object to examine.
nkeys	Returned number of keys.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*nkeys* argument is NULL or *info* argument is MPI_INFO_NULL.

See Also

MPI_Info_get, MPI_Info_get_valuelen, MPI_Info_get_nthkey.

Notes

Noncollective.

6.1.6.7 MPI_Info_get_nthkey

Purpose

Get the nth key defined in an MPI_Info object.

Synopsis

```
#include <mpio.h>

int
MPI_Info_get_nthkey(
        MPI_Info    info,       /* IN  */
        int         n,          /* IN  */
        char *      key         /* OUT */
);
```

Description

MPI_Info_get_nthkey returns the nth *key* in *info*. Keys are numbered 0..N-1 where N is the value returned by **MPI_Info_get_nkeys**. All keys between 0 and N-1 are guaranteed to be defined. The number of a given *key* does not change as long as *info* is not modified with **MPI_Info_set** or **MPI_Info_delete**.

Parameters

info	MPI_Info object to examine.
n	Index of key to retrieve.
key	Returned key.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*info* is MPI_INFO_NULL or *key* is NULL.
MPI_ERR_INFO_NOKEY	*n* is invalid index for *info*.

See Also

MPI_Info_get_nkeys, MPI_Info_get_key.

Notes

Noncollective.

6.1.6.8 MPI_Info_dup

Purpose

Duplicate an MPI_Info object.

Synopsis

```
#include <mpio.h>

int
MPI_Info_dup(
        MPI_Info    info,       /* IN  */
        MPI_Info *  newinfo     /* OUT */
);
```

Description

MPI_Info_dup duplicates an existing MPI_Info object, creating a new object with the same (key,value) pairs and the same ordering of keys.

Parameters

info	MPI_Info object to duplicate.
newinfo	Returned duplicate of *info* object.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*newinfo* argument is NULL.
MPI_ERR_ENOMEM	Unable to allocate program space.

See Also

MPI_Info_free, MPI_Info_create.

Notes

Noncollective.

6.1.6.9 MPI_Info_free

Purpose

Free an MPI_Info object.

Synopsis

```
#include <mpio.h>

int
MPI_Info_free(
        MPI_Info *      info            /* IN/OUT */
);
```

Description

MPI_Info_free frees an MPI_Info object created by **MPI_Info_create** or **MPI_Info_dup**, and sets the *info* object to MPI_INFO_NULL.

Parameters

info	MPI_Info object to free.

Return Values

On successful completion the function returns MPI_SUCCESS. Otherwise, it returns one of the error codes listed below.

Error Conditions

MPI_ERR_ARG	*info* argument is NULL.

See Also

MPI_Info_create, **MPI_Info_dup**.

Notes

Noncollective.

6.2 C++ Language Bindings

This section describes the HPSS MPI-IO application programming interfaces for C++. These APIs are essentially wrappers that map the C++ calls into calls to the corresponding C APIs.

The C++ bindings differ from the C APIs in that these APIs are declared within the scope of a C++ namespace and are methods defined on C++ objects. If errors are detected in C++, an exception is raised instead of returning an error code, so the MPI-IO wrappers provide checks on the return values from the C functions, and raise an exception if **MPI_SUCCESS** is not returned. The mapping from C++ to C also has to take into account the differences in C++ and C MPI objects, which is all done automatically for the application.

C++ applications must be compiled with **#include <mpio.h>**. Furthermore, the inclusion of **<mpio.h>** must precede the inclusion of the host **<mpi.h>** file, to avoid naming conflicts with the host MPI's C++ APIs. The C++ compiler must support namespaces.

It is also possible for C++ applications to invoke the MPI-IO C functions directly, as well as the host MPI C functions. In this case, the application writer must provide the C++-2-C mappings needed. This is useful in the event the host MPI does not provide the C++ MPI namespace and other supporting MPI functions since the HPSS MPI-IO only provides C++ bindings for the subset of MPI methods that are defined or needed for MPI-IO.

The descriptions below provide only the language bindings for each API, to illustrate the difference in syntax and parameter declarations for the C++ APIs. Please refer to the corresponding C API in 6.1 for the complete functionality specification for each API.

Synopsis

```
#include <mpio.h>

void MPI::File::Call_errhandler(
        int         errorcode) const

void MPI::File::Close()

static MPI::Errhandler MPI::File::Create_errhandler(
        MPI::File::Errhandler_fn *      function)

static void MPI::File::Delete(
        const char *            filename,
        const MPI::Info&        info)

static MPI::File MPI::File::Open(
        const MPI::Intracomm&           comm,
        const char *                    filename,
        int                             amode,
        const MPI::Info&                info)

int MPI::File::Get_amode() const

int MPI::File::Get_atomicity() const

MPI::Offset MPI::File::Get_byte_offset(
        const MPI::Offset disp)

MPI::Errhandler MPI::File::Get_errhandler() const

MPI::Group MPI::File::Get_group() const

MPI::Info MPI::File::Get_info() const

MPI::Offset MPI::File::Get_position() const

MPI::Offset MPI::File::Get_position_shared() const

MPI::Offset MPI::File::Get_size() const

MPI::Aint MPI::File::Get_type_extent(
        const MPI::Datatype& datatype) const

void MPI::File::Get_view(
        MPI::Offset&            disp,
        MPI::Datatype&          etype,
        MPI::Datatype&          filetype,
        char *                  datarep)

MPI::Request MPI::File::Iread(
        void *                  buf,
        int                     count,
        const MPI::Datatype&    datatype)
```

```
MPI::Request MPI::File::Iread_at(
        MPI::Offset              offset,
        void *                   buf,
        int                      count,
        const MPI::Datatype&     datatype)

MPI::Request MPI::File::Iread_shared(
        void *                   buf,
        int                      count,
        const MPI::Datatype&     datatype)

MPI::Request MPI::File::Iwrite(
        void *                   buf,
        int                      count,
        const MPI::Datatype&     datatype)

MPI::Request MPI::File::Iwrite_at(
        MPI::Offset              offset,
        void *                   buf,
        int                      count,
        const MPI::Datatype&     datatype)

MPI::Request MPI::File::Iwrite_shared(
        void *                   buf,
        int                      count,
        const MPI::Datatype&     datatype)

static MPI::File MPI::File::Open(
        const MPI::Intracomm&    comm,
        const char *             filename,
        int                      amode,
        const MPI::Info&         info)

void MPI::File::Preallocate(
        MPI::Offset      size)

void MPI::File::Read(
        void *                   buf,
        int                      count,
        const MPI::Datatype&     datatype)

void MPI::File::Read(
        void *                   buf,
        int                      count,
        const MPI::Datatype&     datatype,
        MPI::Status&             status)

void MPI::File::Read_all(
        void *                   buf,
        int                      count,
        const MPI::Datatype&     datatype)

void MPI::File::Read_all(
        void *                   buf,
```

```
        int                     count,
        const MPI::Datatype&    datatype,
        MPI::Status&            status)

void MPI::File::Read_all_begin(
        void *                  buf,
        int                     count,
        const MPI::Datatype&    datatype)

void MPI::File::Read_all_end(
        void *                  buf)

void MPI::File::Read_all_end(
        void *                  buf,
        MPI::Status&            status)

void MPI::File::Read_at(
        MPI::Offset             offset,
        void *                  buf,
        int                     count,
        const MPI::Datatype&    datatype)

void MPI::File::Read_at(
        MPI::Offset             offset,
        void *                  buf,
        int                     count,
        const MPI::Datatype&    datatype,
        MPI::Status&            status)

void MPI::File::Read_at_all(
        MPI::Offset             offset,
        void *                  buf,
        int                     count,
        const MPI::Datatype&    datatype)

void MPI::File::Read_at_all(
        MPI::Offset             offset,
        void *                  buf,
        int                     count,
        const MPI::Datatype&    datatype,
        MPI::Status&            status)

void MPI::File::Read_at_all_begin(
        MPI::Offset             offset,
        void *                  buf,
        int                     count,
        const MPI::Datatype&    datatype)

void MPI::File::Read_at_all_end(
        void *                  buf)

void MPI::File::Read_at_all_end(
        void *                  buf,
        MPI::Status&            status)
```

```
void MPI::File::Read_ordered(
        void *                  buf,
        int                     count,
        const MPI::Datatype&    datatype)

void MPI::File::Read_ordered(
        void *                  buf,
        int                     count,
        const MPI::Datatype&    datatype,
        MPI::Status&            status)

void MPI::File::Read_ordered_begin(
        void *                  buf,
        int                     count,
        const MPI::Datatype&    datatype)

void MPI::File::Read_ordered_end(
        void *                  buf)

void MPI::File::Read_ordered_end(
        void *                  buf,
        MPI::Status&            status)

void MPI::File::Read_shared(
        void *                  buf,
        int                     count,
        const MPI::Datatype&    datatype)

void MPI::File::Read_shared(
        void *                  buf,
        int                     count,
        const MPI::Datatype&    datatype,
        MPI::Status&            status)

void MPI::File::Seek(
        MPI::Offset             offset,
        int                     whence)

void MPI::File::Seek_shared(
        MPI::Offset             offset,
        int                     whence)

void MPI::File::Set_atomicity(
        bool                    flag)

void MPI::File::Set_errhandler(
        const MPI::Errhandler&  errhandler)

void MPI::File::Set_info(
        const MPI::Info&        info)

void MPI::File::Set_size(
        MPI::Offset             size)
```

```
void MPI::File::Set_view(
        MPI::Offset              disp,
        const MPI::Datatype&     etype,
        const MPI::Datatype&     filetype,
        const char *             datarep,
        const MPI::Info&         info)

void MPI::File::Sync()

void MPI::File::Write(
        void *                   buf,
        int                      count,
        const MPI::Datatype&     datatype)

void MPI::File::Write(
        void *                   buf,
        int                      count,
        const MPI::Datatype&     datatype,
        MPI::Status&             status)

void MPI::File::Write_all(
        void *                   buf,
        int                      count,
        const MPI::Datatype&     datatype)

void MPI::File::Write_all(
        void *                   buf,
        int                      count,
        const MPI::Datatype&     datatype,
        MPI::Status&             status)

void MPI::File::Write_all_begin(
        void *                   buf,
        int                      count,
        const MPI::Datatype&     datatype)

void MPI::File::Write_all_end(
        void *                   buf)

void MPI::File::Write_all_end(
        void *                   buf,
        MPI::Status&             status)

void MPI::File::Write_at(
        MPI::Offset              offset,
        void *                   buf,
        int                      count,
        const MPI::Datatype&     datatype)

void MPI::File::Write_at(
        MPI::Offset              offset,
        void *                   buf,
        int                      count,
```

```
        const MPI::Datatype&   datatype,
        MPI::Status&           status)

void MPI::File::Write_at_all(
        MPI::Offset            offset,
        void *                 buf,
        int                    count,
        const MPI::Datatype&   datatype)

void MPI::File::Write_at_all(
        MPI::Offset            offset,
        void *                 buf,
        int                    count,
        const MPI::Datatype&   datatype,
        MPI::Status&           status)

void MPI::File::Write_at_all_begin(
        MPI::Offset            offset,
        void *                 buf,
        int                    count,
        const MPI::Datatype&   datatype)

void MPI::File::Write_at_all_end(
        void *                 buf)

void MPI::File::Write_at_all_end(
        void *                 buf,
        MPI::Status&           status)

void MPI::File::Write_ordered(
        void *                 buf,
        int                    count,
        const MPI::Datatype&   datatype)

void MPI::File::Write_ordered(
        void *                 buf,
        int                    count,
        const MPI::Datatype&   datatype,
        MPI::Status&           status)

void MPI::File::Write_ordered_begin(
        void *                 buf,
        int                    count,
        const MPI::Datatype&   datatype)

void MPI::File::Write_ordered_end(
        void *                 buf)

void MPI::File::Write_ordered_end(
        void *                 buf,
        MPI::Status&           status)

void MPI::File::Writed_shared(
        void *                 buf,
```

```
        int                         count,
        const MPI::Datatype&        datatype)

void MPI::File::Write_shared(
        void *                      buf,
        int                         count,
        const MPI::Datatype&        datatype,
        MPI::Status&                status)

static MPI::Info MPI::Info::Create()

void MPI::Info::Delete(
        const char *                key)

MPI::Info MPI::Info::Dup() const

void MPI::Free()

bool MPI::Info::Get(
        const char *                key,
        int                         valuelen,
        char *                      value) const

int MPI::Info::Get_nkeys() const

void MPI::Info::Get_nthkey(
        int                         n,
        char *                      key) const

bool MPI::Info::Get_valuelen(
        const char *                key,
        int&                        valuelen) const

void MPI::Info::Set(
        const char *                key,
        const char *                value)

void MPI::Register_datarep(
        const char *                                datarep,
        MPI::Datarep_conversion_function *          read_conversion_fn,
        MPI::Datarep_conversion_function*           write_conversion_fn,
        MPI::Datarep_extent_function *              dtype_file_extent_fn,
        void *                                      extra_state)
```

6.3 Fortran Language Bindings

This section describes the HPSS MPI-IO application programming interfaces for Fortran77. All of the Fortran APIs are essentially wrappers that translate the Fortran arguments into appropriate C arguments, invoke the C APIs, and translate the C results into the appropriate Fortran outputs. The Fortran interfaces differ from C in that error return values are assigned to the output parameter

IERROR; that is, these interfaces are subroutines instead of functions. Other differences between Fortran and C are that address-sized values (such as offsets, displacements, etc.) are INTEGER-sized. This means that in Fortran, **MPI_Aint** is a 32-bit quantity, but in C, it is generally a 64-bit quantity, for most host MPIs. Another difference is that handles of opaque objects are INTEGERs in Fortran. The MPI-2 standard has defined conversion functions for translating Fortran-2-C and C-2-Fortran, which is automatically done for the MPI-IO APIs defined here.

Applications using the HPSS MPI-IO interfaces must be compiled with **#include <mpiof.h>**, using a compiler that supports C preprocessing directives and KIND-parameterized integers.

The descriptions below provide only the language bindings for each API, to illustrate the difference in syntax and parameter declarations for the Fortran APIs. Please refer to the corresponding C API in 6.1 for the complete functionality specification for each API.

Synopsis

#include <mpiof.h>

MPI_FILE_CALL_ERRHANDLER(*FH***,** *ERRORCODE***,** *IERROR***)**
INTEGER *FH***,** *ERRORCODE***,** *IERROR*

MPI_FILE_CLOSE(*FH***,** *IERROR***)**
INTEGER *FH***,** *IERROR*

MPI_FILE_CREATE_ERRHANDLER(*FUNCTION***,** *ERRHANDLER***,** *IERROR***)**
EXTERNAL *FUNCTION*
INTEGER *ERRHANDLER***,** *IERROR*

MPI_FILE_DELETE(*FILENAME***,** *INFO***,** *IERROR***)**
CHARACTER*(*) *FILENAME*
INTEGER *INFO***,** *IERROR*

MPI_FILE_GET_AMODE(*FH***,** *AMODE***,** *IERROR***)**
INTEGER *FH***,** *AMODE***,** *IERROR*

MPI_FILE_GET_ATOMICITY(*FH***,** *FLAG***,** *IERROR***)**
INTEGER *FH***,** *FLAG***,** *IERROR*

MPI_FILE_GET_BYTE_OFFSET(*FH***,** *OFFSET***,** *DISP***,** *IERROR***)**
INTEGER *FH***,** *IERROR*
INTEGER (KIND=MPI_OFFSET_KIND) *OFFSET***,** *DISP*

MPI_FILE_GET_ERRHANDLER(*FH***,** *ERRHANDLER***,** *IERROR***)**
INTEGER *FH***,** *ERRHANDLER***,** *IERROR*

MPI_FILE_GET_GROUP(*FH***,** *GROUP***,** *IERROR***)**
INTEGER *FH***,** *GROUP***,** *IERROR*

MPI_FILE_GET_INFO(*FH***,** *INFO_USED***,** *IERROR***)**
INTEGER *FH***,** *INFO_USED***,** *IERROR*

MPI_FILE_GET_POSITION(*FH***,** *OFFSET***,** *IERROR***)**
INTEGER *FH***,** *IERROR*
INTEGER (KIND=MPI_OFFSET_KIND) *OFFSET*

MPI_FILE_GET_POSITION_SHARED(*FH, OFFSET, IERROR***)**
INTEGER *FH, IERROR*
INTEGER (KIND=MPI_OFFSET_KIND) *OFFSET*

MPI_FILE_GET_SIZE(*FH, SIZE, IERROR***)**
INTEGER *FH, IERROR*
INTEGER (KIND=MPI_OFFSET_KIND) *SIZE*

MPI_FILE_GET_TYPE_EXTENT(*FH, DATATYPE, EXTENT, IERROR***)**
INTEGER *FH, DATATYPE, IERROR*
INTEGER (KIND=MPI_ADDRESS_KIND) *EXTENT*

MPI_FILE_GET_VIEW(*FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR***)**
INTEGER *FH, ETYPE, FILETYPE, IERROR*
CHARACTER*(*) *DATAREP*
INTEGER (KIND=MPI_OFFSET_KIND) *DISP*

MPI_FILE_IREAD(*FH, BUF, COUNT, DATATYPE, REQUEST, IERROR***)**
<type> *BUF***(*)**
INTEGER *FH, COUNT, DATATYPE, REQUEST, IERROR*

MPI_FILE_IREAD_AT(*FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR***)**
<type> *BUF***(*)**
INTEGER *FH, COUNT, DATATYPE, REQUEST, IERROR*
INTEGER (KIND=MPI_OFFSET_KIND) *OFFSET*

MPI_FILE_IREAD_SHARED(*FH, BUF, COUNT, DATATYPE, REQUEST, IERROR***)**
<type> *BUF***(*)**
INTEGER *FH, COUNT, DATATYPE, REQUEST, IERROR*

MPI_FILE_IWRITE(*FH, BUF, COUNT, DATATYPE, REQUEST, IERROR***)**
<type> *BUF***(*)**
INTEGER *FH, COUNT, DATATYPE, REQUEST, IERROR*

MPI_FILE_IWRITE_AT(*FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR***)**
<type> *BUF***(*)**
INTEGER *FH, COUNT, DATATYPE, REQUEST, IERROR*
INTEGER (KIND=MPI_OFFSET_KIND) *OFFSET*

MPI_FILE_IWRITE_SHARED(*FH, BUF, COUNT, DATATYPE, REQUEST, IERROR***)**
<type> *BUF***(*)**
INTEGER *FH, COUNT, DATATYPE, REQUEST, IERROR*

MPI_FILE_OPEN(*COMM, FILENAME, AMODE, INFO, FH, IERROR***)**
CHARACTER* (*) *FILENAME*
INTEGER *COMM, AMODE, INFO, IERROR*

MPI_FILE_PREALLOCATE(*FH, SIZE, IERROR***)**
INTEGER *FH, IERROR*
INTEGER (KIND=MPI_OFFSET_KIND) *SIZE*

MPI_FILE_READ(*FH, BUF, COUNT, DATATYPE, STATUS, IERROR***)**
<type> *BUF***(*)**
INTEGER *FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR*

MPI_FILE_READ_ALL(*FH***,** *BUF***,** *COUNT***,** *DATATYPE***,** *STATUS***,** *IERROR***)**
<type> *BUF***(*)**
INTEGER *FH***,** *COUNT***,** *DATATYPE***,** *STATUS(MPI_STATUS_SIZE)***,** *IERROR*

MPI_FILE_READ_ALL_BEGIN(*FH***,** *BUF***,** *COUNT***,** *DATATYPE***,** *IERROR***)**
<type> *BUF***(*)**
INTEGER *FH***,** *COUNT***,** *DATATYPE***,** *IERROR*

MPI_FILE_READ_ALL_END(*FH***,** *BUF***,** *STATUS***,** *IERROR***)**
<type> *BUF***(*)**
INTEGER *FH***,** *STATUS(MPI_STATUS_SIZE)***,** *IERROR*

MPI_FILE_READ_AT(*FH***,** *OFFSET***,** *BUF***,** *COUNT***,** *DATATYPE***,** *STATUS***,** *IERROR***)**
<type> *BUF***(*)**
INTEGER *FH***,** *COUNT***,** *DATATYPE***,** *STATUS(MPI_STATUS_SIZE)***,** *IERROR*
INTEGER (KIND=MPI_OFFSET_KIND) *OFFSET*

MPI_FILE_READ_AT_ALL(*FH***,** *OFFSET***,** *BUF***,** *COUNT***,** *DATATYPE***,** *STATUS***,** *IERROR***)**
<type> *BUF***(*)**
INTEGER *FH***,** *COUNT***,** *DATATYPE***,** *STATUS(MPI_STATUS_SIZE)***,** *IERROR*
INTEGER (KIND=MPI_OFFSET_KIND) *OFFSET*

MPI_FILE_READ_AT_ALL_BEGIN(*FH***,** *OFFSET***,** *BUF***,** *COUNT***,** *DATATYPE***,** *IERROR***)**
<type> *BUF***(*)**
INTEGER *FH***,** *COUNT***,** *DATATYPE***,** *IERROR*
INTEGER (KIND=MPI_OFFSET_KIND) *OFFSET*

MPI_FILE_READ_AT_ALL_END(*FH***,** *BUF***,** *STATUS***,** *IERROR***)**
<type> *BUF***(*)**
INTEGER *FH***,** *STATUS(MPI_STATUS_SIZE)***,** *IERROR*

MPI_FILE_READ_ORDERED(*FH***,** *BUF***,** *COUNT***,** *DATATYPE***,** *STATUS***,** *IERROR***)**
<type> *BUF***(*)**
INTEGER *FH***,** *COUNT***,** *DATATYPE***,** *STATUS(MPI_STATUS_SIZE)***,** *IERROR*

MPI_FILE_READ_ORDERED_BEGIN(*FH***,** *BUF***,** *COUNT***,** *DATATYPE***,** *IERROR***)**
<type> *BUF***(*)**
INTEGER *FH***,** *COUNT***,** *DATATYPE***,** *IERROR*

MPI_FILE_READ_ORDERED_END(*FH***,** *BUF***,** *STATUS***,** *IERROR***)**
<type> *BUF***(*)**
INTEGER *FH***,** *STATUS(MPI_STATUS_SIZE)***,** *IERROR*

MPI_FILE_READ_SHARED(*FH***,** *BUF***,** *COUNT***,** *DATATYPE***,** *STATUS***,** *IERROR***)**
<type> *BUF***(*)**
INTEGER *FH***,** *COUNT***,** *DATATYPE***,** *STATUS(MPI_STATUS_SIZE)***,** *IERROR*

MPI_FILE_SEEK(*FH***,** *OFFSET***,** *WHENCE***,** *IERROR***)**
INTEGER *FH***,** *WHENCE***,** *IERROR*
INTEGER (KIND=MPI_OFFSET_KIND) *OFFSET*

MPI_FILE_SEEK_SHARED(*FH***,** *OFFSET***,** *WHENCE***,** *IERROR***)**
INTEGER *FH***,** *WHENCE***,** *IERROR*

INTEGER (KIND=MPI_OFFSET_KIND) *OFFSET*

MPI_FILE_SET_ATOMICITY(*FH, FLAG, IERROR***)**
INTEGER *FH, FLAG, IERROR*

MPI_FILE_SET_ERRHANDLER(*FH, ERRHANDLER, IERROR***)**
INTEGER *FH, ERRHANDLER, IERROR*

MPI_FILE_SET_INFO(*FH, INFO, IERROR***)**
INTEGER *FH, INFO, IERROR*

MPI_FILE_SET_SIZE(*FH, SIZE, IERROR***)**
INTEGER *FH, IERROR*
INTEGER (KIND=MPI_OFFSET_KIND) *SIZE*

MPI_FILE_SET_VIEW(*FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR***)**
INTEGER *FH, ETYPE, FILETYPE, INFO, IERROR*
CHARACTER*(*) *DATAREP*
INTEGER (KIND=MPI_OFFSET_KIND) *DISP*

MPI_FILE_SYNC(*FH, IERROR***)**
INTEGER *FH, IERROR*

MPI_FILE_WRITE(*FH, BUF, COUNT, DATATYPE, STATUS, IERROR***)**
<type> *BUF***(*)**
INTEGER *FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR*

MPI_FILE_WRITE_ALL(*FH, BUF, COUNT, DATATYPE, STATUS, IERROR***)**
<type> *BUF***(*)**
INTEGER *FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR*

MPI_FILE_WRITE_ALL_BEGIN(*FH, BUF, COUNT, DATATYPE, IERROR***)**
<type> *BUF***(*)**
INTEGER *FH, COUNT, DATATYPE, IERROR*

MPI_FILE_WRITE_ALL_END(*FH, BUF, STATUS, IERROR***)**
<type> *BUF***(*)**
INTEGER *FH, STATUS(MPI_STATUS_SIZE), IERROR*

MPI_FILE_WRITE_AT(*FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR***)**
<type> *BUF***(*)**
INTEGER *FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR*
INTEGER (KIND=MPI_OFFSET_KIND) *OFFSET*

MPI_FILE_WRITE_AT_ALL(*FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR***)**
<type> *BUF***(*)**
INTEGER *FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR*
INTEGER (KIND=MPI_OFFSET_KIND) *OFFSET*

MPI_FILE_WRITE_AT_ALL_BEGIN(*FH, OFFSET, BUF, COUNT, DATATYPE, IERROR***)**
<type> *BUF***(*)**
INTEGER *FH, COUNT, DATATYPE, IERROR*
INTEGER (KIND=MPI_OFFSET_KIND) *OFFSET*

MPI_FILE_WRITE_AT_ALL_END(*FH***,** *BUF***,** *STATUS***,** *IERROR***)**
<type> *BUF***(*)**
INTEGER *FH***,** *STATUS(MPI_STATUS_SIZE)***,** *IERROR*

MPI_FILE_WRITE_ORDERED(*FH***,** *BUF***,** *COUNT***,** *DATATYPE***,** *STATUS***,** *IERROR***)**
<type> *BUF***(*)**
INTEGER *FH***,** *COUNT***,** *DATATYPE***,** *STATUS(MPI_STATUS_SIZE)***,** *IERROR*

MPI_FILE_WRITE_ORDERED_BEGIN(*FH***,** *BUF***,** *COUNT***,** *DATATYPE***,** *IERROR***)**
<type> *BUF***(*)**
INTEGER *FH***,** *COUNT***,** *DATATYPE***,** *IERROR*

MPI_FILE_WRITE_ORDERED_END(*FH***,** *BUF***,** *STATUS***,** *IERROR***)**
<type> *BUF***(*)**
INTEGER *FH***,** *STATUS(MPI_STATUS_SIZE)***,** *IERROR*

MPI_FILE_WRITE_SHARED(*FH***,** *BUF***,** *COUNT***,** *DATATYPE***,** *STATUS***,** *IERROR***)**
<type> *BUF***(*)**
INTEGER *FH***,** *COUNT***,** *DATATYPE***,** *STATUS(MPI_STATUS_SIZE)***,** *IERROR*

MPI_INFO_CREATE(*INFO***,** *IERROR***)**
INTEGER *INFO***,** *IERROR*

MPI_INFO_DELETE(*INFO***,** *KEY***,** *IERROR***)**
INTEGER *INFO***,** *IERROR*
CHARACTER *(*) *KEY*

MPI_INFO_DUP(*INFO***,** *NEWINFO***,** *IERROR***)**
INTEGER *INFO***,** *NEWINFO***,** *IERROR*

MPI_INFO_FREE(*INFO***,** *IERROR***)**
INTEGER *INFO***,** *IERROR*

MPI_INFO_GET(*INFO***,** *KEY***,** *VALUELEN***,** *VALUE***,** *FLAG***,** *IERROR***)**
INTEGER *INFO***,** *VALUELEN***,** *IERROR*
CHARACTER *(*) *KEY***,** *VALUE*
LOGICAL *FLAG*

MPI_INFO_GET_NKEYS(*INFO***,** *NKEYS***,** *IERROR***)**
INTEGER *INFO***,** *NKEYS***,** *IERROR*

MPI_INFO_GET_NTH_KEY(*INFO***,** *N***,** *KEY***,** *IERROR***)**
INTEGER *INFO***,** *N***,** *IERROR*
CHARACTER *(*) *KEY*

MPI_INFO_GET_VALUELEN(*INFO***,** *KEY***,** *VALUELEN***,** *FLAG***,** *IERROR***)**
INTEGER *INFO***,** *VALUELEN***,** *IERROR*
CHARACTER *(*) *KEY*
LOGICAL *FLAG*

MPI_INFO_SET(*INFO***,** *KEY***,** *VALUE***,** *IERROR***)**
INTEGER *INFO***,** *IERROR*
CHARACTER *(*) *KEY***,** *VALUE*

MPI_REGISTER_DATAREP(*DATAREP***,** *READ_CONVERSION_FN***,**
*WRITE_CONVERSION_FN***,** *DTYPE_FILE_EXTENT_FN***,** *EXTRA_STATE***,**
*IERROR***)**
CHARACTER *(*) *DATAREP*
EXTERNAL *READ_CONVERSION_FN***,** *WRITE_CONVERSION_FN*
EXTERNAL *DTYPE_FILE_EXTENT_FN*
INTEGER (KIND=MPI_ADDRESS_KIND) *EXTRA_STATE*
INTEGER *IERROR*

6.4 Data Definitions

This section describes all externally used data definitions that are provided by MPI-IO. A data definition may be represented by constructs such as data structures and constants. For each data definition, a description and a format (including parameter descriptions) are provided.

6.4.1 MPI-IO Standard Data Definitions

The following types are specified in the MPI-2 standard and used by MPI-IO.

Most of the types added by MPI-IO are opaque MPI object types, such as **MPI_File**, **MPI_Info**, and **MPI_File_errhandler**. To an application, each of these types is accessed only through a handle. The only MPI-IO data type whose contents are explicitly available to an application is **MPI_Offset**.

MPI-IO and other MPI-2 constants and defaults are described in 6.1 with the relevant APIs that use them.

6.4.1.1 MPI_Offset

Description
An **MPI_Offset** is required to be a signed, 64-bit integral type that may be used in integer arithmetic operations. MPI-IO assumes the availability of a native 64-bit integral type on platforms supporting MPI-IO/HPSS. For the platforms MPI-IO currently supports, this is available as type **long long**.

Format
typedef long long MPI_Offset;

In C++, **MPI::Offset** is the corresponding type, defined within the **MPI** namespace.

In Fortran77, the corresponding type is an **INTEGER (KIND=MPI_OFFSET_KIND)**.

6.5 Troubleshooting

A number of problems can arise in developing HPSS MPI-IO applications, from compile and load errors to run time diagnostics. This section attempts to summarize common errors and potential solutions, and to give hints on how to achieve optimal performance.

6.5.1 Compilation issues

A thread-safe compilation should be specified. For C++, the compiler must support namespaces. For Fortran, the compiler must support C preprocessing directives. Most compilers have command line options to produce object files and executables that are thread-safe, usually including macro definitions for correct preprocessing of header files, and the automatic loading of thread-safe languages. For both AIX and Solaris, use **-D_THREAD_SAFE** or guarantee that your default compilation mode predefines this for you. On Solaris, you should also use **-mt** on the compilation and load lines.

Note that the host MPI usually provides specific compiler scripts or executables that provide the appropriate include and load paths automatically. Specifically, it is recommended that you use the following host interfaces with HPSS MPI-IO:

mpcc_r with IBM's PE MPI

tmcc with Solaris' HPC MPI

mpicc with MPICH

These host MPI scripts may predefine needed macros and include paths for you, but this section describes what is needed so that you can provide these yourself or verify that the host script is providing them.

In order to compile for HPSS MPI-IO, applications and libraries must have **#include "mpio.h"** in their sources, and must provide the include path **-I/opt/hpss/include** on the compile line. In general, the inclusion of **mpio.h** should replace then inclusion of the host MPI's **mpi.h**, as the latter is included through **mpio.h** in a language-appropriate way. For C++ compilations, the inclusion of **mpio.h** must precede that of the host **mpi.h**, or there will be naming conflicts. For Fortran compilations, **mpiof.h** should be used instead of **mpio.h**.

For applications that will use the DCE-based MPI-IO (uses the HPSS DCE CLAPI), additional macros need to be defined to assure that executables are compatible with DCE libraries, in addition to the macros that are needed for thread-safety. Similarly, for applications that use the non-DCE CLAPI, different macros are needed. The DCE flags needed vary, depending on whether you are on an AIX or Solaris platform:

With DCE AIX:

```
-D_AIX_PTHREADS_D7 -D_AES_SOURCE -D_AIX32_THREADS=1 -DAIX_PROD
```

With DCE Solaris:

```
-D_XOPEN_SOURCE -D_XOPEN_SOURCE_EXTENDED=1 -D__EXTENSIONS__
```

With non-DCE, AIX or Solaris:

```
-DNO_DCE
```

In addition to the macros to be defined, you need to specify the include paths to search for header files such as **mpio.h** and the other files it implicitly includes. You will have to include the host MPI path, *MPI_PATH*, and if using the DCE CLAPI, the path to the Encina client software, *ENCINA_PATH*. (Check with your HPSS system administrator for the correct *MPI_PATH* and *ENCINA_PATH* to use, and see the sample Makefile in Appendix F for how these are typically defined.) The following **-I** directives should be included on the compile line:

-I/opt/hpss/include -I*MPI_PATH*/include

With DCE AIX, you need to add:
-I/usr/include/dce -I*ENCINA_PATH*/include

The following errors can arise during compilation, which need to be resolved as described.

- Missing **mpio.h** or **mpio_MPI_config.h**: the compiler's preprocessor is unable to locate the HPSS MPI-IO include directory. Check that **-I/opt/hpss/include** is being used on the compile line. If it is included, check with your HPSS sysadmin as MPI-IO was not built for your system.

- Missing **mpi.h**: the compiler's preprocessor is unable to locate the host MPI include directory. Check that you are using the host-provided MPI compile interface as described above, and check where the host MPI that was used to build HPSS MPI-IO is installed. You may need to add **-I** with the appropriate path to the host MPI **include** directory to your compile line.

- Conflicts between the host **mpi.h** and **mpio.h** definitions (e.g. duplicate or missing definitions are reported): most likely, this indicates that HPSS MPI-IO was built with a different host MPI than the one you are attempting to use. Check with your HPSS sysadmin to verify what host MPI was used to configure HPSS MPI-IO. The sysadmin may need to update HPSS MPI-IO for a newer version of the host MPI, or rebuild HPSS MPI-IO if the one installed does not match the intended use. If the problems persist, HPSS MPI-IO may never have been configured for the host MPI you are trying to use, and the sysadmin will need to request an HPSS configuration update.

- Missing DCE/Encina include files: if the DCE or Encina clients have not been fully installed on your platform, missing include files will be reported at compile time. Contact your HPSS sysadmin for assistance in obtaining the needed files.

6.5.2 Load/Link issues

As described in Section 6.5.1, the host-provided MPI interface (**mpicc, mpcc_r, or mpcc**) should be used for loading as well as for compiling, to guarantee the host MPI and dependent libraries are loaded correctly. The HPSS libraries as described in Section 1.5.4must also be loaded. As with compilation, additional library paths and libraries need to be used on the load line, and these vary, depending on whether or not you are using DCE, AIX/Solaris, and the source language of the application. Your host MPI script may provide some of these, which you must verify this, and also check that required ordering is provided.

Library ordering is important! There are multiple definitions of some basic functions in these libraries, and failing to load the libraries in the correct order can result in unusual runtime errors that are difficult to debug.

All MPI-IO applications must first load the MPI-IO and HPSS libraries:
-L/opt/hpss/lib -lmpioapi -lhpss

Next, any platform-specific libraries should be loaded, and these vary slightly, depending on whether or not DCE is being used:

With DCE AIX:
-ldcepthreads -lpthreads_compat -lpthreads

With non-DCE AIX:
-lpthreads_compat -lpthreads -lm

With DCE Solaris:
-lnsl -lsocket -z muldefs

With non-DCE Solaris:
-lnsl -lsocket -lm -z muldefs

If DCE is being used, the Encina and DCE libraries are loaded next:
-L*ENCINA_PATH*/lib -lEncina -lEncClient -ldce

Finally, the host MPI library, ***mpi_lib***, should be included:
-l*mpi_lib*

Note that if you are using the MPI host-provided script to do the load, this will automatically be added for you. If not, you need to use the host MPI name for ***mpi_lib***, which is **mpich** if the MPI host is MPICH, **mpi_r** if the MPI host is the IBM SMP, and **mpi_mt** if the MPI host is SUN HPC.
-L*MPI_PATH* -l*mpi_lib*

See the sample Makefile included in Appendix F.

Notes:

- Solaris loads require -z muldefs to suppress error messages about multiple definitions, but because this allows multiple definitions, some implicitly loaded libraries must be explicitly moved up in the load sequence to guarantee the right ordering. For Solaris Fortran codes, you must explicitly load the C library (**-lc**) before the DCE libraries. For Solaris C++ codes, you must explicitly order some of the implicitly loaded libraries. Instead of **-lmpi_mt**, use:

  -lmpi++_mt -lmpi_mt -lm -lw -lcx -lc

  Thread-safe versions of all libraries should be used. Using the host-provided scripts should automatically do this for you, but you should verify this if there is any doubt.

The following load/link errors may occur with HPSS MPI-IO applications.

- Missing **libmpioapi.a** or **libhpss.a**: the loader is unable to locate the HPSS library directory. You need to add **-L/opt/hpss** (or other **-L** option, as appropriate) to your load line.

- Missing **libmpi.a** or **libmpich.a** or **libmpi_r.a**: the loader is unable to locate the host MPI library. Check that you are using the host-provided MPI compile/link script, as described in 6.5.1 above. You may need to add **-L** with the appropriate path to the host MPI **lib** directory to your load line.

- Missing **libdce.a**, **libEncina.a**, or **libEncClient.a**: the loader is unable to locate the DCE and Encina libraries. These may need to be installed for your platform. Contact your HPSS sysadmin for assistance.

6.5.3 Run-time errors

Unlike most MPI applications, the default error handling for MPI-IO applications is set in **MPI_Init** to **MPI_ERRORS_RETURN**. This forces most errors to be returned to the application rather than causing the application to abort. (There are some internal errors that are nonrecoverable, and if these are detected, the application will be aborted.) Thus the application developer must check for potential error returns and provide error recovery in the presence of errors. For example, if an **MPI_File_open** fails, it is obviously erroneous to attempt to read or write using the returned file handle.

Error messages that describe the error can be obtained using the **MPI_Error_string** function, and the environmment variables **MPIO_DEBUG** and **HPSS_DEBUG** may be set (to any nonzero value) to enable more diagnostic information being included in **stderr** messages.

Section 6.1 lists all the errors that may be returned by each MPI-IO API, and section 6.1.5 provides more information on APIs for error handling.

Errors relating to HPSS may indicate system failures or unavailability which should be reported to your HPSS sysadmin.

6.5.4 Thread-safety issues

HPSS MPI-IO imposes a multithreaded model on applications which may not otherwise be thread-aware. Application developers need to be conscious of the multithreading of the HPSS MPI-IO library, and to take some precautions when using HPSS MPI-IO.

Not all host MPIs provide a thread-safe MPI library. HPSS MPI-IO attempts to circumvent thread race conditons by preventing more than one thread from making MPI calls simultaneously by using a mutex lock/unlock wrapping around MPI calls. This is successful within the MPI-IO library itself, but if the application is multithreaded, or if libraries are used that make MPI calls, there is no guarantee that the resulting mixture will be thread-safe.

The best solution is to use a thread-safe host MPI. If your MPI host is thread-safe, the rest of this section will probably not apply to you. If not, HPSS MPI-IO may still be viable with your host MPI, but more of a burden is placed on the application developer to avoid thread-safety errors.

The typical error seen in an application with a lack of thread-safety is that the application deadlocks. This usually indicates that more than one thread at a time is attempting to make MPI calls in a process. If this is so, examination of the applications' threads with the aid of a debugger will indicate one or more threads waiting to perform a mutex lock, without apparent progress.

Another error that can occur due to multithreading are race conditions that can cause results to differ across different executions of the same application. Race conditions sometimes uncover programming errors, such as failing to initialize a variable before use, that are just never seen when running in nonthreaded mode. More often, they just indicate a need for syncronization that was not previously observed.

With HPSS MPI-IO, both of these situations are usually due to a failure to syncronize between the MPI-IO library and the application. **MPI_File_sync**, which guarantees that all participating processes are at a stable and quiescent state of execution with respect to a given file, may be useful for imposing the needed syncronization. **MPI_Barrier**, which is typically used for MPI

syncronization, may introduce the errors described above, due to the lack of a thread-safe host MPI. Where possible, **MPI_File_sync** should be used instead.

Lastly, a multithreaded application should adhere to the MPI-1 restriction that only one thread is allowed to make MPI calls, and only one collective call may be active at a time on any given communicator. Further, the MPI-2 specification for MPI-IO prohibits concurrent collective activity on a file handle. A failure to observe these restrictions can also lead to deadlock or other thread-safety errors.

6.5.5 Performance issues

The effect of HPSS and MPI-IO environment variables on the potential performance of applications is discussed in 1.1.5 and 1.5.5

Probably the most important performance consideration is matching the pattern of file accesses to the HPSS file storage class characteristics. The optimal performance should be achieved when the striping width and length of the HPSS file match the number of participating processes and the filetype extent per process, respectively. File hints can be used to provide information when the file is opened about how to select an appropriate HPSS COS (Class of Service) for the file characteristics expected. See 6.1.6 for more information on how to use these hints. Recall that HPSS performance is best with large files being accessed in regular (striped) access patterns.

The automatic caching facility can be enabled by opening the file with **MPI_MODE_UNIQUE_OPEN**. Caching of data slices from a file is expected to improve small data accesses when the accesses to the file by each participating process exhibit high locality of reference. Caching issues are discussed in more detail in 6.5.6

Use of collective data accesses where possible will allow MPI-IO to optimize access to the HPSS file system, compared to equivalent noncollective accesses. Collective accesses can also minimize potential fragmentation of the HPSS file due to the distribution of the file data across processes. Excessive file fragmentation can result in HPSS disallowing further writes to a file.

6.5.6 Caching issues

Application developers must be aware of potential hazards and performance issues when enabling the automatic caching facility by opening files with **MPI_MODE_UNIQUE_OPEN**. When caching is enabled, there will be a cache-miss overhead imposed, which essentially is a double buffering penalty. If data accesses exhibit good locality of reference, the overhead penalty will quickly be ameliorated by perfomance improvement after the cache has been primed.

Caching should not be enabled for data access patterns that do not exhibit locality.

Caching should not be enabled if the same file is concurrently open through any other process not participating in the file open. Cache and file coherency can only be guaranteed if the unique open mode restrictions are observed.

The caching strategy used determines whether or not a file is eligible for cached accessing when the file view is set. To be eligible, each participating process' view must be distinct and have no overlapping areas of the file shared with another process. Note that the default file view used when a file is opened will not be eligible for cached access, unless the communicator used to open the file has only one participating process. The filetypes of all the participating processes are used to

determine a natural cache size to use per process, which usually represents the stripe size of data per process in the file. If this cache size falls within prescribed min / max limits, a cache space will be allocated for each process for 1 or more cache buffers of the determined cache size.

Subsequent data accesses of the file will be cached, provided the amount of data being accessed is less than or equal to the size of a cache buffer and are contained within a single cache buffer (i.e., do not span more than one cache buffer). A least-recently-used strategy will be used to locate a cache buffer to flush and reuse, in the event of a cache miss.

The default maximum cache space allocated for a file is 64 MB. The file hint for **cache_buffer_space** (as described in 6.1.6) can be used to override the default minimum cache size, which is 4 KB. The default minimum cache size allowed is 4 KB and the maximum cache size allowed is 16 MB. The number of cache buffers per file is determined from the max cache space and the cache size: the cache space allocated will be divided up into as many buffers as possible.

Caching may cause **MPI_ERR_ENOMEM** errors, if the cache space allocated for files is excessive. If this is suspected, caching should be disabled, unless the space available to the application can be increased (e.g., by increasing user or system limits).

Users should be aware of a potential need for extra synchronization in the presence of caching to avoid deadlocks. The underlying caching ops to load or flush cache buffers generate reads or writes which may need to be synchronized. For example, **MPI_File_get_size()** is a noncollective call and in the presence of caching will only be accurate per process. In order to guarantee that the correct collective size is obtained, all processes must synchronize by first invoking **MPI_File_sync()**, which will flush all cache buffers as needed and guarantee that the written file size is correct for all processes.

Chapter 7 Site Interfaces

This chapter describes all Site Interfaces which are provided for use by two shared libraries (Account Validation site policy and Gatekeeping site policy) which will each be dynamically loaded with the Gatekeeper Server. The Site interface specification includes the following information:

- Name
- Purpose
- Syntax
- Description
- Parameters
- Return Values
- Error Conditions
- Related Information
- Clients
- Notes
- Example Use

7.1 Gatekeeping

This section describes the Gatekeeping Site Interfaces. Function definitions for these Site Interfaces are provided with the HPSS release, thus the Gatekeeper Server installed by default does NO gatekeeping. Sites will need to enhance these interfaces to implement local policy rules. (The site interface code exists in /opt/hpss/src/sitelib/gk.) Each Site Interface will be called by a Gatekeeping Service API. Please refer to the "Example Use" section of each Site Interface for example "pseudo code" implementation details of a site defined policy.

Please note that the code listed here is merely to give an idea of example uses. It has not been tested in an actual system.

7.1.1 gk_site_Close

Purpose

This is an internal, site defined-and-implemented function that is called by the Gatekeeper whenever a file is closed. This function is not an RPC. It is a call to a procedure in a shared library.

Syntax

```
signed32
gk_site_Close (
        uuid_t          ControlNo);     /* IN */
```

Description

This function is written by the customer site. It will be called by **gk_Close** while processing a close of an HPSS file.

Parameters

ControlNo	Unique identifier for the opened file which is now being closed. This number was generated by the Gatekeeper and passed into **gk_site_Open** or **gk_site_OpenStats**.

Return Values

Upon successful completion, a value of zero (0) is returned. Non-zero values are described in the Error Conditions.

Error Conditions

The **gk_site_Close** routine is unsuccessful if any of the following are true:

HPSS_EAGAIN	Resources are temporarily unavailable.
HPSS_ENOENT	Could not find an entry matching the ControlNo field in *EntryInfo*.
HPSS_ENOTREADY	Initialization has not completed.

Related Information

See also **gk_Close**, **gk_Open**, **gk_site_Open**, and **gk_site_OpenStats**.

Clients

Gatekeeper.

Notes

None.

Example Use

```
Log entrance of this routine to the Site Policy Logfile.
Lock the Request Cache.
Lookup the ControlNo in the Request Cache.
If not found, then
```

```
   It must have already been removed.
   NOTE:  There is a situation in which this'll happen.
          It could be that the GK was bounced while the BFS was handling
          the termination.  So, don't do anything rash.
   Unlock the Request Cache.
   Log an Error to the Site Policy Logfile.
   return.
If the Request was NOT an AuthorizedCaller request, then
   If the State of this request is GOOD, then
      Decrement the Host Count of the good opens pending.
      Decrement the User Count of the good opens pending.
      Decrement the global number of good opens pending.
   Else if the State of this request is RETRY, then
      Decrement the global number of retry opens pending.
      If the Request was in a retry state due to the host policy
             denying the open, then
         Decrement the Host's Count of the retry opens pending.
      If the Request was in a retry state due to the user policy
             denying the open, then
         Decrement the User's Count of the retry opens pending.
Delete the Request from the Request Cache.
Unlock the Request Cache.
Log exit of this routine to the Site Policy Logfile.
```

7.1.2 gk_site_Create

Purpose

This is an internal, site defined-and-implemented function that implements site policies which decide if a caller is authorized to create the file now, later, or not at all. This function is not an RPC. It is a call to a procedure in a shared library.

Syntax

```
signed32
gk_site_Create (
        gk_EntryInfo_t      EntryInfo,      /* IN */
        unsigned32          *WaitTimeP);    /* OUT */
```

Description

This function is written by the customer site. It is called by **gk_Create** whenever an HPSS file create occurs.

Parameters

EntryInfo	Information about the file being created.
WaitTimeP	A pointer to the number of seconds to wait before retrying a request.

Return Values

Upon successful completion, a value of zero (0) is returned which tells the Gatekeeper to continue with the create request. After the Gatekeeper has returned success to the caller, it expects **gk_site_CreateComplete** to be called when the file has been created or if an error occurs.

All non-zero return values will be treated as errors, thus the client will do the appropriate error handling and terminate the create request. Non-zero error values are described in the Error Conditions.

Error Conditions

The **gk_site_Create** routine is unsuccessful if any of the following are true:

HPSS_EACCES	Permission is denied.
HPSS_EAGAIN	Resources are temporarily unavailable.
HPSS_ENOTREADY	Initialization has not completed.
HPSS_EPERM	The operation is not permitted.
HPSS_ERETRY	The operation should be retried after the delay, in seconds, returned by the *WaitTimeP* parameter.
HPSS_ETHRESHOLD_DENY	This error code is returned to deny access due to exceeding a threshold. The BFS will return this error to the Client API. The Client API will map this error into EBUSY before returning the error code to the application.
HPSS_EUSER_DENY	This error code is returned to deny access to a particular user. The BFS will return this error to the Client API. The Client

API will map this error into EACCESS before returning the error code to the application.

Related Information

See also **gk_Create**, **gk_CreateComplete**, **gk_site_CreateComplete** and **gk_site_CreateStats**.

Clients

Gatekeeper.

Notes

Routine **gk_site_CreateStats** is called instead of **gk_site_Create** whenever an HPSS file create occurs from an authorized caller when authorized caller requests are monitored.

If the operation should be retried, HPSS_RETRY is returned and the number of seconds the caller should wait before retrying is returned in the *WaitTimeP* parameter. If the site does not specify the *WaitTimeP* parameter (returns 0), the default value from the Gatekeeper's specific configuration field DefaultWaitTime is used instead.

Example Use:

```
Log entrance of this routine to the Site Policy Logfile using the
     RequestId passed in the EntryInfo.
Initialize the WaitTimeP return parameter to 0.
If we're not monitoring for creates, then
   Log an ALARM Error to the Site Policy Logfile.
   Return, but don't return an error status -- just let the create con-
tinue.
If there is a no Site Policy file
   Log an Error to the Site Policy Logfile.
   Return good status.
Lock the Request Cache.
Find the Request in the Request Cache.
If not found, then
   Create a new Request Entry and add it to the Request Cache.
Else
   The Request is being retried.
If there exists a "Maximum Number of Creates Per Host Policy", then
   Note: Translating socket address into hostnames is expensive, so
         maintain a cache to translate HostAddrs to HostNames.
   Lock the HostAddrToHostName Cache.
   Lookup the EntryInfo.HostAddr in the HostAddrToHostName Cache.
   If it doesn't exist, then
      gethostbyaddr_h
     If Error returned by multithreaded safe get host by addr lookup, then
          Log an error to the Site Policy Logfile.
          Delete the Request from the Request Cache.
          Unlock all locks.
          Return HPSS_EFAULT.
      Create, initialize and add a new HostAddrToHostName cache entry.
```

```
      Save away the HostName.
      Unlock the HostAddrToHostName Cache.

      Lookup the HostName in the Host Entry Cache.
      Note: The Host Entry Cache is sharing the Request Cache Lock.
      If not found, then
         Create a new Host Entry and add it to the Host Cache.

      If the Host Entry's good creates pending >= Maximum Creates Per Host,
      then
         Mark this request as a RETRY due to host policy denial.
         Set Error to HPSS_ERETRY.
         If the Request was not originally a RETRY, then
            Increment the retry creates pending in the Host Entry.

   If there exists a "Maximum Number of Creates Per User Policy" AND Error
   wasn't set in the Host Policy code above, then

      Lookup the UserId/DCECellId pair in the User Entry Cache.
      Note: The UserId and DCECellId together will uniquely identify a
            particular user in a federated name space (cross cell)
            environment.
      Note: The User Entry Cache is sharing the Request Cache Lock.
      If not found, then
         Create a new User Entry and add it to the User Cache.

      If the User Entry's good creates pending >= Maximum Creates Per User,
      then
         Mark this request as a RETRY due to user policy denial.
         Set Error to HPSS_ERETRY.
         If the Request was not originally a RETRY, then
            Increment the retry creates pending in the User Entry.

   If we want to return HPSS_ERETRY, then
     If this originally wasn't a retry request (i.e. first delay of request),
      then
         Increment the global retry creates pending count.
      Increment the RetryCount associated with the Request Entry.
      Mark the Request Entry's state as RETRY.
      Compute the WaitTime corresponding to the number of seconds to wait
      before the Client API will retry this request.
   Else if no Error, then
      Increment the global good creates pending count.
      Mark the Request Entry's state as GOOD.
      If this request is being retried again, then
```

```
      Decrement the global retry creates pending count.
   If Host policy exists, then
      Decrement the Host Entry's good creates pending count.
     If this request was previously retried due to host policy denial, then
         Decrement the Host Entry's retry creates pending count.
   If User policy exists, then
      Decrement the User Entry's good creates pending count.
     If this request was previously retried due to user policy denial, then
         Decrement the Host User's retry creates pending count.

Unlock the Request Cache.
Log exit of this routine to the Site Policy Logfile.
Return the status.
```

7.1.3 gk_site_CreateComplete

Purpose

This is an internal, site defined-and-implemented function that is called by the Gatekeeper when a file create has completed. This function is not an RPC. It is a call to a procedure in a shared library.

Syntax

```
signed32
gk_site_CreateComplete (
        uuid_t                  ControlNo);        /* IN */
```

Description

This function is written by the customer site. It will be called by **gk_CreateComplete** while processing a file create completion of an HPSS file.

Parameters

ControlNo	Unique identifier for the created file which is now being completed. This number was generated by the Gatekeeper and passed into **gk_site_Create** or **gk_site_CreateStats**.

Return Values

Upon successful completion, a value of zero (0) is returned. Non-zero values are described in the Error Conditions.

Error Conditions

The **gk_site_CreateComplete** routine is unsuccessful if any of the following are true:

HPSS_EAGAIN	Resources are temporarily unavailable.
HPSS_ENOENT	Could not find an entry matching the ControlNo field in *EntryInfo*.
HPSS_ENOTREADY	Initialization has not completed.

Related Information

See also **gk_Create**, **gk_CreateComplete**, **gk_site_Create** and **gk_site_CreateStats**.

Clients

Gatekeeper.

Notes

None.

Example Use:

Please see section gk_site_Close.

7.1.4 gk_site_CreateStats

Purpose

This is an internal, site defined-and-implemented function that is called asynchronously by the Gatekeeper when an authorized caller is creating a file. It is only called when authorized caller requests and create requests are being monitored. This function is not an RPC. It is a call to a proce-

dure in a shared library.

Syntax

void
gk_site_CreateStats (
gk_EntryInfo_t *EntryInfo*); /* IN */

Description

This function is written by the customer site. It is called asynchronously by **gk_Create** whenever an HPSS file create occurs from an authorized caller when both create requests and authorized caller requests are monitored.

Parameters

EntryInfo	Information about the file being created.

Return Values

None.

Error Conditions

None.

Related Information

See also **gk_Create**, **gk_CreateComplete**, **gk_site_Create** and **gk_site_CreateComplete**.

Clients

Gatekeeper.

Notes

Routine **gk_site_CreateStats** is called instead of **gk_site_Create** whenever an HPSS file create occurs from an authorized caller when authorized caller requests are monitored.

The Gatekeeper will queue all asynchronous calls to the site interfaces so that they are processed in the order they are received. For example, if a site is monitoring authorized caller, create and open requests and an authorized caller create request is issued before an authorized caller open request, then the Gatekeeper will queue these requests so that **gk_site_CreateStats** is called before **gk_site_OpenStats**.

Example Use:

```
Log entrance of this routine to the Site Policy Logfile using the
     RequestId passed in the EntryInfo.
If we're not monitoring for creates, then
   Log an ALARM Error to the Site Policy Logfile.
   Return.

If there is a no Site Policy file
   Log an Error to the Site Policy Logfile.
   Return.

Lock the Request Cache.
Find the Request in the Request Cache.
If not found, then
```

```
      Create a new Request Entry and add it to the Request Cache.

   If there exists a "Maximum Number of Creates Per Host Policy", then
      Note: Translating socket address into hostnames is expensive, so
            maintain a cache to translate HostAddrs to HostNames.
      Lock the HostAddrToHostName Cache.
      Lookup the EntryInfo.HostAddr in the HostAddrToHostName Cache.
      If it doesn't exist, then
         gethostbyaddr_h
        If Error returned by multithreaded safe get host by addr lookup, then
             Log an error to the Site Policy Logfile.
             Delete the Request from the Request Cache.
             Unlock all locks.
             Return.
         Create, initialize and add a new HostAddrToHostName cache entry.
      Save away the HostName.
      Unlock the HostAddrToHostName Cache.

      Lookup the HostName in the Host Entry Cache.
      Note: Assume the Host Entry Cache is sharing the Request Cache Lock.
      If not found, then
         Create a new Host Entry and add it to the Host Cache.

   If there exists a "Maximum Number of Creates Per User Policy", then

      Lookup the UserId/DCECellId pair in the User Entry Cache.
      Note: The UserId and DCECellId together will uniquely identify a
            particular user in a federated name space (cross cell)
            environment.
      Note: The User Entry Cache is sharing the Request Cache Lock.
      If not found, then
         Create a new User Entry and add it to the User Cache.

   Mark the Request Entry's state as GOOD.
   Unlock the Request Cache.
   Log exit of this routine to the Site Policy Logfile.
   Return.
```

7.1.5 gk_site_GetMonitorTypes

Purpose

This is an internal, site defined-and-implemented function that is called by the Gatekeeper get the types of items being monitored. This function is not an RPC. It is a call to a procedure in a shared library.

Syntax

```
signed32
gk_site_GetMonitorTypes (
        u_signed64          MonitorTypeBitsP); /* OUT */
```

Description

This interface will lookup the local site policies to determine which request types are being monitored. It will then return a bitvector of items being monitored. The Bitfile Server will call **gk_GetMonitorTypes** when (re)connecting to the Gatekeeper to learn which request types are being monitored so that it can call the Gatekeeper appropriately. The Gatekeeper will call **gk_site_GetMonitorTypes**. Currently the following request types can be monitored: authorized caller, create, open, and stage.

It is important that the Site Interfaces return a status in a timely fashion. Create, open, and stage requests from DFS, XFS, NFS, and MPS (authorized callers) are timing sensitive, thus the Site Interfaces won't be permitted to delay or stop these requests, however the Site Interfaces may choose to be involved in keeping statistics on these requests by monitoring requests from authorized callers.

Parameters

MonitorTypeBitsP — A bit vector (0 origin array of bits) in which the appropriate bit is set (on) for each request type that is currently being monitored. Request types index are:

GK_MONITOR_AUTHORIZED_CALLER	0
GK_MONITOR_CREATE	1
GK_MONITOR_OPEN	2
GK_MONITOR_STAGE	3

Return Values

Upon successful completion, a value of zero (0) is returned. Non-zero values are described in the Error Conditions.

Error Conditions

The **gk_site_GetMonitorTypes** routine is unsuccessful if any of the following are true:

HPSS_EAGAIN	Resources are temporarily unavailable.

Related Information

See also **gk_GetMonitorTypes**.

Clients

Gatekeeper

Notes

The following table describes which Site Interface will be called for each request type when authorized caller requests are monitored.

Request Type	Authorized Caller	All Other Users
Create	gk_site_CreateStats	gk_site_Create
Open	gk_site_OpenStats	gk_site_Open
Stage	gk_site_StageStats	gk_site_Stage

Example Use:

```
Log entrance of this routine to the Site Policy Logfile.
Set the bits corresponding to what we're monitoring in the MonitorType-
BitsP.
(e.g. *MonitorTypeBitsP = orbit64m(*MonitorTypeBitsP, GK_MONITOR_OPEN))
Log exit of this routine to the Site Policy Logfile.
Return status.
```

7.1.6 gk_site_Init

Purpose

This is an internal, site defined and implemented function that is called when the Gatekeeper is (re)initialized. This function is used to do whatever initialization is needed by the site module. This function is not an RPC. It is a call to a procedure in a shared library.

Syntax

signed32
gk_site_Init (
    char **SitePolicyPathNameP*); /* IN */

Description

This function is written by the customer site. It will be called by the Gatekeeper's internal initialization routine whenever the Gatekeeper initializes or reinitializes.

Parameters

SitePolicyPathNameP	A pointer to the path name of the file where the site policy is stored. This path name is defined to be HPSS_MAX_DCE_NAME bytes long which is 128 bytes.

Return Values

Upon successful completion, a value of zero (0) is returned. Non-zero values are described in the Error Conditions.

Error Conditions

The **gk_site_Init** routine is unsuccessful if any of the following are true:

HPSS_EAGAIN	Resources are temporarily unavailable.

Related Information

None.

Clients

Gatekeeper.

Notes

If an error value is returned, the Gatekeeper Server will not initialize.

Example Use:

```
Initialize mutex locks, global variables, etc.
If the passed in SitePolicyPathNameP is non null and not nil, then
   Read the site policy file looking for things like:
   - The types of requests we're monitoring (authorized caller, creates,
     opens, stages)
   - Pathname to a log file.
   - Maximum size of the log file.
   - Debug settings.
   - Pathname to a dump file.
   - Maximum number of creates per host.
   - Maximum number of creates per user.
   - Maximum number of opens per host.
   - Maximum number of opens per user.
   - Maximum number of stages per host. (Note: Needs to be greater than
     the maximum number of opens per host.)
   - Maximum number of stages per user. (Note: Needs to be greater than
     the maximum number of opens per user.)
Open and initialize the logfile (if one is configured).
Log exit of this routine to the Site Policy Logfile.
Return status.
```

7.1.7 gk_site_Open

Purpose

This is an internal, site defined-and-implemented function that implements site policies which decide if a caller is authorized to open the file now, later, or not at all. This function is not an RPC. It is a call to a procedure in a shared library.

Syntax

```
signed32
gk_site_Open (
        gk_EntryInfo_t      EntryInfo,        /* IN */
        unsigned32          *WaitTimeP);      /* OUT */
```

Description

This function is written by the customer site. It is called by **gk_Open** whenever an HPSS file open occurs.

Parameters

EntryInfo	Information about the file being opened.
WaitTimeP	A pointer to the number of seconds to wait before retrying a request.

Return Values

Upon successful completion, a value of zero (0) is returned which tells the Gatekeeper to continue with the open request. After the Gatekeeper has returned success to the caller, it expects **gk_site_Close** to be called when the file has been closed or if an error occurred

All non-zero return values will be treated as errors, thus the client will do the appropriate error handling and terminate the open request. Non-zero error values are described in the Error Conditions.

Error Conditions

The **gk_site_Open** routine is unsuccessful if any of the following are true:

HPSS_EACCES	Permission is denied.
HPSS_EAGAIN	Resources are temporarily unavailable.
HPSS_ENOTREADY	Initialization has not completed.
HPSS_EPERM	The operation is not permitted.
HPSS_ERETRY	The operation should be retried after the delay, in seconds, returned by the *WaitTimeP* parameter.
HPSS_ETHRESHOLD_DENY	This error code is returned to deny access due to exceeding a threshold. The BFS will return this error to the Client API. The Client API will map this error into EBUSY before returning the error code to the application.
HPSS_EUSER_DENY	This error code is returned to deny access to a particular user. The BFS will return this error to the Client API. The Client

API will map this error into EACCESS before returning the error code to the application.

Related Information

See also **gk_Close**, **gk_Open**, **gk_site_Close**, and **gk_site_OpenStats**.

Clients

Gatekeeper.

Notes

Routine **gk_site_OpenStats** is called instead of **gk_site_Open** whenever an HPSS file open occurs from an authorized caller when authorized caller requests are monitored.

If the operation should be retried, HPSS_RETRY is returned and the number of seconds the caller should wait before retrying is returned in the *WaitTimeP* parameter. If the site does not specify the *WaitTimeP* parameter (returns 0), the default value from the Gatekeeper's specific configuration field DefaultWaitTime is used instead.

Example Use:

Please see section gk_site_Create.

7.1.8 gk_site_OpenStats

Purpose

This is an internal, site defined-and-implemented function that is called asynchronously by the Gatekeeper when an authorized caller is opening a file. It is only called when authorized caller requests and open requests are being monitored. This function is not an RPC. It is a call to a procedure in a shared library.

Syntax

```
void
gk_site_OpenStats (
        gk_EntryInfo_t          EntryInfo);          /* IN */
```

Description

This function is written by the customer site. It is called by **gk_Open** whenever an HPSS file open occurs from an authorized caller when both open requests and authorized caller requests are monitored.

Parameters

EntryInfo	Information about the file being opened.

Return Values

None.

Error Conditions

None.

Related Information

See also **gk_Close**, **gk_Open**, **gk_site_Close**, and **gk_site_Open**.

Clients

Gatekeeper.

Notes

Routine **gk_site_OpenStats** is called instead of **gk_site_Open** whenever an HPSS file open occurs from an authorized caller when authorized caller requests are monitored.

The Gatekeeper will queue all asynchronous calls to the site interfaces so that they are processed in the order they are received. For example, if a site is monitoring authorized caller, create and open requests and an authorized caller create request is issued before an authorized caller open request, then the Gatekeeper will queue these requests so that **gk_site_CreateStats** is called before **gk_site_OpenStats**.

Example Use:

Please see section gk_site_CreateStats.

7.1.9 gk_site_PassThru

Purpose

This is an internal, site defined-and-implemented function that is called by the Gatekeeper to pass information to the site interface. This function is not an RPC. It is a call to a procedure in a shared library.

Syntax

```
signed32
gk_site_PassThru (
        u_signed64          Param64,          /* IN */
        unsigned char       *ParamString);    /* IN */
```

Description

This function is written by the customer site. It will be used to pass information into the site interface. The site interface can define what these parameters mean.

This function will be called by **gk_PassThru**.

Parameters

Param64	A 64 bit parameter.
ParamString	NULL-terminated string parameter.

Return Values

Upon successful completion, a value of zero (0) is returned. Non-zero values are described in the Error Conditions.

Error Conditions

The **gk_site_PassThru** routine is unsuccessful if any of the following are true:

HPSS_EAGAIN	Resources are temporarily unavailable.
HPSS_ENOTREADY	Initialization has not completed.

Related Information

See also **gk_PassThru**.

Clients

Gatekeeper.

Notes

The site may want to define bits in *Param64* to correspond to special site actions. The *ParamString* may be used in conjunction with a bit(s) in *Param64* or for some other use. For example, if bit position 0 is ON in *Param64* then dump the current state to some file name passed in *ParamString*. The site will need to write its own client to use this interface (i.e. BFS and SSM will not use this interface).

Example Use:

```
Log entrance of this routine to the Site Policy Logfile.
If Param64 is zero, then
   Log exit of this routine to the Site Policy Logfile.
   Return status.
Verify bits passed into Param64 are valid options.
If an unsupported bit is set in Param64, then
   Log an exit error to the Site Policy Logfile.
   Return status.
If turned on the DumpState bit, then
   If ParamString is null or nil, then
     Dump the state of each cache and all the other good information into
      the dump file.
   Else
     Dump the state of each cache and all the other good information into
      the pathname stored in ParamString.
Log exit of this routine to the Site Policy Logfile.
Return status.
```

7.1.10 gk_site_ReadSitePolicy

Purpose

This is an internal, site defined-and-implemented function that is called by the Gatekeeper to inform the site interface that the site policy file needs to be read. This function is not an RPC. It is a call to a procedure in a shared library.

Syntax

signed32
gk_site_ReadSitePolicy (void);

Description

This function is written by the customer site. It will be used to inform the site interface that the site policy file has been modified. This will allow the site interface to re-read the site policy file (if defined) and enforce the new policy.

If the monitoring types are changed, the Gatekeeper and Bitfile Server will need to be restarted in order for the Bitfile Server to learn about it. Therefore, ignore changes in monitor type policy.

This function will be called by **gk_ReadSitePolicy**.

Parameters

None.

Return Values

Upon successful completion, a value of zero (0) is returned. Non-zero values are described in the Error Conditions.

Error Conditions

The **gk_site_ReadSitePolicy** routine is unsuccessful if any of the following are true:

HPSS_EAGAIN	Resources are temporarily unavailable.
HPSS_ENOTREADY	Initialization has not completed.

Related Information

See also **gk_ReadSitePolicy,** and **gk_GetMonitorTypes**.

Clients

Gatekeeper.

Notes

If the monitoring types are changed, the Gatekeeper and Bitfile Server will need to be restarted in order for the Bitfile Server to learn about it. Therefore, ignore changes in monitor type policy.

Example Use:

```
Log entrance of this routine to the Site Policy Logfile.
If a site policy file exists, then
   Read the site policy file looking for things like:
   - The types of requests we're monitoring (authorized caller, creates,
     opens, stages)
   - Pathname to a log file.
```

```
   - Maximum size of the log file.
   - Debug settings.
   - Pathname to a dump file.
   - Maximum number of creates per host.
   - Maximum number of creates per user.
   - Maximum number of opens per host.
   - Maximum number of opens per user.
   - Maximum number of stages per host. (Note: Needs to be greater than
     the maximum number of opens per host.)
   - Maximum number of stages per user. (Note: Needs to be greater than
     the maximum number of opens per user.)
Close the logfile (if one is already opened).
Open the possibly new logfile (if one is configured).
Log exit of this routine to the Site Policy Logfile.
Return status.
```

7.1.11 gk_site_Shutdown

Purpose

This is an internal, site defined-and-implemented function that is called when the Gatekeeper is shutdown. This function is used to do whatever shutdown and cleanup is needed by the site module. This function is not an RPC. It is a call to a procedure in a shared library.

Syntax

signed32
gk_site_Shutdown (void);

Description

This function is written by the customer site. It will be called by the Gatekeeper's internal shutdown routine whenever the Gatekeeper is issued a shutdown request.

Parameters

None.

Return Values

Upon successful completion, a value of zero (0) is returned. Non-zero values are described in the Error Conditions.

Error Conditions

The **gk_site_Shutdown** routine is unsuccessful if any of the following are true:

HPSS_EAGAIN	Resources are temporarily unavailable.

Related Information

None.

Clients

Gatekeeper.

Notes

The Gatekeeper Server will not call this routine if it is issued a halt request or is crashing due to some critical problem.

Example Use:

```
Log entrance of this routine to the Site Policy Logfile.
Flush and close the logfile (if one is already opened).
Return status.
```

7.1.12 gk_site_Stage

Purpose

This is an internal, site defined-and-implemented function that implements site policies which decide if a caller is authorized to stage the file now, later, or not at all. This function not an RPC. It is a call to a procedure in a shared library.

Syntax

```
signed32
gk_site_Stage (
        gk_EntryInfo_t      EntryInfo,      /* IN */
        unsigned32          *WaitTimeP);    /* OUT */
```

Description

This function is written by the customer site. It is called by **gk_Stage** whenever an HPSS file stage occurs.

Parameters

EntryInfo	Information about the file being staged.
WaitTimeP	A pointer to the number of seconds to wait before retrying a request.

Return Values

Upon successful completion, a value of zero (0) is returned which tells the Gatekeeper to continue with the stage request. After the Gatekeeper has returned success to the caller, it expects **gk_site_StageComplete** to be called when the file has been staged or if an error occurred

All non-zero return values will be treated as errors, thus the client will do the appropriate error handling and terminate the stage request. Non-zero error values are described in the Error Conditions.

Error Conditions

The **gk_site_Stage** routine is unsuccessful if any of the following are true:

HPSS_EACCES	Permission is denied.
HPSS_EAGAIN	Resources are temporarily unavailable.
HPSS_ENOTREADY	Initialization has not completed.
HPSS_EPERM	The operation is not permitted.
HPSS_ERETRY	The operation should be retried after the delay, in seconds, returned by the *WaitTimeP* parameter.
HPSS_ETHRESHOLD_DENY	This error code is returned to deny access due to exceeding a threshold. The BFS will return this error to the Client API. The Client API will map this error into EBUSY before returning the error code to the application.
HPSS_EUSER_DENY	This error code is returned to deny access to a particular user. The BFS will return this error to the Client API. The Client

API will map this error into EACCESS before returning the error code to the application.

Related Information

See also **gk_site_StageComplete**, **gk_site_StageStats**, **gk_Stage**, and **gk_StageComplete**.

Clients

Gatekeeper.

Notes

Routine **gk_site_StageStats** is called instead of **gk_site_Stage** whenever an HPSS file stage occurs from an authorized caller when authorized caller requests are monitored.

If the operation should be retried, HPSS_RETRY is returned and the number of seconds the caller should wait before retrying is returned in the *WaitTimeP* parameter. If the site does not specify the *WaitTimeP* parameter (returns 0), the default value from the Gatekeeper's specific configuration field DefaultWaitTime is used instead.

Example Use:

Please see section gk_site_Create.

7.1.13 gk_site_StageComplete

Purpose

This is an internal, site defined-and-implemented function that is called by the Gatekeeper when a file stage has completed. This function is not an RPC. It is a call to a procedure in a shared library.

Syntax

```
signed32
gk_site_StageComplete (
        uuid_t              ControlNo);        /* IN */
```

Description

This function is written by the customer site. It will be called by **gk_StageComplete** while processing a file stage completion of an HPSS file.

Parameters

ControlNo	Unique identifier for the staged file which is now being completed. This number was generated by the Gatekeeper and passed into **gk_site_Stage** or **gk_site_StageStats**.

Return Values

Upon successful completion, a value of zero (0) is returned. Non-zero values are described in the Error Conditions.

Error Conditions

The **gk_site_StageComplete** routine is unsuccessful if any of the following are true:

HPSS_EAGAIN	Resources are temporarily unavailable.
HPSS_ENOENT	Could not find an entry matching the ControlNo field in *EntryInfo*.

HPSS_ENOTREADY	Initialization has not completed.

Related Information

See also **gk_site_Stage**, **gk_site_StageStats**, **gk_Stage**, and **gk_StageComplete**.

Clients

Gatekeeper.

Notes

None.

Example Use:

Please see section gk_site_Close.

7.1.14 gk_site_StageStats

Purpose

This is an internal, site defined-and-implemented function that is called asynchronously by the Gatekeeper when an authorized caller is staging a file. It is only called when authorized caller requests and stage requests are being monitored. This function is not an RPC. It is a call to a procedure in a shared library.

Syntax

```
void
gk_site_StageStats (
        gk_EntryInfo_t      EntryInfo);      /* IN */
```

Description

This function is written by the customer site. It is called by **gk_Stage** whenever an HPSS file stage occurs from an authorized caller when both stage requests and authorized caller requests are monitored.

Parameters

EntryInfo	Information about the file being staged.

Return Values

None.

Error Conditions

None.

Related Information

See also **gk_site_Stage**, **gk_site_StageComplete**, **gk_Stage**, and **gk_StageComplete**.

Clients

Gatekeeper.

Notes

Routine **gk_site_StageStats** is called instead of **gk_site_Stage** whenever an HPSS file stage occurs from an authorized caller when authorized caller requests are monitored.

The Gatekeeper will queue all asynchronous calls to the site interfaces so that they are processed

in the order they are received. For example, if a site is monitoring authorized caller, create and open requests and an authorized caller create request is issued before an authorized caller open request, then the Gatekeeper will queue these requests so that **gk_site_CreateStats** is called before **gk_site_OpenStats**.

Example Use:

Please see section gk_site_CreateStats.

7.2 Account Validation Site Interface

This section describes the site customizable parts of the Account Validation APIs. These routines are stored in a shared library to ease customization.

7.2.1 av_site_AcctIdxToName

Purpose

Customizable routine to convert an account index to an account name

Syntax

```
signed32
av_site_AcctIdxToName(
        handle_t        Binding,        /* IN */
        hpss_reqid_t    RequestId,      /* IN */
        unsigned32      CellId,         /* IN */
        unsigned32      Uid,            /* IN */
        unsigned32      Gid,            /* IN */
        unsigned32      Flags,          /* IN */
        acct_rec_t      Acct,           /* IN */
        char            *AcctName,      /* OUT */
        unsigned32      *OkToCache      /* OUT */
        );
```

Description

This site customizable routine converts from an account index to an account name. By default, this routine is nothing more than a skeleton that returns HPSS_EUSEDEFAULT immediately.

Parameters

Inputs

Binding	Binding handle to this server.
RequestId	Request Id to use when logging messages for this request.
CellId	Cell id of this account.
Uid	User id (UID) of user.
Gid	Group id (GID) of user.
Flags	May be zero or more of the following OR'd together: ACCT_FLAGS_VALID_INDEX – Verify that the specified user is allowed to use this account index.
Acct	Account Index to translate.

Outputs

AcctName	Pointer to returned account name.
OkToCache	True if Account Validation Library should cache the result. See Notes for more information.

Return Values

Upon successful completion, zero (0) is returned. A non-zero value indicates an error condition.

Error Conditions

HPSS_EUSEDEFAULT	Tells the caller to use the default built-in behavior for this routine.

Other suggested error codes to use are:

HPSS_EBUSY	The server is busy. The request should be retried later.
HPSS_ENOENT	The account in Acct does not exist. AcctName is set to the string form of Acct. If ACCT_FLAGS_VALID_INDEX is set in the Flags parameter, you should return HPSS_EPERM instead..
HPSS_EPERM	The specified user is not allowed to use that account index.
HPSS_ESYSTEM	This routine received an internal error. Usually a site customization problem.

Related Information

av_AcctIdxToName

Clients

Client API via Gatekeeper

Notes

This routine can be customized by a site. It is the site's responsibility to make sure that consistent results are returned and performance is not adversely impacted.

The OkToCache argument should normally be TRUE. If a site customizes this routine it should only set this argument to FALSE if the returned information can change over time and should not be cached by the client.

Example Usage

This example is the default behavior of the av_AcctIdxToName routine. See the implementation of that routine, if available, for more details.

```
av_AcctIdxToName() {
if site style accounting {
 if Acct == ACCT_REC_DEFAULT
     lookup user default acct;
       else lookup account Acct;
 if (not found) {
     if Flags & ACCT_FLAGS_VALID_INDEX
         error = HPSS_EPERM;
            else error = HPSS_ENOENT;
       }
else if (unix style accounting)
 if (Acct == ACCT_REC_DEFAULT) Acct = Uid;
 if (Acct != Uid && Flags & ACCT_FLAGS_VALID_INDEX)
     error = HPSS_EPERM;
 else {
```

```
        convert Cell,Acct into name
        if not found && Flags & ACCT_FLAGS_VALID_INDEX
            error = HPSS_EPERM;
    }
  }
  return error;
}
```

7.2.2 av_site_AcctNameToIdx

Purpose

Customizable routine to convert an account name to an account index

Syntax

```
signed32
av_site_AcctNameToIdx(
        handle_t        Binding,        /* IN */
        hpss_reqid_t    RequestId,      /* IN */
        unsigned32      CellId,         /* IN */
        unsigned32      Uid,            /* IN */
        unsigned32      Gid,            /* IN */
        unsigned32      Flags,          /* IN */
        char            *InAcctName,    /* IN */
        char            *OutAcctName,   /* OUT */
        acct_rec_t      *Acct,          /* OUT */
        unsigned32      *OkToCache      /* OUT */
        );
```

Description

This site customizable routine converts from an account name to an account index. By default, this routine is nothing more than a skeleton that returns HPSS_EUSEDEFAULT immediately.

Parameters

Inputs

Binding — Binding handle to this server.

RequestId — Request Id to use when logging messages for this request.

CellId — Cell identifier of user.

Uid — User id (UID) of user.

Gid — Group id (GID) of user.

Flags — May be zero or more of the following OR'd together:

ACCT_FLAGS_VALID_INDEX – Verify that the specified user is allowed to use this account index.

InAcctName — Optional pointer to account name to translate. If this argument is NULL or points to an empty string, information about the user's default account index is returned.

Outputs

OutAcctName — Optional pointer to returned account name. The name of the user's default account is returned if the InAcctName parameter is NULL or points to an empty string. Otherwise, the value of InAcctName is returned. Note the name of the

	default account for UNIX Style accounting is the empty string.
Acct	Pointer to the returned account index.
OkToCache	True if Account Validation Library should cache the result. See Notes for more information.

Return Values

Upon successful completion, zero (0) is returned. A non-zero value indicates an error condition.

Error Conditions

HPSS_EUSEDEFAULT	Tells the caller to use the default built-in behavior for this routine.

Other suggested error codes to use are:

HPSS_EBUSY	The server is busy. The request should be retried later.
HPSS_ENOENT	The account specified does not exist. Acct is set to the integer form of AcctName. If ACCT_FLAGS_VALID_INDEX is set in the Flags parameter, you should return HPSS_EPERM instead.
HPSS_EPERM	The specified user is not allowed to use that account name.
HPSS_ESYSTEM	This routine received an internal error. Usually a site customization problem.

Related Information

av_AcctNameToIdx

Clients

Client API via Gatekeeper

Notes

This routine can be customized by a site. It is the site's responsibility to make sure that consistent results are returned and performance is not adversely impacted.

The OkToCache argument should normally be TRUE. If a site customizes this routine it should only set this argument to FALSE if the returned information can change over time and should not be cached by the client.

Example Usage

This example is the default behavior of the av_AcctNameToIdx See the implementation of that routine, if available, for more details.

```
av_AcctNameToIdx() {
lookup_default = (InAcctName is NULL or empty);
if site style accounting {
 if lookup_default
     read default account for this user from metadata
     if (not found and requiring default account)
```

```
            return HPSS_EPERM;
        else if Flags & ACCT_FLAGS_VALID_INDEX
            read this user's account by name
        else
            read account metadata by name
    } else if unix style accounting {
        if (lookup_default)
            *Acct = Uid;
        else {
            try to convert AcctName to number
            if successful, *Acct = number;
        }
    }

    if we set *Acct above {
        error = lookup user from uid
        if not found and require default account
            error = HPSS_EPERM;
    } else {
        error = lookup user from account name
        if (not found and Flags & ACCT_FLAGS_VALID_INDEX
            error = HPSS_EPERM;
    }
    return error;
}
```

7.2.3 *av_site_Initialize*

Purpose

Customizable routine to perform initialization of the Account Validation APIs.

Syntax

```
signed32
av_site_Initialize(
        acct_config_t        *AcctPolicy        /* IN */
        );
```

Description

This routine initializes the site customized APIs which receive client requests. Sites should initialize any common state shared between the various site routines.

Parameters

Input

AcctPolicy	Pointer to accounting policy metadata record.

Return Values

Upon successful completion, zero (0) is returned. A non-zero value indicates an error condition.

Error Conditions

Suggested error codes to use:

HPSS_SYSTEM	Site code cannot initialize properly.
HPSS_ENOMEM	Out of memory.

Related Information

av_Initialize

Clients

Gatekeeper

Notes

This routine can be customized by a site. It is the site's responsibility to make sure that consistent results are returned and performance is not adversely impacted. Any error returned from this routine is considered fatal and will keep the server from starting up properly.

Example Usage

```
av_site_Initialize() {
allocate space for local site policy information;
read in local site policy information;
}
```

7.2.4 av_site_Shutdown

Purpose

Customizable routine to shutdown the interface to the Account Validation APIs.

Syntax

signed32
av_site_Shutdown(void);

Description

This routine cleans up any state created by the customizable site routines.

Parameters

None

Return Values

Upon successful completion, zero (0) is returned. A non-zero value indicates an error condition.

Error Conditions

Suggested error codes to use:

HPSS_EINVAL	The APIs have not been initialized so there is nothing to shutdown.

Related Information

None

Clients

Gatekeeper

Notes

This routine can be customized by a site. It is the site's responsibility to make sure that consistent results are returned and performance is not adversely impacted. Any error returned from this routine is logged but otherwise ignored.

Example Usage

```
av_site_Shutdown() {
deallocate space for local site policy information;
}
```

7.2.5 *av_site_ValidateAccount*

Purpose

Customizable routine to validate that a user is allowed to use an account index.

Syntax

```
signed32
av_site_ValidateAccount(
        handle_t        Binding,        /* IN */
        hpss_reqid_t    RequestId,      /* IN */
        unsigned32      CellId,         /* IN */
        unsigned32      Uid,            /* IN */
        acct_rec_t      Acct,           /* IN */
        unsigned32      Flags,          /* IN */
        acct_rec_t      *ParentAcct,    /* IN */
        acct_rec_t      *OutAcct,       /* OUT */
        unsigned32      *OkToCache      /* OUT */
        );
```

Description

This routine verifies that Acct is a valid account index to use for the user specified by the Uid and CellId given. By default, this routine is nothing more than a skeleton that returns HPSS_EUSEDEFAULT immediately. See av_ValidateAccount for the default behavior of this routine.

Parameters

Inputs

Binding — Binding handle to this server.

RequestId — Request Id to use when logging messages for this request.

CellId — Cell identifier of user.

Uid — User id (UID) of user.

Acct — Account Index to validate. . If this is ACCT_REC_DEFAULT, the specified user's default account index, if any, is used instead.

Flags — One or more of the following OR'd together:

ACCT_VALIDATE_FLAGS_CREATING – Caller is performing a file or directory creation operation.

ACCT_VALIDATE_FLAGS_AUTH_CALLER – Request is being made on the behalf of an authorized caller.

ParentAcct — Pointer to parent account index.

Outputs

OutAcct	Account Index to use.
OkToCache	True if Account Validation Library should cache the result. See Notes for more information.

Return Values

Upon successful completion, zero (0) is returned signifying that the account is valid..

A non-zero value indicates an error condition.

Error Conditions

HPSS_EUSEDEFAULT	Tells the caller to use the default built-in behavior for this routine.

Other suggested error codes to use are:

HPSS_EBUSY	The server is busy. The request should be retried later.
HPSS_EPERM	The specified user is not allowed to use that account index.
HPSS_ESYSTEM	This routine received an internal error. Usually a site customization problem.

Related Information

av_ValidateAccount

Clients

Bitfile Server and Name Server via Gatekeeper

Notes

This routine can be customized by a site. It is the site's responsibility to make sure that consistent results are returned and performance is not adversely impacted.

The OkToCache argument should normally be TRUE. If a site customizes this routine it should only set this argument to FALSE if the returned information can change over time and should not be cached by the client.

If this request is being performed on the behalf of an authorized caller, it is generally not appropriate to return a permission error from this routine. See av_ValidateAccount.

Example Usage

This example is the default behavior of the av_ValidateAccount See the implementation of that routine, if available, for more details.

```
av_ValidateAccount() {
if site style and creating a file and account inheritance is on {
 if (parent account supplied and
     (Acct is ACCT_REC_DEFAULT or parent account)) {
     return with parent account;
 } else if (no parent account supplied and
            Acct != ACCT_REC_DEFAULT) {
     /* assume caller (BFS) has determined parent already */
```

```
        return with Acct;
    }
   }
   if (user is superuser)
    return with Acct;
   if (user is “nobody”) {
    if auth caller
        return with ACCT_REC_DEFAULT
    return HPSS_EPERM;
   }
   if site style accounting {
    Read the account from metadata.
    If (not found and Acct == ACCT_REC_DEFAULT and auth caller) {
        Create default account for this user;
    } else if not found {
        error = HPSS_EPERM;
    }
   } else if unix style {
    if Acct == UID or ACCT_REC_DEFAULT or auth caller
        return with Uid;
    error = HPSS_EPERM;
   }
   return error;
  }
```

7.2.6 *av_site_ValidateChacct*

Purpose

Determine the account code to use when a file's account index.

Syntax

```
signed32
av_site_ValidateChacct(
        handle_t        Binding,        /* IN */
        hpss_reqid_t    RequestId,      /* IN */
        unsigned32      UserCellId,     /* IN */
        unsigned32      UserUid,        /* IN */
        unsigned32      FileCellId,     /* IN */
        unsigned32      FileUid,        /* IN */
        unsigned32      FileGid,        /* IN */
        acct_rec_t      OldAcct,        /* IN */
        acct_rec_t      NewAcct,        /* IN */
        acct_rec_t      *OutAcct,       /* OUT */
        unsigned32      *OkToCache      /* OUT */
        );
```

Description

This routine determines what account index should be used when a file or directory's account index changes. By default, this routine is nothing more than a skeleton that returns HPSS_EUSEDEFAULT immediately. See av_ValidateChacct for the default behavior of this routine.

Parameters

Inputs

Binding	Binding handle to this server.
RequestId	Request Id to use when logging messages for this request.
UserCellId	Cell identifier of user performing the operation.
UserUid	User id (UID) of user performing the operation.
FileCellId	Cell identifier of file's owner.
FileUid	User id (UID) of file's owner.
FileGid	User id (UID) of file's owner.
OldAcct	Old Account Index of file.
NewAcct	New Account Index to put onto the file. If this is ACCT_REC_DEFAULT then the user's default account index is used.

Outputs

OutAcct	Returned account index to use.
OkToCache	True if Account Validation Library should cache the result. See Notes for more information.

Return Values

Upon successful completion, zero (0) is returned. A non-zero value indicates an error condition.

Error Conditions

HPSS_EUSEDEFAULT	Tells the caller to use the default built-in behavior for this routine.

Other suggested error codes to use are:

HPSS_EBUSY	The server is busy. The request should be retried later.
HPSS_EPERM	The account specified may not be used by this user.
HPSS_ESYSTEM	This routine received an internal error. Usually a site customization problem.

Related Information

av_site_ValidateChacct

Clients

Client API via Gatekeeper

Notes

This routine can be customized by a site. It is the site's responsibility to make sure that consistent results are returned and performance is not adversely impacted.

The OkToCache argument should normally be TRUE. If a site customizes this routine it should only set this argument to FALSE if the returned information can change over time and should not be cached by the client.

Example Usage

This example is the default behavior of the av_ValidateChacct See the implementation of that routine, if available, for more details.

```
av_ValidateChacct() {
if user is "nobody", return with *OutAcct = ACCT_REC_DEFAULT;
if site style accounting {
 if user is superuser and Acct != ACCT_REC_DEFAULT
     return with NewAcct;
 error = read user account from metadata
 if not found {
     if NewAcct == ACCT_REC_DEFAULT, return 0
     else return HPSS_EPERM;
 }
} else if unix style accounting {
 if NewAcct is FileUid or ACCT_REC_DEFAULT
```

```
        return with *OutAcct = FileUid;
    error = HPSS_EPERM;
  }
  return error;
}
```

7.2.7 *av_site_ValidateChown*

Purpose

Determine the account code to use when a file's ownership or group changes.

Syntax

```
signed32
av_site_ValidateChown(
        handle_t        Binding,        /* IN */
        hpss_reqid_t    RequestId,      /* IN */
        unsigned32      OldCellId,      /* IN */
        unsigned32      OldUid,         /* IN */
        unsigned32      OldGid,         /* IN */
        acct_rec_t      OldAcct,        /* IN */
        unsigned32      NewCellId,      /* IN */
        unsigned32      NewUid,         /* IN */
        unsigned32      NewGid,         /* IN */
        acct_rec_t      SessionAcct,    /* IN */
        acct_rec_t      *OutAcct,       /* OUT */
        unsigned32      *OkToCache      /* OUT */
        );
```

Description

This routine determines what account index should be used when a file or directory's ownership changes or its group changes. By default, this routine is nothing more than a skeleton that returns HPSS_EUSEDEFAULT immediately. See av_ValidateChown for the default behavior of this routine.

Parameters

Inputs

Binding	Binding handle to this server.
RequestId	Request Id to use when logging messages for this request.
OldCellId	Cell identifier of file's owner.
OldUid	User id (UID) of file's owner.
OldGid	Group id (GID) of file.
OldAcct	Account Index of file.
NewCellId	New Cell identifier of file's owner.
NewUid	New User id (UID) of file's owner.
NewGid	New Group id (GID) of file.
SessionAcct	User's current session account index.

Outputs

OutAcct	Returned account index to use.
OkToCache	True if Account Validation Library should cache the result. See Notes for more information.

Return Values

Upon successful completion, zero (0) is returned. A non-zero value indicates an error condition.

Error Conditions

HPSS_EUSEDEFAULT	Tells the caller to use the default built-in behavior for this routine.

Other suggested error codes to use are:

HPSS_EBUSY	The server is busy. The request should be retried later.
HPSS_EPERM	The account specified may not be used by this user.
HPSS_ESYSTEM	This routine received an internal error. Usually a site customization problem.

Related Information

av_site_ValidateChown

Clients

Client API via Gatekeeper

Notes

This routine can be customized by a site. It is the site's responsibility to make sure that consistent results are returned and performance is not adversely impacted.

The OkToCache argument should normally be TRUE. If a site customizes this routine it should only set this argument to FALSE if the returned information can change over time and should not be cached by the client.

Example Usage

This example is the default behavior of the av_ValidateChown See the implementation of that routine, if available, for more details.

```
av_ValidateChown() {
if user is "nobody", return with *OutAcct = ACCT_REC_DEFAULT;
if site style accounting {
 error = read user account from metadata
 if not found {
     error = 0;
     *OutAcct = ACCT_REC_DEFAULT;
 }
} else if unix style accounting {
 *OutAcct = NewUid;
}
```

```
return error;
}
```

7.2.8 av_site_ValidateCreate

Purpose

Customizable routine to determine the account code to use when creating a file or directory.

Syntax

```
signed32
av_site_ValidateCreate (
        handle_t        Binding,        /* IN */
        hpss_reqid_t    RequestId,      /* IN */
        unsigned32      CellId,         /* IN */
        unsigned32      Uid,            /* IN */
        unsigned32      Gid,            /* IN */
        acct_rec_t      Acct,           /* IN */
        acct_rec_t      ParentAcct,     /* IN */
        acct_rec_t      *OutAcct,       /* OUT */
        unsigned32      *OkToCache      /* OUT */
        );
```

Description

This routine determines what account index should be used when a file or directory is created. By default, this routine is nothing more than a skeleton that returns HPSS_EUSEDEFAULT immediately. See av_ValidateCreate for the default behavior of this routine.

Parameters

Inputs

Binding	Binding handle to this server.
RequestId	Request Id to use when logging messages for this request.
CellId	Cell identifier of new file's owner.
Uid	User id (UID) of new file's owner.
Gid	Group id (GID) of new file.
Acct	Current session account index.
ParentAcct	Account Index of parent directory.

Outputs

OutAcct	Returned account index to use.
OkToCache	True if Account Validation Library should cache the result. See Notes for more information.

Return Values

Upon successful completion, zero (0) is returned. A non-zero value indicates an error condition.

Error Conditions

HPSS_EUSEDEFAULT	Tells the caller to use the default built-in behavior for this routine.

Other suggested error codes to use are:

HPSS_EBUSY	The server is busy. The request should be retried later.
HPSS_EPERM	The account specified may not be used by this user.
HPSS_ESYSTEM	This routine received an internal error. Usually a site customization problem.

Related Information

av_ValidateCreate

Clients

Client API via Gatekeeper

Notes

This routine can be customized by a site. It is the site's responsibility to make sure that consistent results are returned and performance is not adversely impacted.

The OkToCache argument should normally be TRUE. If a site customizes this routine it should only set this argument to FALSE if the returned information can change over time and should not be cached by the client.

Example Usage

This example is the default behavior of the av_ValidateCreate See the implementation of that routine, if available, for more details.

```
av_ValidateCreate() {
if site style accounting and account inheritance and
 parent account is supplied {
 return with parent account;
}
if user is "nobody", return with ACCT_REC_DEFAULT;
if site style accounting {
 if (user is superuser and Acct != ACCT_REC_DEFAULT)
     return ok with OutAcct = Acct;
 error = read user account from metadata;
 if not found {
     if Acct == ACCT_REC_DEFAULT, return with OutAcct = Acct;
     else return with HPSS_EPERM;
 }
} else if unix style accounting {
 *OutAcct = Uid;
}
log any error;
return error;
}
```

Chapter 8 Access Control List API Functions

8.1 API Interfaces

This section describes all the APIs that are provided in libhacl. In this discussion, the term "object" is used to refer to files and directories.

8.1.1 hacl_ConvertACLToHACL

Purpose

Convert an access control list to string format.

Synopsis

```
#include "hacl.h"
int
hacl_ConvertACLToHACL(
        ns_ACLConfArray_t  *ACL_n,        /* IN */
        hacl_acl_t         **ACL_h);      /* OUT */
```

Description

Convert an access control list from the format used by the client API routines into a format suitable for printing or use by other hacl routines.

Parameters

ACL_n	the ACL to be converted
ACL_h	the converted ACL

Return Values

If successful, returns zero. Otherwise, returns one of the values described below.

Error Conditions

EINVAL	ACL_n was not a valid nameserver ACL
(other)	an error returns by hpss_ConvertNamesToIds

See Also

Notes

8.1.2 hacl_ConvertHACLToACL

Purpose

Convert and access control list to HPSS format

Synopsis

```
#include "hacl.h"
int
hacl_ConvertHACLToACL(
        hacl_acl_t          *ACL_h,        /* IN */
        ns_ACLConfArray_t   **ACL_n)       /* OUT */
```

Description

Convert an access control list into a format suitable for use with the client API functions that manage access control lists.

Parameters

ACL_h	the ACL to be converted
ACL_n	the converted ACL

Return Values

If successful, returns zero. Otherwise, returns one of the values described below.

Error Conditions

EINVAL	ACL_h was not a valid access control list
(other)	error returned by hpss_ConvertNamesToIds

See Also

Notes

When ACL_n is no longer needed, the memory assigned to it must be returned by calling free().

8.1.3 hacl_ConvertHACLToString

Purpose

Convert an access control list to a form suitable for printing

Synopsis

```
#include "hacl.h"
int
hacl_ConvertHACLToString(
        hacl_acl_t          *ACL_h,        /* IN */
        int                 Flags,         /* IN */
        char                *EntrySeparator, /* IN */
        char                **ACL_s);      /* OUT */
```

Description

Convert an access control list into a form suitable for printing.

Parameters

ACL_h	the ACL to be converted
Flags	flags that define how output is to be formatted
EntrySeparator	string to be used to separate ACL entries
ACL_s	the converted ACL

Return Values

If successful, returns zero. Otherwise, returns one of the values described below.

Error Conditions

ENOMEM	not enough memory available to hold ACL_S

See Also

Notes

The Flags argument is the sum of values that determine how the ACL will be formatted. The following bits are defined:

HACL_M_USE_LOCAL_CELL_SPEC

HACL_M_USE_MASK_OBJ

If HACL_M_USE_LOCAL_CELL_SPEC is set, the id of the local cell will be displayed in all ACL entries, even those that refer to principals in the local cell. Otherwise the cell will only be displayed for entries that refer to foreign principals.

If HACL_M_USE_MASK_OBJ is set, the string will be adjusted to reflect the mask object. Otherwise the actual ACL will be shown.

When ACL_s is no longer needed, the memory assigned to it must be returned by calling free().

8.1.4 hacl_ConvertHACLPermsToPerms

Purpose

Convert permission string to HPSS format

Synopsis

```
#include "hacl.h"
int
hacl_ConvertHACLPermsToPerms(
        char            *HPerms,        /* IN */
        unsigned char   *Perms);        /* OUT */
```

Description

Convert a permission string in the format "rwxcid" into a form suitable for use with the client API ACL management functions. Both uppercase and lower case letters may be used, A hyphen may

be used to indicate missing permission bits.

Parameters

HPerms	the permissions to convert
Perms	the converted permissions

Return Values

If successful, returns zero. Otherwise, returns one of the values described below.

Error Conditions

EINVAL	Hperms contained an unrecognized character

See Also

Notes

8.1.5 hacl_ConvertHACLTypeToType

Purpose

Convert access control list entry type to HPSS format

Synopsis

```
#include "hacl.h"
int
hacl_hacl_ConvertHACLTypeToType(
        char            HType,          /* IN */
        unsigned char   *Type);         /* OUT */
```

Description

Convert an ACL entry type into a form suitable for use with client API functions. Valid type names include any of the standard DCE ACL entry types. These include user_obj, group_obj, other_obj, user, group, foreign_user, foreign_group, foreign_other, any_other, mask_obj, user_obj_delegate, group_obj_delegate, other_obj_delegate, foreign_other_delegate, and any_other_delegate. Strings can be entered in upper or lower case letters. The extended type is not allowed. The unauthorized type is allowed; but most client API routines will treat this ACL entry type as invalid. The output type will be a valid nameserver ACL type such as ACL_OBJ_MASK, etc.

Parameters

HType	the type to convert
Type	the converted type

Return Values

If successful, returns zero. Otherwise, returns one of the values described below.

Error Conditions

EINVAL	HType is not a valid ACL entry type

See Also

Notes

8.1.6 hacl_ConvertPermsToHACLPerms

Purpose

Convert HPSS permission mask to string

Synopsis

```
#include "hacl.h"
int
hacl_hacl_ConvertPermsToHACLPerms(
        unsigned char      Perms,        /* IN */
        char               *HPerms);     /* OUT */
```

Description

Convert the permission mask returned by hpss_GetACL into a form suitable for printing and/or input into other libhacl routines. The output takes the form of a string (e.g., "rwxcid") with missing permissions replaced by a hyphen (e.g., "rwx---").

Parameters

Perms	the permisions to convert
HPerms	the converted permissions

Return Values

If successful, returns zero. Otherwise, returns one of the values described below.

Error Conditions

EINVAL	Perms is not a valid permission mask

See Also

Notes

8.1.7 hacl_ConvertStringsToHACL

Purpose

Convert access control list strings to HACL format

Synopsis

```
#include "hacl.h"
int
hacl_ConvertStringsToHACL(
        int                NumEntries,       /* IN */
        char               *Entries[],       /* IN */
        char               *DefaultCell,     /* IN */
        char               *WhiteSpaceChars, /* IN */
        int                Flags,            /* IN */
        hacl_acl_t         **ACL_h);         /* OUT */
```

Description

Convert one or more string representations of access control list entries into a structure for input to other libhacl functions.

Parameters

NumEntries	Number of elements in Entries
Entries	Array of ACL strings
DefaultCell	Default Cell
WhiteSpaceChars	Entry separator(s)
Flags	Flags
ACL_h	Resulting ACL

Return Values

If successful, returns zero. Otherwise, returns one of the values described below.

Error Conditions

ENOMEM	cannot allocate memory for ACL
EINVAL	one or more ACL entries is in error

See Also

Notes

The Entries argument is an array of strings, each of which represents one or more ACL entries. If there are several ACLs in any one string, they are separated by the character(s) listed in WhiteSpace. The Flags argument defines the format of the incoming string. The following bits are defined:

HACL_M_REQUIRE_PERMS

HACL_M_USE_LOCAL_CELL_SPEC

If HACL_M_REQUIRE_PERMS is set, then the permission string must be supplied as part of the ACL entry. Otherwise, the permission string is optional. The first case would be used when an ACL is being created, while the second case would be used when an ACL entry is being deleted.

If HACL_M_USE_LOCAL_CELL_SPEC is set, the id of the local cell can optionally be provided in all ACL entries, even those that refer to principals in the local cell. Otherwise the cell should be specified only for entries that refer to foreign principals.

When ACL_h is no longer needed, the memory assigned to it must be returned using free().

8.1.8 hacl_ConvertTypeToHACLType

Purpose

Convert an access control list entry type to string format

Synopsis

```
#include "hacl.h"
int
hacl_ConvertTypeToHACLType(
```

```
        char            Type,           /* IN */
        char            *HType);        /* OUT */
```

Description

Convert an ACL entry type into a form suitable for printing or use with other functions in libhacl. The input type should be a valid nameserver ACL type such as ACL_OBJ_MASK, etc. The output is the corresponding string (eg, "mask_obj").

Parameters

Type	the entry type to convert
HType	the converted type

Return Values

If successful, returns zero. Otherwise, returns one of the values described below.

Error Conditions

EINVAL	Type was not a valid ACL entry type

See Also

Notes

8.1.9 hacl_DeleteHACL

Purpose

Delete selected entries from an object's access control list

Synopsis

```
#include "hacl.h"
int
hacl_DeleteHACL(
        char            *Path,          /* IN */
        unsigned32      Options,        /* IN */
        hacl_acl_t      *ACL_h);        /* IN */
```

Description

Delete the access control list entries specified in ACL_h from the file named by Path. The Options argument is used to select whether an initial container or initial object ACLs are to be processed.

Parameters

Path	path to the object
Options	processing options
ACL_h	ACL entries to be deleted

Return Values

If successful, returns zero. Otherwise, returns one of the values described below.

Error Conditions

EACCES	search permission is denied on a component in Path

EFAULT	the Path or ACL parameter is a NULL pointer
EINVAL	ACL_h contains invalid data
ENAMETOOLONG	the length of Path exceeds a system-imposed limit
ENOENT	the named object does not exist, or Path is an empty string
ENOTDIR	a component of Path is not a directory
EPERM	no privilege for attempted operation
ESRCH	a specified ACL entry could not be found in the object's existing ACL

See Also

Notes

The Options flag determines processing options. Values include:

HPSS_ACL_OBJECT_ACL - regular ACL

HPSS_ACL_INITIAL_CONTAINER_ACL - initial container ACL

HPSS_ACL_INITIAL_OBJECT_ACL - initial object ACL

8.1.10 hacl_GetHACL

Purpose

Get an object's access control list

Synopsis

```
#include "hacl.h"
int
hacl_GetHACL(
        char            *Path,          /* IN */
        unsigned32      Options,        /* IN */
        hacl_acl_t      **ACL_h);       /* OUT */
```

Description

Get the access control list of a object. The Options argument is used to select whether an initial container or initial object ACL is to be obtained.

Parameters

Path	path to the object
Options	processing options
ACL_h	the object's access control list

Return Values

If successful, returns zero. Otherwise, returns one of the values described below.

Error Conditions

EACCES	search permission is denied on a component in Path
EFAULT	the Path or ACL parameter is a NULL pointer
ENAMETOOLONG	the length of Path exceeds a system-imposed limit
ENOENT	the named object does not exist, or Path is an empty string
ENOTDIR	a component of Path is not a directory
EPERM	no privilege for attempted operation

See Also

Notes

The Options flag determines processing options. Values include:

HPSS_ACL_OBJECT_ACL - regular ACL

HPSS_ACL_INITIAL_CONTAINER_ACL - initial container ACL

HPSS_ACL_INITIAL_OBJECT_ACL - initial object ACL

When ACL_h is no longer needed, the memory assigned to it must be returned by calling free().

8.1.11 hacl_SetHACL

Purpose

Replace an object's access control list with a new one

Synopsis

```
#include "hacl.h"
int
hacl_SetHACL(
        char            *Path,          /* IN */
        unsigned32      Options,        /* IN */
        hacl_acl_t      *ACL_h);        /* IN */
```

Description

Replace the access control list of the object named by Path with a new one given by ACL_h. The Options argument is used to select whether an initial container or initial object ACL are to be processed.

Parameters

Path	path to the object
Options	processing options
ACL_h	the new ACL

Return Values

If successful, returns zero. Otherwise, returns one of the values described below.

Error Conditions

EACCES	search permission is denied on a component in Path
EFAULT	the Path or ACL parameter is a NULL pointer
EINVAL	ACL_h contains invalid data
ENAMETOOLONG	the length of Path exceeds a system-imposed limit
ENOENT	the named object does not exist, or Path is an empty string
ENOTDIR	a component of Path is not a directory
EPERM	no privilege for attempted operation

See Also

Notes

The Options flag determines processing options. Values include:

HPSS_ACL_OBJECT_ACL - regular ACL

HPSS_ACL_INITIAL_CONTAINER_ACL - initial container ACL

HPSS_ACL_INITIAL_OBJECT_ACL - initial object ACL

8.1.12 hacl_SortHACL

Purpose

Sort an access control list into canonical order

Synopsis

```
#include "hacl.h"
void
hacl_SortHACL(
        hacl_acl_t          *ACL_h);        /* IN/OUT */
```

Description

Sort the entries of an access control list into a standard order.

Parameters

ACL_h	the ACL to be sorted

Return Values

None.

Error Conditions

None. The function always succeeds.

See Also

Notes

The "standard" order arranges ACL entries in the order that they will be considered to decide whether a principal has access to an object. The order is the same order that is used by the dcecp and acl_edit utilities for listing ACL entries. For example, the mask_obj will appear first, followed by the user_obj, user, foreign_user, etc. entries. If there are two entries of the same type, they will be sorted based on the EntryName and/or CellName fields.

This routine should be called before displaying an ACL using hacl_ConvertHACLToString.

8.1.13 hacl_UpdateHACL

Purpose

Change selected entries in an object's access control list

Synopsis

```
#include "hacl.h"
int
hacl_UpdateHACL(
        char            *Path,          /* IN */
        unsigned32      Options,        /* IN */
        hacl_acl_t      *ACL_h);        /* IN */
```

Description

Update selected entries in an access control list of an object. The Options argument is used to select whether an initial container or initial object ACL is to be processed.

Parameters

Path	path to the object
Options	processing options
ACL_h	ACL to be updated

Return Values

If successful, returns zero. Otherwise, returns one of the values described below.

Error Conditions

EACCES	search permission is denied on a component in Path
EFAULT	the Path or ACL parameter is a NULL pointer
EINVAL	ACL_h contains invalid data
ENAMETOOLONG	the length of Path exceeds a system-imposed limit
ENOENT	the named object does not exist, or Path is an empty string
ENOTDIR	a component of Path is not a directory
EPERM	no privilege for attempted operation

See Also

Notes

The Options flag determines processing options. Values include:

HPSS_ACL_OBJECT_ACL - regular ACL

HPSS_ACL_INITIAL_CONTAINER_ACL - initial container ACL

HPSS_ACL_INITIAL_OBJECT_ACL - initial object ACL

HPSS_ACL_DONT_CALC_MASK - don't calculate mask

HPSS_ACL_CALC_MASK_IGNORE_ERRORS - calculate mask unconditionally

8.2 Data Definitions

This section describes the data definitions used by the routines in libhacl.

8.2.1 HACL-style Access Control List - hacl_acl_t

Description

The Access Control List structure defines an array of ACL entries.

Format

```
typedef struct hacl_acl {
int                 Length;
char                DefaultCell[HPSS_MAX_DCE_NAME];
hacl_acl_entry_t    ACLEntry[1];
} hacl_acl_t;
```

Length

The number of ACL entries in array ACLEntry.

DefaultCell

The cell used to set a context for interpreting the ACL.

ACLEntry

An array of ACL entries.

8.2.2 HACL-style Access Control List Entry - hacl_acl_entry_t

Description

The Access Control List Entry structure defines one entry in an ACL.

Format

```
typedef struct hacl_acl_entry {
char                EntryType[HACL_MAX_TYPE];
char                Perms[HACL_MAX_PERMS];
char                EntryName[HPSS_MAX_PRINCIPAL_NAME];
char                CellName[HPSS_MAX_DCE_NAME];
} hacl_acl_entry_t;
```

EntryType

The type of ACL entry; e.g., "user_obj", etc.

Perms

The permissions, composed of characters in "rwxcid-".

EntryName

Depending on the EntryType, the User name, group name, or cell name.

CellName

The cell name. Qualifies user and group names to make them unique.

Appendix A Programming Examples

Coding examples for opening, reading, writing, and closing HPSS files are provided below. The majority of the HPSS Client API functions mimic the standard POSIX I/O and file functions. As a result, UNIX programmers should have little difficulty in using the APIs, with the exception of enhanced function calls such as the **hpss_ReadList** and **hpss_WriteList** APIs. Examples of these extended functions are provided in Examples 3 and 4.

Sample Makefile for Example Code

The following Makefile may be used to compile the example code in the sections below: The Makefile is for an AIX system.

```
CC          = xlc_r4
CFLAGS      = -I/opt/hpss/include -I/usr/lpp/encina/include


LIB_DIRS    = -L/opt/hpss/lib \
              -L/usr/lpp/encina/lib

LIBS        = $(LIB_DIRS) \
              -ldce \
              -lEncSfs \
              -lEncina \
              -lhpss \
              /opt/hpss/tools/lib/tools.o

.c.o:;@echo $(INDENT) Compiling $< ...
   @$(CC) $(CFLAGS) -c $<

all: api_read api_write api_readlist api_writelist diropts fileattrs \
   links statfs api_config
```

```
api_read: read.o
   #$(CC) $(LIBS) -o $@ read.o

api_write: write.o
   #$(CC) $(LIBS) -o $@ read.o

api_readlist: readlist.o
   #$(CC) $(LIBS) -o $@ readlist.o

api_writelist: writelist.o
   #$(CC) $(LIBS) -o $@ writelist.o

diropts: diropts.o
   #$(CC) $(LIBS) -o $@ diropts.o

fileattrs: fileattrs.o
   #$(CC) $(LIBS) -o $@ fileattrs.o

links: links.o
   #$(CC) $(LIBS) -o $@ links.o

statfs: statfs.o
   #$(CC) $(LIBS) -o $@ statfs.o

api_config: api_config.o
   #$(CC) $(LIBS) -o $@ api_config.o
```

Example 1: Simple write - hpss_Write

This example demonstrates a simple application which opens an HPSS file, initializes an output buffer with each 8 bytes numbered, writes multiple records, and then closes the file. This program is passed the following arguments:

argv[1]	HPSS file name
argv[2]	number of bytes to write
argv[3]	Class of Service ID
argv[4]	record size

```
#include <errno.h>
```

```
#include <fcntl.h>
#include <malloc.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

#include "hpss_api.h"
#include "hpss_types.h"
#include "u_signed64.h"

#define MAX_BUF_SIZE 16777216

/*
 *  MAIN
 *
 */

main( argc, argv )
int  argc;
char *argv[];
{
    char                    *buf;
    char                    *buf_ptr;
    register                cnt;
    int                         eflags, eperms, erecsize;
    int                         fd, last_blk_cnt, rc, rc2;
    hpss_cos_hints_t            hints_in;   /* hints input structure */
    hpss_cos_hints_t            hints_out;  /* hints output structure */
    hpss_cos_priorities_t       hints_pri;  /* hints priority structure */
    u_signed64                  erecsize64, rec_cnt, i, count, local_count;
    u_signed64                  word64_num;

    /* Initialize local variables */

    rc, rc2 = 0;
    word64_num = cast64m( 1 );
    decchar_to_u64( argv[2], &count, 20 );
    eflags = O_WRONLY | O_CREAT;
    eperms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH;
    memset( &hints_in, 0, sizeof(hpss_cos_hints_t) );
    memset( &hints_pri, 0, sizeof(hpss_cos_priorities_t) );
```

```
/*
 *  if hints are specified (argv[3])
 *      Call atoi() to convert the COSId to an integer
 *  else
 *      Set the COSId to 1
 *  Set the COS ID priority in COS priority structure
*/

if (argc >= 4)
    hints_in.COSId =  atoi( argv[3] );
else
    hints_in.COSId = 1;

hints_pri.COSIdPriority = REQUIRED_PRIORITY;

/*  if record size is specified (argv[4])
 *      Call atoi() to convert record size to an integer
 *  else
 *      Set record size to the default (1MB)
 */

if (argc >= 5)
    erecsize =  atoi( argv[4] );
else
    erecsize = 1048576;

/*
 *  Call hpss_Open() to open the HPSS file
 *  if return code < 0
 *      Output an error message and return
 */

if ((fd = hpss_Open( argv[1], eflags, eperms, &hints_in, &hints_pri,
    &hints_out )) < 0) {
    printf( "HPSS file open failed, rc = %d\n", fd );
    exit( -1 );
}

/*
 *  Call malloc() to allocate an output buffer
 *  if return code = NULL
 *      Output an error message and return
 */
```

```
if ((buf_ptr = (char *) malloc( erecsize )) == NULL) {
    perror( "Malloc failed" );
    return( -1 );
}

/*
 *  loop until requested number of bytes has been written to the HPSS file
 *      Number each 8 bytes of the output buffer
 *      Call hpss_Write() to write the HPSS file
 *      if return code < 0
 *          Output an error message and return
 */

erecsize64 = cast64m( erecsize );
local_count = add64m( count, erecsize64 );
local_count = sub64m( local_count, cast64m( 1 ) );
rec_cnt = div64m( local_count, erecsize );
last_blk_cnt = low32m( mod64m( count, erecsize ) );
if (last_blk_cnt == 0) last_blk_cnt = erecsize;
i = cast64m( 1 );

for (; ge64m(rec_cnt, i); ) {
    if (eq64m( rec_cnt, i)) erecsize = last_blk_cnt;
    buf = buf_ptr;

    for (cnt=0; cnt<(erecsize+7)/8; cnt++) {
        *((u_signed64 *) buf) = word64_num;
        word64_num = add64m( word64_num, cast64m( 1 ) );
        buf += 8;
    }

    if ((rc = hpss_Write( fd, buf_ptr, erecsize )) < 0) {
        printf( "HPSS write failed, rc = %d\n", rc );
        rc2 = -1;
        break;
    }
    i = add64m( i, cast64m( 1 ));
}

/*
 *  Call hpss_Close() to close the HPSS file
 *  if return code < 0
```

```
     *      Output an error message and return
     */

    if ((rc = hpss_Close( fd )) < 0) {
        printf( "HPSS file close failed, rc = %d\n", rc );
    }

    if (rc2 == 0) printf( "file %s created\n", argv[1] );
    exit( rc2 );
}
```

Example 2: Simple read - hpss_Read

This example demonstrates a simple application which opens an HPSS file, reads and verifies the file contents, and then closes the file. This program is passed the following arguments:

argv[1]	HPSS file name

```
#include <fcntl.h>
#include <malloc.h>
#include <string.h>
#include <unistd.h>

#include "hpss_api.h"
#include "u_signed64.h"

#define MAX_BUF_SIZE 16777216

main( int argc, char *argv[] )
{
    char                        *buf;
    register                    cnt;
    int                         flags, perms, recsize;
    char                        *buf_ptr;
    int                         fd, i0, i1, last_blk_cnt, rc, rc2;
    hpss_cos_hints_t            hints_in;
    hpss_cos_hints_t            hints_out;
    hpss_cos_priorities_t       hints_pri;
    u_signed64                  recsize64, rec_cnt, i;
    u_signed64                  word64_num;

    /* Initialize local variables */
```

```
rc, rc2 = 0;
word64_num = cast64m( 1 );
flags = O_RDONLY;
recsize = MAX_BUF_SIZE;
memset( &hints_in, 0, sizeof(hpss_cos_hints_t) );
memset( &hints_pri, 0, sizeof(hpss_cos_priorities_t) );

/*
 *  Call hpss_Open() to open the HPSS file
 *  if return code < 0
 *      Output an error message and return
 */

if ((fd = hpss_Open( argv[1], flags, perms, &hints_in, &hints_pri,
    hints_out )) < 0) {
    printf( "HPSS open failed, rc = %d\n", fd );
    return( -1 );
}

/*
 *  Call malloc() to allocate an output buffer
 *  if return code = NULL
 *      Output an error message and return
 */

if ((buf_ptr = (char *) malloc( MAX_BUF_SIZE )) == NULL) {
    perror( "Malloc failed" );
    return( -1 );
}

/*
 *  loop until end of file
 *      Call hpss_Read() to read the HPSS file
 *      if return code < 0
 *          Output an error message and return
 *      Verify each 8 bytes of the input buffer is correctly numbered
 */

recsize64 = cast64m( recsize );

for (;;) {
```

```
        if ((rc = hpss_Read( fd, buf_ptr, recsize )) < 0) {
            printf( "HPSS read failed, rc = %d\n", rc );
            rc2 = -1;
            break;
        }

        if (rc == 0) break;
        buf = buf_ptr;

        for (cnt=0; cnt<(rc+7)/8; cnt++) {

            if (memcmp( buf, &word64_num, 8 )) {
                i0 = *buf;
                i1 = *(buf + 4);
                printf( "Error in file contents: word64=%d %d, value=%d %d\n",
                        word64_num.high, word64_num.low, i0, i1 );
            }

            word64_num = add64m( word64_num, cast64m( 1 ) );
            buf += 8;
        }

    }

    /*
     *  Call hpss_Close() to close the HPSS file
     *  if return code < 0
     *      Output an error message and return
     */

    if ((rc = hpss_Close( fd )) < 0) {
        printf( "HPSS close failed, rc = %d\n", rc );
    }

    exit( rc2 );
}
```

Example 3: Write List - Mover to Mover Protocol

This example is provided to illustrate use of the hpss_WriteList() function.

```
/
*=====================================================================
======
```

```
 *

 * Name:

 *      writelist.c - Write an HPSS file in parallel using
hpss_Writelist,

 *                    multiple data-receiving threads, and the mover
protocol

 *                    routines, supporting TCP and shared-memory data

 *                    transfers

 *

 * Disclaimer:

 *      This software is provided "as is" and may be freely copied and

 *      modified as desired.

 *

 * Usage:

 *      writelist [-vcx] [-n maxConnections] [-s bufferSize]

 *               [-C cos] [-S filesize] [-h hostname]*

 *               [-p tcp|shm]* [-x sbmax] <path>

 *

 *      -v             Prints verbose output

 *      -c             Prints low-level control information

 *      -t             Print transfer rate in whole bytes per second

 *

 *      -n maxConnections

 *                    Maximum number of open Mover transfer connections

 *                     (default is DEFAULT_MAX_CONNECTIONS).

 *

 *      -s bufferSize  Buffer size used to send data within each
transfer
```

```
 *                      thread (default is defined by
DEFAULT_BUFFER_SIZE).

 *

 *      -C cos         Class of Service to use for creating file.
Default

 *                      is to select one based on file size.

 *

 *      -S filesize    Total number of bytes to write.

 *                      (default is defined by DEFAULT_FILE_SIZE).

 *

 *      -h hostname

 *                      Specifies one or more hostnames associated with
the

 *                      desired network interface(s) for TCP-based
transfers

 *                      (default is network associated with default
hostname).

 *                      If multiple hostnames are specified, each
transfer

 *                      threads will be assigned one of the network
interfaces

 *                      in a round-robin fashion.

 *

 *      -p tcp|shm

 *                    By default, TCP and shared memory transfers are *

 *                      enabled. The -p option can be repeatedly used to

 *                    specify which transfer options to make available.
For

 *                      example, to restrict transfers to TCP only, use

 *                      "-p tcp".

 *
```

```
 *      -x sbmax       Sets the initial upper limit on the TCP socket
 *                     send/receive buffer sizes
 *
 *      <path>         The HPSS file to write.  Relative pathnames are
 *                     resolved from the perspective of the user's home
 *                     directory within HPSS.
 *
 * Description:
 *
 *      This program writes all data stored in an HPSS file named <path>
using
 *      parallel I/O via the hpss_WriteList API and HPSS Mover protocol
 *      functions.  The program negotiates with the corresponding HPSS
Movers
 *      to determine which transfer protocol to use.  The application is
 *      coded to handle TCP and/or shared memory transfers.
 *
 *      The -v option enables the user to see each transfer of data from
a
 *      Mover, the order the data is sent, and what protocol is used.
The
 *      -c option shows control debug information.
 *
 *      The -t argument can be used to output a throughput number that
can be
 *      more easily used by other programs, spreadsheets, etc.
 *
 *      The -s argument defines the size of each memory buffer used to
send
 *      data within each transfer thread.  If not specified, a default
value
```

```
 *      is used.

 *

 *      The -n argument defines the maximum number of simultaneous
connections

 *      to HPSS Movers, which also corresponds to the number of memory
buffers

 *      used in receiving data in parallel.  If not specified, a default
value

 *      is used.  This program will create a contiguous shared-memory
segment

 *      to send data, the size of which is <bufferSize> times
<maxConnections>

 *      (regardless of which transfer protocol is selected).

 *

 *      Since segments of an HPSS bitfile may be striped across multiple

 *      devices (and Movers) and/or may reside at different levels in a

 *      hierarchy, multiple buffers/threads can be used to send data in

 *      parallel to different Movers who are trying to send data

 *      independently.

 *

 *      The -h option is used to specify an alternate hostname
interface(s) to

 *      use for TCP-based data transfers.

 *

 *

 *      For best performance, the buffer size should match either the VV
block

 *      size or the Mover buffer size, whichever is less, and the
maximum

 *      number of connections should be equal to the total number of
devices

 *      the file is spread across.
```

```
 *

 *      This program requires that the user already have DCE credentials
prior

 *      to invocation.

 *


*=====================================================================
====*/


#include <sys/types.h>

#include <sys/socket.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <unistd.h>

#include <netinet/in.h>

#include <netinet/tcp.h>

#include <signal.h>


#include "hpss_api.h"

#include "u_signed64.h"

#include "mvr_protocol.h"


#include "support.h"



/* Define program default values */


#define DEFAULT_BUFFER_SIZE        (4*1048576)

#define DEFAULT_MAX_CONNECTIONS    32
```

```
#define DEFAULT_SOCKET_SBMAX       (8*1048576)
#define DEFAULT_FILE_SIZE          (100*1048576)

/* Define maximum values for sanity checks */

#define MAX_BUFFER_SIZE            (32*1048576)
#define MAX_MAX_CONNECTIONS        32

/* Define local function prototypes */

void manage_mover_connections ();
void transfer_routine  (int socketDes);
void socket_setoptions (int socketFd);
void signal_thread ();
void handle_signals ();
void init_buf(char *Buf,int Size);

/* Define a structure of information to track for each Mover transfer
 * connection/thread
 */
typedef struct {
  int       active;           /* Whether thread/connection is active */
  pthread_t threadId;         /* Id of the transfer thread */
  int       controlSocketFd;  /* Control socket descriptor */
  int       shmId;            /* Shared memory segment id */
} connection_info_t;
```

```
/* Define global variables (globals start with capital letter)
 */
int         RequestId;        /* HPSS request id */
int         TransferStatus;   /* Overall status of the data transfer
                               * (HPSS_E_NOERROR if ok) */
int         FileDes;          /* HPSS file descriptor */
int         ControlSocket = 0;/* Central Mover connection socket */
sigset_t    SigMask;          /* Signal mask */

/* Define global variables associated with command-line options
 */
typedef struct {
  struct hostent *hostEntry;
  char            hostname[128];
  unsigned long   ipAddr;
} tcphost_t;

unsigned32  NumHosts;             /* -h argument counter */
tcphost_t  *HostList;             /* -h argument (length is NumHosts) */
int         VerboseOutput = 0;    /* -v argument (0=off, 1=on) */
int         ControlOutput = 0;    /* -c argument (0=off, 1=on) */
int         WholeBytesOutput = 0; /* -x argument (0=off, 1=on) */
unsigned32  MaxConnections;       /* -n argument */
unsigned32  BufferSize;           /* -s argument */
unsigned32  SbMax = DEFAULT_SOCKET_SBMAX; /* -x argument */
u_signed64  FileSize;             /* Size of the HPSS file */
```

```
/* The following global variables are protected by the mutex
GlobalVarMutex

 */

pthread_mutex_t GlobalVarMutex;



connection_info_t  *Connections;/* Array of connection thread info */



int         CurrentHost = 0;    /* Index into HostList for next TCP
address */

u_signed64  TotalBytesWritten;  /* Actual bytes sent to Movers */



/
*=====================================================================
======

 * Function:

 *    hpss_error - Print out info on an HPSS error condition

 *

 * Arguments:

 *    function     Name of the HPSS function that returned "status"

 *    status       HPSS error code

 *

 * Return Values:

 *    <none>


*=====================================================================
====*/



void hpss_error (char *function, signed32 status)

{

  fprintf(stderr, "%s (%ld): %s\n", function, status,
status_string(status));
```

```
}


/
*=========================================================================
======

 * Function:

 *    terminate - Gracefully terminate the process, closing resources

 *                appropriately

 *

 * Arguments:

 *    exitStatus  Value used as the exit status

 *

 * Return Values:

 *    This function terminates the process and therefore does not

 *    return.

*=========================================================================
====*/


void terminate (int exitStatus)

{

  int index;


  /* Close down the HPSS file if it is open

   */

  if (FileDes > 0) {

    if (ControlOutput) printf("Closing HPSS file descriptor %d\n",
FileDes);

    (void)hpss_Close (FileDes);

  }
```

```
  /* Step through the connections and for any that are active, delete
the

   * shared memory segment if it exists

   */

  for (index=0; index < MaxConnections; index++) {



    if (Connections[index].active && Connections[index].shmId != -1) {

      if (ControlOutput) printf("Deleting shared memory for thread
%d\n",

                                index+1);

      shmctl (Connections[index].shmId, IPC_RMID, (struct shmid_ds
*)NULL);

    }

  }



  exit (exitStatus);

}



/
*=====================================================================
======

 * main


*=====================================================================
====*/



main (int argc, char *argv[])

{

  int                   i, badUsage; /* Counters and flags */

  size_t                tmp;        /* Temporary variables */
```

```
  int                    cos=0;      /* Specified COS (if any) */

  char                   *programName, *s;

  IOD_t                  iod;  /* IOD passed to hpss_WriteList */

  IOR_t                  ior;  /* IOR returned from hpss_WriteList */

  srcsinkdesc_t          src, sink; /* IOD source/sink descriptors */

  hpss_cos_hints_t       hints;

  hpss_cos_priorities_t pris;

  hpss_cos_md_t          cosinfo;

  struct sockaddr_in     controlSocketAddr; /* control socket addresses
*/

  int                    writeListFlags = 0; /* Flags on hpss_WriteList
call */

  pthread_t              manageConnectionsThread; /* Spawned thread id */

  pthread_addr_t         pthreadStatus;

  signed32               status;        /* HPSS return code/status */

  timestamp_t            startTime, endTime, totalTime; /*various
timestamps*/


  totalTime.tv_sec = totalTime.tv_usec = 0;

  RequestId = getpid();


  memset (&iod,  0, sizeof(iod));

  memset (&src,  0, sizeof(src));

  memset (&sink, 0, sizeof(sink));

  memset(&hints,0,sizeof(hints));

  memset(&pris,0,sizeof(pris));


  programName     = argv[0];

  badUsage        = 0;
```

```
MaxConnections  = DEFAULT_MAX_CONNECTIONS;

BufferSize      = DEFAULT_BUFFER_SIZE;

FileSize        = cast64m(DEFAULT_FILE_SIZE);

src.Flags      = 0;


/* Process the arguments

 */


while (--argc > 0 && (*++argv)[0] == '-') {


  for (s = argv[0]+1; *s != '\0'; s++) {


    switch (*s) {


    case 'v':                 /* for verbose output */

      VerboseOutput = 1;

      break;


    case 'c':                 /* for low-level control output */

      ControlOutput = 1;

      break;


    case 't':                 /* for printing whole byte throughput
rate */

      WholeBytesOutput = 1;

      break;
```

```
      case 'n':                 /* to specify max connections/no.
buffers */

        if (argc > 1 && (*(argv+1))[0] != '-') {

          MaxConnections = atoi((++argv)[0]);

          argc -= 1;

        }

        else

          badUsage = 1;

        break;


      case 's':                 /* to specify buffer size */

        if (argc > 1 && (*(argv+1))[0] != '-') {

          BufferSize = atobytes((++argv)[0]);

          argc -= 1;

        }

        else

          badUsage = 1;

        break;


      case 'x':                 /* to specify socket buffer sizes */

        if (argc > 1 && (*(argv+1))[0] != '-') {

          SbMax = atobytes((++argv)[0]);

          argc -= 1;

        }

        else

          badUsage = 1;

        break;
```

```
        case 'S':                 /* to specify file size */
          if (argc > 1 && (*(argv+1))[0] != '-') {
            FileSize =  atobytes64((++argv)[0]);
            argc -= 1;
          }
          else
            badUsage = 1;
          break;

        case 'C':                 /* to specify class of service */
          if (argc > 1 && (*(argv+1))[0] != '-') {
            cos = atoi((++argv)[0]);
            argc -= 1;
          }
          else
            badUsage = 1;
          break;

        case 'h':                 /* TCP hostname */
          if (argc > 1 && (*(argv+1))[0] != '-') {
            if (!HostList) {
              HostList = (tcphost_t *)malloc (sizeof(*HostList));
            }
            else {
             HostList = (tcphost_t *)realloc (HostList, sizeof(*HostList)
*
                                                (NumHosts + 1));
            }
```

```
            strncpy (HostList[NumHosts].hostname, (++argv)[0],
                     sizeof(HostList[NumHosts].hostname));

            /* Make sure it is a legitimate hostname */

            HostList[NumHosts].hostEntry =
              gethostbyname (HostList[NumHosts].hostname);

            if (!(HostList[NumHosts].hostEntry)) {
              fprintf (stderr, "Invalid hostname \"%s\"\n",
                       HostList[NumHosts].hostname);
              perror ("gethostbyname");
              badUsage = 1;
            }
            else {
              HostList[NumHosts].ipAddr =
                *((unsigned32*)(HostList[NumHosts].hostEntry-
>h_addr_list[0]));
              ++NumHosts;
            }

            argc -= 1;
          }
          else
            badUsage = 1;
          break;
```

```
        case 'p':                  /* transfer protocol */
          if (argc > 1) {
            ++argv;

            if (!strcmp(argv[0], "tcp")) {
              src.Flags |= XFEROPT_IP;
            }
            else if (!strcmp(argv[0], "shm")) {
              src.Flags |= XFEROPT_SHMEM;
            }
            else {
              printf ("Invalid transfer protocol - use tcp or shm\n");
              badUsage = 1;
            }
            argc -= 1;
          }
          else
            badUsage = 1;
          break;

        default:
          badUsage = 1;

        } /* end switch */
      } /* end for */
    } /* end while */
```

```
  if (argc != 1) badUsage = 1;


  if (badUsage) {
    printf("Usage:\n\n");
    printf("%s [-vct] [-n maxConnections] [-s bufferSize]\n",
           programName);
    printf("         [-C cos] [-S filesize] [-h hostname]*\n");
    printf("         [-p tcp|shm]* [-x sbmax] <path>\n\n");
    printf("-v\n");
    printf("\tPrints verbose output.\n\n");
    printf("-c\n");
    printf("\tPrints low-level control information.\n\n");
    printf("-t\n");
    printf("\tPrints throughput rate in whole bytes.\n\n");
    printf("-n maxConnections\n");
    printf("\tMaximum number of concurrent transfer threads that ");
    printf("are available for\n");
    printf("\tcommunicating simultaneously with HPSS Movers.  This ");
    printf("corresponds to the\n");
    printf("\tnumber of buffers allocated to send HPSS data.  ");
    printf("Default is %d.\n\n", DEFAULT_MAX_CONNECTIONS);
    printf("-s bufferSize\n");
   printf("\tBuffer size used to send data within each transfer thread.
");
    printf("Values\n");
    printf("\tsuch as \"2mb\" can be specified.  Default is ");
    print_bytes (DEFAULT_BUFFER_SIZE);
    printf(".\n\n");
```

```
    printf("-C cos\n");

    printf("\tClass of Service to use for creating file.\n");

    printf("\tDefault is to select one based on file size.\n\n");

    printf("-S filesize\n");

    printf("\tTotal number of files to write. Values such as \"2mb\" can
");

    printf("be specified.\n");

    printf("\tDefault is ");

    print_bytes (DEFAULT_FILE_SIZE);

    printf(".\n\n");

    printf("-h hostname\n");

    printf("\tSpecifies one or more hostnames associated with the
desired ");

    printf("network\n");

    printf("\tinterface(s) for TCP-based transfers (default is network
");

    printf("associated\n");

    printf("\twith default hostname).  If multiple hostnames are ");

    printf("specified, each\n");

    printf("\ttransfer thread will be assigned one of the network ");

    printf("interfaces in a\n");

    printf("\tround-robin fashion.\n\n");

    printf("-p tcp|shm\n");

    printf("\tBy default, TCP and shared memory transfers are enabled.

    printf("The -p option can be repeatedly used to specify which\n");

    printf("\ttransfer options to make available. \n\n");

    printf("-x sbmax\n");

    printf("\tSets the initial upper limit on the TCP socket send/
receive ");
```

```
    printf("buffer sizes\n");

    printf("\tDefault is ");

    print_bytes (SbMax);

    printf(".\n\n");

    printf("<path>\n");

    printf("\tThe HPSS file to write.  Relative pathnames are resolved
");

    printf("from the\n");

    printf("\tperspective of the user's home directory within HPSS.\n");


    terminate (1);

  }


  /* Perform some sanity checks on values

   */

  if (MaxConnections > MAX_MAX_CONNECTIONS) {

    printf("Maximum limit on number of buffers is %d\n",
MAX_MAX_CONNECTIONS);

    terminate (1);

  }


  if (BufferSize > MAX_BUFFER_SIZE) {

    printf ("Maximum limit on buffer size is ");

    print_bytes (MAX_BUFFER_SIZE);

    printf ("\n");

    terminate (1);

  }
```

```
  /* If no transfer protocol(s) were specified, use all of them
   */
  if (!src.Flags) {
    src.Flags  = XFEROPT_IP | XFEROPT_SHMEM;
  }
  src.Flags |= /* HOLD_RESOURCES | */ CONTROL_ADDR;


  /* If no hostname was specified, use the local default hostname for
TCP
   * transfers
   */
  if (!HostList) {
    HostList = (tcphost_t *) malloc (sizeof(*HostList));


    if (gethostname (HostList[0].hostname,
                     sizeof(HostList[0].hostname)) < 0) {
      perror ("gethostname");
      terminate (1);
    }


    HostList[0].hostEntry = gethostbyname (HostList[0].hostname);


    if (!(HostList[NumHosts].hostEntry)) {
      perror ("gethostbyname");
      terminate (1);
    }
    HostList[0].ipAddr =
      *((unsigned32*)(HostList[0].hostEntry->h_addr_list[0]));
```

```
  ++NumHosts;
}

/* Set up signal handling
 */
handle_signals();

/* Allocate the array of transfer/connection information
 */
Connections = (connection_info_t *) malloc (sizeof(Connections[0]) *
                                            MaxConnections);
memset (Connections, 0, sizeof(Connections[0]) * MaxConnections);

/* Pass the COS as a hint, if it was specified.  If not specified, a
 * COS will be selected automatically based on file size.  Even if
 * was give a COS as a hint, still provide file size info so a
 * proper segment size will be selected.
 */
if (cos) {
  hints.COSId = cos;
  pris.COSIdPriority = REQUIRED_PRIORITY;
}
hints.MinFileSize = FileSize;
hints.MaxFileSize = FileSize;
pris.MinFileSizePriority = REQUIRED_PRIORITY;
pris.MaxFileSizePriority = REQUIRED_PRIORITY;
```

```
pthread_mutex_init (&GlobalVarMutex, pthread_mutexattr_default);


/* Open the HPSS file (argv[0] points to the HPSS pathname)
 */
FileDes=hpss_Open(argv[0], O_WRONLY | O_TRUNC | O_CREAT, 0666,
                  &hints, &pris, NULL);


if (FileDes < 0) {
  hpss_error ("hpss_Open", FileDes);
  terminate (FileDes);
}


if (VerboseOutput) {
  printf("File size is ");
  print_bytes64(FileSize);
  putchar('\n');
}



/* Create the local control socket which all Movers will
 * initially connect to
 */
ControlSocket = socket (AF_INET, SOCK_STREAM, 0);


if (ControlSocket == -1) {
  perror ("socket");
  terminate (1);
```

```
}

if (ControlOutput) {
  printf ("Control socket is socket %d\n", ControlSocket);
}

(void)memset (&controlSocketAddr, 0, sizeof(controlSocketAddr));
controlSocketAddr.sin_family      = AF_INET;
controlSocketAddr.sin_addr.s_addr = INADDR_ANY;
controlSocketAddr.sin_port        = 0;

if (bind (ControlSocket, (const struct sockaddr*)&controlSocketAddr,
          sizeof(controlSocketAddr)) == -1) {
  perror ("bind");
  terminate (1);
}

tmp = sizeof (controlSocketAddr);

if (getsockname (ControlSocket,
                 (struct sockaddr *)&controlSocketAddr,
                 (size_t *)&tmp) == -1) {
  perror ("getsockname");
  terminate (1);
}

if (listen (ControlSocket, SOMAXCONN) == -1) {
```

```
    perror ("listen");

    terminate (1);

  }


  /* Start the thread to receive control connections from individual
Movers

   */

  pthread_create (&manageConnectionsThread, pthread_attr_default,

                  (pthread_startroutine_t) manage_mover_connections,

                  (pthread_addr_t)NULL);

  pthread_yield();


  /* Set the source/sink length to the number of bytes we want

   */

  sink.Offset = src.Offset = cast64m(0);

  sink.Length = src.Length = FileSize;


  /* Define source and sink descriptors and the IOD

   */

  sink.SrcSinkAddr.Type = CLIENTFILE_ADDRESS;

  sink.SrcSinkAddr.Addr_u.ClientFileAddr.FileDes = FileDes;

  sink.SrcSinkAddr.Addr_u.ClientFileAddr.FileOffset = cast64m(0);


  src.SrcSinkAddr.Type = NET_ADDRESS;

  src.SrcSinkAddr.Addr_u.NetAddr.SockTransferID =cast64m (RequestId);

  src.SrcSinkAddr.Addr_u.NetAddr.SockAddr.addr  =HostList[0].ipAddr;

  src.SrcSinkAddr.Addr_u.NetAddr.SockAddr.port
=controlSocketAddr.sin_port;
```

```
src.SrcSinkAddr.Addr_u.NetAddr.SockAddr.family=controlSocketAddr.sin_fa
mily;

  src.SrcSinkAddr.Addr_u.NetAddr.SockOffset     =cast64m (0);


  iod.Function       = IOD_WRITE;

  iod.RequestID      = RequestId;

  iod.SrcDescLength  = 1;

  iod.SinkDescLength = 1;

  iod.SrcDescList    = &src;

  iod.SinkDescList   = &sink;




  if (ControlOutput) {

    printf("Client request id is %d\n", RequestId);

  }


  TotalBytesWritten = cast64m(0);


  startTime = get_current_timestamp();

  TransferStatus = HPSS_E_NOERROR;


  if (ControlOutput) {

      printf ("Issuing hpss_WriteList call for ");

      print_bytes64 (FileSize);

      printf ("\n");

  }
```

```
memset (&ior,  0, sizeof(ior));

status = hpss_WriteList (&iod, writeListFlags, &ior);

if (status) {
    hpss_error ("hpss_WriteList", status);

    if (ior.Status != HPSS_E_NOERROR) {
      hpss_error ("IOR status", ior.Status);
      printf ("Returned flags is 0x%x, bytes moved is ", ior.Flags);
      print_bytes64 (TotalBytesWritten);
      putchar ('\n');
      terminate (1);
    }

    if (TransferStatus == HPSS_E_NOERROR) TransferStatus = status;
  }


endTime = get_current_timestamp ();

totalTime = diff_timestamps (startTime, endTime);

/* Close the HPSS file
 */
status = hpss_Close (FileDes);
```

```
if (status < 0) {
  hpss_error ("hpss_Close", status);
}

/* Let's make sure that all data has actually been sent before we kill
 * any active transfer threads
 */
pthread_mutex_lock (&GlobalVarMutex);

if (neq64m (FileSize, TotalBytesWritten)) {

  struct timespec delay = { 1, 0 }; /* 1 second */

  printf("filesize is ");
  print_bytes64(FileSize);
  printf(", TotalBytesWritten is ");
  print_bytes64(TotalBytesWritten);
  printf("\n");

  pthread_mutex_unlock (&GlobalVarMutex);

  /* Wait for all transfer threads to complete before moving on
   */
  for (tmp=0, i=0; i < MaxConnections; i++) {

    pthread_mutex_lock (&GlobalVarMutex);
```

```
        while (Connections[i].active) {

          pthread_mutex_unlock (&GlobalVarMutex);

          if ((VerboseOutput || ControlOutput) && !tmp) {
            printf ("Waiting on thread %d to complete...\n", i+1);
            tmp = 1;                /* only show the message once */
          }

          (void) pthread_delay_np (&delay);

          pthread_mutex_lock (&GlobalVarMutex);
        }
        pthread_mutex_unlock (&GlobalVarMutex);
      }
    }

    /* Print stats
     */
    if (!eq64m (TotalBytesWritten, cast64m(0))) {
      u_signed64 usecs64;
      unsigned32 throughput;

      usecs64    = add64m (mul64m (cast64m (totalTime.tv_sec), 1000000),
                           cast64m (totalTime.tv_usec));

      throughput = cast32m (div2x64m (mul64m (TotalBytesWritten, 1000000),
```

```
                                        usecs64));


      if (WholeBytesOutput) {

        printf ("%ld\n", throughput);

      }

      else {

        print_bytes64 (TotalBytesWritten);

        printf(" successfully written in %d.%06d sec -> ",

             totalTime.tv_sec, totalTime.tv_usec);


        usecs64 = mul64m (cast64m (totalTime.tv_sec), 1000000);

        inc64m (usecs64, cast64m (totalTime.tv_usec));


        print_bytes_per_second (cast32m (div2x64m (mul64m
(TotalBytesWritten,

                                1000000), usecs64)));

        putchar('\n');

      }

      fflush (stdout);

    }

    pthread_mutex_unlock (&GlobalVarMutex);


    /* Now cancel the manage_mover_connections thread

     */

    (void)pthread_cancel (manageConnectionsThread);

    (void)pthread_join   (manageConnectionsThread, &pthreadStatus);

    (void)pthread_detach (&manageConnectionsThread);
```

```
  terminate (0);

}



/
*=====================================================================
======

 * Function:

 *    manage_mover_connections - Accept socket connections from HPSS
Movers &

 *                               spawn a new thread to handle each Mover

 *                               connection and data transfer

 * Return Values:

 *    <none>


*=====================================================================
====*/



void manage_mover_connections ()

{

  int moverSocketFd;                /* New Mover socket file descriptor
*/

  int index;                        /* Counters */

  int tmp;                         /* Temporary variable */



  struct sockaddr_in  socketAddr;




  /* Loop until this thread is cancelled

   */

  for (;;) {
```

```
    tmp = sizeof(socketAddr);

    while ((moverSocketFd =
            accept (ControlSocket, (struct sockaddr *)&socketAddr,
                    (size_t *)&tmp)) < 0) {

      if ((errno != EINTR) && (errno != EAGAIN)) {
        perror ("accept");
        TransferStatus = errno;
        break;
      }

    } /* end while */

    if (moverSocketFd < 0) break;

    if (ControlOutput) {
      printf ("Mover control connection accepted on control socket
%d\n",
              moverSocketFd);
    }

    /* Find a connection/transfer thread that is free to accept this
     * connection.  If one is not free, sleep for a bit
     * and try again.
     */
    do {
```

```
    pthread_mutex_lock (&GlobalVarMutex);

    for (index = 0; index < MaxConnections; index++) {

      if (!Connections[index].active) {
        Connections[index].active = 1;
        Connections[index].controlSocketFd = moverSocketFd;
        break;
      }
    }
    pthread_mutex_unlock (&GlobalVarMutex);

    /* Sleep (without blocking the process) if no free buffer/thread
     * was found
     */
    if (index == MaxConnections) {
      struct timespec delay = { 0, 500000 };

      (void) pthread_delay_np (&delay);
    }

  } while (index == MaxConnections);

  socket_setoptions (moverSocketFd);

  /* Spawn a thread to handle this transfer request
```

```
     */
    pthread_create (&Connections[index].threadId, pthread_attr_default,
                    (pthread_startroutine_t) transfer_routine,
                    (pthread_addr_t) index);
    pthread_yield();

  } /* end for */

  return;
}

/
*=====================================================================
======
 * Function:
 *   handle_signals - Routine to cause process to catch common signals
and
 *                    gracefully terminate

*=====================================================================
====*/

void handle_signals()
{
  pthread_t threadId;                /* Signal thread id */

  sigemptyset (&SigMask);
  sigaddset (&SigMask, SIGHUP);
  sigaddset (&SigMask, SIGINT);
  sigaddset (&SigMask, SIGQUIT);
```

```
  sigaddset (&SigMask, SIGTERM);


  (void) sigprocmask (SIG_SETMASK, &SigMask, (sigset_t *)NULL);


  /* Spawn a thread to catch signals

   */

  pthread_create (&threadId, pthread_attr_default,

                  (pthread_startroutine_t) signal_thread,

                  (pthread_addr_t) NULL);

  pthread_yield();

}


/
*=====================================================================
======

 * Function:

 *   signal_thread - Thread to catch signals and gracefully terminate
the

 *                   process by removing any allocated shared memory


*=====================================================================
====*/


void signal_thread()

{

  int index;

  int status = sigwait (&SigMask);


  /* Step through the connections and for any that are active, delete
the
```

```
   * shared memory segment if it exists
   */

  if (ControlOutput) printf("****** signal received ******\n");

  printf("index=%d, MaxConnections=%d\n",index,MaxConnections);

  for (index=0; index < MaxConnections; index++) {

printf("Connections[%d].active=%d\n",index,Connections[index].active);
    printf("Connections[%d].shmId=%d\n",index,Connections[index].shmId);
    if (Connections[index].active && Connections[index].shmId != -1) {
      if (ControlOutput)
        printf("Deleting shared memory for thread %d\n", index+1);
      shmctl (Connections[index].shmId, IPC_RMID, (struct shmid_ds
*)NULL);
    }
  }

  exit (status);
}

/
*=================================================================
======
 * Function:
 *   transfer_routine - Send data transfer using mover protocol
 *
```

```
 * Arguments:

 *   index  - Index into Connections array for this thread

 *

 * Return Values:

 *    <none>

*=====================================================================
====*/


void transfer_routine (int index)

{

  int                  status, tmp; /* Return, temporary values */

  int                 transferListenSocket; /* Socket listen descriptors
*/

  int                  transferSocketFd; /* Transfer accept socket */

  struct sockaddr_in  transferSocketAddr; /* Transfer socket address */

  int                  bytesSent;

  initiator_msg_t     initMessage, initReply;

  initiator_ipaddr_t  ipAddr;        /* TCP socket address info */

  initiator_shmaddr_t shmAddr;        /* Shared memory address info */

  completion_msg_t    completionMessage;

  char                 *buffer;        /* Transfer data buffer */



  if (ControlOutput)

    printf("Thread %d - Started, using control socket %d\n", index+1,

           Connections[index].controlSocketFd);


  Connections[index].shmId = -1;
```

```
transferListenSocket = transferSocketFd = -1;

buffer = NULL;

/* Loop until we reach a condition to discontinue talking with Mover
 */
while (TransferStatus == HPSS_E_NOERROR) {

  /* Get the next transfer initiation message from the Mover.
   * HPSS_ECONN will be returned when the Mover is done.
   */
  status = mvrprot_recv_initmsg (Connections[index].controlSocketFd,
                                 &initMessage);

  if (ControlOutput)
    printf("Thread %d - mvrprot_recv_initsg returned %ld\n",
           index+1, status);

  if (status == HPSS_ECONN) {
    if (ControlOutput) printf("Connection closed by mover...\n");
    break;                        /* break out of the while loop */
  }
  else if (status != HPSS_E_NOERROR) {

    hpss_error ("mvrprot_recv_initmsg returned", status);
    TransferStatus = status;
    continue;
```

```
        }

        if (ControlOutput) {
          printf ("Thread %d - Mover ready to accept ", index+1);
          print_bytes64 (initMessage.Length);
          printf (" at offset ");
          print_bytes64 (initMessage.Offset);
          printf (" via %s\n",
                  initMessage.Type == NET_ADDRESS ? "TCP" :
                  initMessage.Type == SHM_ADDRESS ? "SHM" :);
        }

        /* Tell the Mover we will send the address next
         */
        initReply.Flags = MVRPROT_COMP_REPLY | MVRPROT_ADDR_FOLLOWS;

        /* Let's agree to use the transfer protocol selected by the Mover
    and
         * let's accept the offset.  However, the number of bytes the Mover
    can
         * transfer at one time is limited by our buffer size, so we tell
    the
         * Mover how much of the data he has offerred that we are willing to
         * accept.
         */
        initReply.Type   = initMessage.Type;
        initReply.Offset = initMessage.Offset;

        if (gt64m (initMessage.Length, cast64m(BufferSize)))
```

```
      initReply.Length = cast64m(BufferSize);
    else
      initReply.Length = initMessage.Length;

    /* Send our response back to the Mover
     */
    status = mvrprot_send_initmsg (Connections[index].controlSocketFd,
                                   &initReply);

    if (status != HPSS_E_NOERROR) {
      hpss_error ("mvrprot_send_initmsg", status);
      TransferStatus = status;
      continue;
    }

    /* Based on the type of transfer protocol, allocate memory, send
address
     * information, and send the data to the HPSS Mover
     */
    switch (initMessage.Type) {

    case SHM_ADDRESS:

      /* If we have not already created the shared memory segment for
this
       * thread, do it now
       */
      if (!buffer) {
```

```
    Connections[index].shmId = shmget (IPC_PRIVATE, BufferSize,
                                       S_IRWXU | S_IRWXG | S_IRWXO);
    if (Connections[index].shmId == -1) {
      perror ("shmget");
      TransferStatus = errno;
      continue;
    }

    buffer = shmat (Connections[index].shmId, NULL, 0);

    if (!buffer) {
      perror ("shmat");
      TransferStatus = errno;
      continue;
    }

    init_buf(buffer,BufferSize);

    memset (&shmAddr, 0, sizeof(shmAddr));
    shmAddr.Flags             = 0;
    shmAddr.ShmAddr.Flags     = 0;
    shmAddr.ShmAddr.ShmID     = Connections[index].shmId;
    shmAddr.ShmAddr.ShmOffset = 0;
  }

  /* Tell the Mover what our shared memory address is
```

```
        */
      status = mvrprot_send_shmaddr (Connections[index].controlSocketFd,
                                     &shmAddr);

      if (status != HPSS_E_NOERROR) {
        hpss_error ("mvrprot_send_shmaddr", status);
        TransferStatus = status;
        continue;
      }

      /* At this point, the Mover is taking the data out of shared
memory
       * and we wait for the completion message
       */
      break;

    case NET_ADDRESS:

      /* The first time through, allocate the memory buffer and data
transfer
       * socket
       */
      if (!buffer) {
        buffer = malloc (BufferSize);
        if (!buffer) {
          perror ("malloc");
          TransferStatus = errno;
          continue;
```

```
        }

        transferListenSocket = socket (AF_INET, SOCK_STREAM, 0);

        if (transferListenSocket == -1) {
          perror ("socket");
          TransferStatus = errno;
          continue;
        }

        if (ControlOutput) {
          printf ("Thread %d - Opened transfer listen socket %d\n",
index+1,
                  transferListenSocket);
        }

        (void)memset (&transferSocketAddr, 0,
sizeof(transferSocketAddr));
        transferSocketAddr.sin_family      = AF_INET;
         transferSocketAddr.sin_port        = 0;

        /* Select the hostname (IP address) in a round-robin fashion
         */
        pthread_mutex_lock (&GlobalVarMutex);

        transferSocketAddr.sin_addr.s_addr =
HostList[CurrentHost++].ipAddr;
        if (CurrentHost == NumHosts) CurrentHost = 0;
```

```
        pthread_mutex_unlock (&GlobalVarMutex);


        if (bind (transferListenSocket,

                  (const struct sockaddr*)&transferSocketAddr,

                  sizeof(transferSocketAddr)) == -1) {

          perror ("bind");

          return;

        }


        tmp = sizeof (transferSocketAddr);


        (void)memset (&transferSocketAddr, 0,
sizeof(transferSocketAddr));


        if (getsockname (transferListenSocket,

                         (struct sockaddr *)&transferSocketAddr,

                         (size_t *)&tmp) == -1) {

          perror ("getsockname");

          TransferStatus = errno;

          continue;

        }


        if (listen (transferListenSocket, SOMAXCONN) == -1) {

          perror ("listen");

          TransferStatus = errno;

          continue;

        }
```

```
        if (VerboseOutput) {

          printf ("Thread %d - Using TCP network address
%d.%d.%d.%d:%d\n",

                  index+1,

                  (transferSocketAddr.sin_addr.s_addr & 0xff000000) >>
24,

                  (transferSocketAddr.sin_addr.s_addr & 0x00ff0000) >>
16,

                  (transferSocketAddr.sin_addr.s_addr & 0x0000ff00) >>
8,

                  (transferSocketAddr.sin_addr.s_addr & 0x000000ff),

                  transferSocketAddr.sin_port);

        }


        memset (&ipAddr,  0, sizeof(ipAddr));

        ipAddr.IpAddr.SockTransferID  = cast64m (RequestId);

        ipAddr.IpAddr.SockAddr.family = transferSocketAddr.sin_family;

        ipAddr.IpAddr.SockAddr.addr   =
transferSocketAddr.sin_addr.s_addr;

        ipAddr.IpAddr.SockAddr.port   = transferSocketAddr.sin_port;

        ipAddr.IpAddr.SockOffset      = cast64m (0);

      }


      /* Tell the Mover what socket to receive the data from

       */

      status = mvrprot_send_ipaddr (Connections[index].controlSocketFd,

                                    &ipAddr);


      if (status != HPSS_E_NOERROR) {

        hpss_error ("mvrprot_send_ipaddr", status);
```

```
          TransferStatus = status;
          continue;
        }

        /* Wait for the new Mover socket connection, if you don't already
         * have one
         */
        if (transferSocketFd == -1) {

          tmp = sizeof(transferSocketAddr);

          while ((transferSocketFd =
                  accept (transferListenSocket,
                          (struct sockaddr *)&transferSocketAddr,
                          (size_t *)&tmp)) < 0) {

            if ((errno != EINTR) && (errno != EAGAIN)) {
              TransferStatus = errno;
              break;
            }
          } /* end while */

          if (ControlOutput)
            printf("Thread %d - accept received, new transfer socket is
%d\n",
                   index+1, transferSocketFd);

          socket_setoptions (transferSocketFd);
```

```
          if (transferSocketFd < 0) continue;

        }

        /* Send the data to the Mover via our socket
         */
        status = mover_socket_send_requested_data (transferSocketFd,
                                                   cast64m (RequestId),
                                                   initMessage.Offset,
buffer,
                                                   low32m
(initReply.Length),
                                                   &bytesSent, 1);
        if (status <= 0) {
          hpss_error ("mover_socket_send_requested_data", status);
          TransferStatus = status;
        }
        break;

      default:
        break;

      } /* end switch */

      /* Get a transfer completion message from the Mover
       */
      status = mvrprot_recv_compmsg (Connections[index].controlSocketFd,
```

```
                                &completionMessage);

if (status != HPSS_E_NOERROR) {
  hpss_error ("mvrprot_recv_compmsg", status);
  TransferStatus = status;
  continue;
}

if (VerboseOutput) {
  printf ("Thread %d - ", index+1);

  print_bytes64 (completionMessage.BytesMoved);
  printf(" sent at offset ");
  print_bytes64 (initMessage.Offset);
  printf(" via ");

  switch (initMessage.Type) {
  case SHM_ADDRESS:
    printf ("SHM\n");
    break;
  case NET_ADDRESS:
    printf ("TCP\n");
    break;
  default:
    break;
  }
}
```

```
  pthread_mutex_lock (&GlobalVarMutex);
  inc64m (TotalBytesWritten, completionMessage.BytesMoved);
  if (ge64m(TotalBytesWritten,FileSize))
  {
      TransferStatus = 0;
      pthread_mutex_unlock(&GlobalVarMutex);
      break;
  }
  pthread_mutex_unlock (&GlobalVarMutex);

} /* end while loop */

if (ControlOutput) {
  printf("Closing down thread %d\n", index+1);
}

/* Clean up, based on the transfer protocol
 */
switch (initMessage.Type) {

case SHM_ADDRESS:

  /* Remove the shared memory segment if it got allocated
   */
  if (Connections[index].shmId != -1) {
    shmdt (buffer);
```

```
      shmctl (Connections[index].shmId, IPC_RMID, (struct shmid_ds
*)NULL);

      Connections[index].shmId = -1;

    }

    break;


  case NET_ADDRESS:

  default:


    /* Close down the TCP transfer socket if it got opened

     */

    if (transferSocketFd != -1) {

      (void) close (transferSocketFd);

    }


    /* Free the buffer if it was allocated

     */

    if (buffer) (void) free(buffer);


    break;


  } /* end switch */


  /* Close the control socket

   */

  (void) close (transferListenSocket);


  /* Close the control socket and mark this connection as not
```

```
     * active

     */

    pthread_mutex_lock (&GlobalVarMutex);


    (void) close (Connections[index].controlSocketFd);


    Connections[index].active = 0;


    pthread_mutex_unlock (&GlobalVarMutex);


    return;

  }


  /
  *=================================================================
  ======

   * Function:

   *   socket_setoptions - Set socket options for a given socket
  descriptor by

   *                       trying to set the send/receive buffer size to
  SbMax.

   *                       If that fails, divide SbMax by 2 until a valid
  value

   *                       is found.

   *

   * Arguments:

   *   socketFd        Socket file descriptor

   *

   * Return Values:

   *   <none>
```

```
*=====================================================================
====*/


void socket_setoptions (int socketFd)

{

  int bufferSize;

  int one = 1;


  while (SbMax > HPSS_SOCKET_BUF_MIN) {


    if ((setsockopt (socketFd, SOL_SOCKET, SO_SNDBUF, &SbMax,

                     sizeof(SbMax)) < 0) ||

        (setsockopt (socketFd, SOL_SOCKET, SO_RCVBUF, &SbMax,

                     sizeof(SbMax)) < 0))

      {

        SbMax = SbMax / 2;

        continue;

      }


    if (ControlOutput) {

      printf("Setting socket %d send/receive buffer size to ",
socketFd);

      print_bytes (SbMax);

      printf("\n");

    }

    break;

  }
```

```
  (void)setsockopt (socketFd, SOL_SOCKET, SO_KEEPALIVE, &one,
sizeof(one));

  (void)setsockopt (socketFd, IPPROTO_TCP, TCP_NODELAY, &one,
sizeof(one));



  return;

}



void init_buf(char *Buf,int Size)

{

    int cnt;

    char *bufptr;



    bufptr = Buf;

    for (cnt = 0; cnt < Size; cnt += 4)

    {

        *(int *)bufptr = cnt / 4;

        bufptr += 4;

    }

}
```

Example 4: Read List - Mover to Mover Protocol

This example is provided to illustrate use of the hpss_ReadList() function.

```
/
*=================================================================
======

 *

 * Name:

 *      readlist.c - Read an HPSS file in parallel using hpss_Readlist,
```

```
 *                    multiple data-receiving threads, and the mover
protocol

 *                    routines, supporting TCP and shared-memory data

 *                    transfers

 *

 * Disclaimer:

 *      This software is provided "as is" and may be freely copied and

 *      modified as desired.

 *

 * Usage:

 *      readlist [-vct] [-n maxConnections] [-s bufferSize] [-h
hostname]*

 *               [-p tcp|shm]* [-o] [-x sbmax] <path>

 *

 *      -v             Prints verbose output

 *      -c             Prints low-level control information

 *      -t             Print transfer rate in whole bytes per second

 *

 *      -n maxConnections

 *                    Maximum number of open Mover transfer connections

 *                     (default is DEFAULT_MAX_CONNECTIONS).

 *

 *      -s bufferSize  Buffer size used to receive data within each
transfer

 *                     thread (default is defined by
DEFAULT_BUFFER_SIZE).

 *

 *      -h hostname

 *                     Specifies one or more hostnames associated with
the
```

```
 *                      desired network interface(s) for TCP-based
transfers
 *                      (default is network associated with default
hostname).
 *                      If multiple hostnames are specified, each
transfer
 *                      threads will be assigned one of the network
interfaces
 *                      in a round-robin fashion.
 *
 *      -p tcp|shm
 *                      By default, TCP and shared memory transfers are
 *                      enabled. The -p option can be repeatedly used to
 *                     specify which transfer options to make available.
For
 *                      example, to restrict transfers to TCP only, use
 *                      "-p tcp".
 *
 *      -o              Instructs HPSS to send data to this application
in
 *                      sequential order.
 *
 *      -x sbmax        Sets the initial upper limit on the TCP socket
 *                      send/receive buffer sizes
 *
 *      <path>          The HPSS file to read.  Relative pathnames are
resolved
 *                     from the perspective of the user's home directory
 *                      within HPSS.
 *
 * Description:
```

```
 *

 *      This program reads all data stored in an HPSS file named <path>
using

 *      parallel I/O via the hpss_ReadList API and HPSS Mover protocol

 *      functions.  The program negotiates with the corresponding HPSS
Movers

 *      to determine which transfer protocol to use.  The application is

 *      coded to handle TCP and/or shared memory transfers.

 *

 *      The -v option enables the user to see each transfer of data from
a

 *      Mover, the order the data is received, and what protocol is
used.  The

 *      -c option shows control debug information.

 *

 *      The -t argument can be used to output a throughput number that
can be

 *      more easily used by other programs, spreadsheets, etc.

 *

 *      The -s argument defines the size of each memory buffer used to
receive

 *      data within each transfer thread.  If not specified, a default
value

 *      is used.

 *

 *      The -n argument defines the maximum number of simultaneous
connections

 *      to HPSS Movers, which also corresponds to the number of memory
buffers

 *      used in receiving data in parallel.  If not specified, a default
value

 *      is used.  This program will create a contiguous shared-memory
segment
```

```
 *      to receive data, the size of which is <bufferSize> times

 *      <maxConnections> (of which transfer protocol is selected).

 *

 *      Since segments of an HPSS bitfile may be striped across multiple

 *      devices (and Movers) and/or may reside at different levels in a

 *      hierarchy, multiple buffers/threads can be used to receive data
in

 *      parallel from different Movers who are trying to send data

 *      independently.

 *

 *      The -h option is used to specify an alternate hostname
interface(s) to

 *      use for TCP-based data transfers.

 *

 *      The -o option causes the HPSS_READ_SEQUENTIAL flag to be used in
the

 *      hpss_ReadList.  This means that at any point in time, the next
byte in

 *      transfer order is being processed - not waiting for a byte later
in

 *      transfer order).

 *

 *      For best performance, the buffer size should match either the VV
block

 *      size or the Mover buffer size, whichever is less, and the
maximum

 *      number of connections should be equal to the total number of
devices

 *      the file is spread across.

 *

 *      This program requires that the user already have DCE credentials
prior
```

```
 *      to invocation.
 *

*=====================================================================
====*/


#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <signal.h>

#include "hpss_api.h"
#include "u_signed64.h"
#include "mvr_protocol.h"

#include "support.h"



/* Define program default values */


#define DEFAULT_BUFFER_SIZE        (4*1048576)
#define DEFAULT_MAX_CONNECTIONS    32
#define DEFAULT_SOCKET_SBMAX       (8*1048576)
```

```
/* Define maximum values for sanity checks */


#define MAX_BUFFER_SIZE            (32*1048576)

#define MAX_MAX_CONNECTIONS        32


/* Define local function prototypes */


void manage_mover_connections ();

void transfer_routine  (int socketDes);

void socket_setoptions (int socketFd);

void signal_thread ();

void handle_signals ();


/* Define a structure of information to track for each Mover transfer

 * connection/thread

 */

typedef struct {

  int       active;             /* Whether thread/connection is active
*/

  pthread_t threadId;           /* Id of the transfer thread */

  int       controlSocketFd;    /* Control socket descriptor */

  int       shmId;              /* Shared memory segment id */

} connection_info_t;


/* Define global variables (globals start with capital letter)

 */

int         RequestId;          /* HPSS request id */

int         TransferStatus;     /* Overall status of the data transfer
```

```
                                 * (HPSS_E_NOERROR if ok) */
int         FileDes;            /* HPSS file descriptor */
int         ControlSocket = 0;  /* Central Mover connection socket */
sigset_t    SigMask;            /* Signal mask */

/* Define global variables associated with command-line options
 */
typedef struct {
  struct hostent *hostEntry;
  char            hostname[128];
  unsigned long   ipAddr;
} tcphost_t;

unsigned32  NumHosts;           /* -h argument counter */
tcphost_t  *HostList;           /* -h argument (length is NumHosts) */
int         VerboseOutput = 0;  /* -v argument (0=off, 1=on) */
int         ControlOutput = 0;  /* -c argument (0=off, 1=on) */
int         WholeBytesOutput = 0;/* -x argument (0=off, 1=on) */
unsigned32  MaxConnections;     /* -n argument */
unsigned32  BufferSize;         /* -s argument */
unsigned32  SbMax = DEFAULT_SOCKET_SBMAX; /* -x argument */

/* The following global variables are protected by the mutex
GlobalVarMutex
 */
pthread_mutex_t GlobalVarMutex;

connection_info_t  *Connections; /* Array of connection thread info */
```

```
int         CurrentHost = 0;    /* Index into HostList for next TCP
address */

u_signed64  TotalBytesRead;     /* Actual bytes received from Movers */

u_signed64  GapBytes;          /* Number of "gap" bytes within the file
*/



/
*=====================================================================
======

 * Function:

 *    hpss_error - Print out info on an HPSS error condition

 *

 * Arguments:

 *    function     Name of the HPSS function that returned "status"

 *    status       HPSS error code

 *

 * Return Values:

 *    <none>


*=====================================================================
====*/



void hpss_error (char *function, signed32 status)

{

  fprintf (stderr,"%s (%ld): %s\n", function, status, status_string
(status));

}



/
*=====================================================================
======
```

```
 * Function:

 *    terminate - Gracefully terminate the process, closing resources

 *                appropriately

 *

 * Arguments:

 *    exitStatus  Value used as the exit status

 *

 * Return Values:

 *    This function terminates the process and therefore does not

 *    return.

*=====================================================================
====*/


void terminate (int exitStatus)

{

  int index;


  /* Close down the HPSS file if it is open

   */

  if (FileDes > 0) {

    if (ControlOutput) printf("Closing HPSS file descriptor %d\n",
FileDes);

    (void)hpss_Close (FileDes);

  }


  /* Step through the connections and for any that are active, delete
the

   * shared memory segment if it exists
```

```
   */

  for (index=0; index < MaxConnections; index++) {


    if (Connections[index].active && Connections[index].shmId != -1) {

      if (ControlOutput)

        printf("Deleting shared memory for thread %d\n", index+1);

      shmctl (Connections[index].shmId, IPC_RMID, (struct shmid_ds
*)NULL);

    }

  }


  exit (exitStatus);

}


/
*=================================================================
======

 * main


*=================================================================
====*/


main (int argc, char *argv[])

{

  int                   i, badUsage; /* Counters and flags */

  size_t                tmp;    /* Temporary variables */

  char                  *programName, *s;

  IOD_t                 iod;    /* IOD passed to hpss_ReadList */

  IOR_t                 ior;    /* IOR returned from hpss_ReadList */

  srcsinkdesc_t         src, sink; /* IOD source/sink descriptors */
```

```
  struct sockaddr_in   controlSocketAddr; /* control socket addresses */

  int                  readListFlags = 0; /* Flags on hpss_ReadList call
*/

  u_signed64           fileSize; /* Size of the HPSS file */

  u_signed64           bytesMoved; /* Bytes transferred, as returned
from IOR

                                   */

  pthread_t            manageConnectionsThread; /* Spawned thread id */

  pthread_addr_t       pthreadStatus;

  signed32             status; /* HPSS return code/status */

  timestamp_t          startTime, endTime, totalTime; /* various
timestamps */


  totalTime.tv_sec = totalTime.tv_usec = 0;


  memset (&iod,  0, sizeof(iod));

  memset (&src,  0, sizeof(src));

  memset (&sink, 0, sizeof(sink));


  programName     = argv[0];

  badUsage        = 0;

  MaxConnections  = DEFAULT_MAX_CONNECTIONS;

  BufferSize      = DEFAULT_BUFFER_SIZE;

  sink.Flags      = 0;


  /* Process the arguments

   */


  while (--argc > 0 && (*++argv)[0] == '-') {
```

```
    for (s = argv[0]+1; *s != '\0'; s++) {

      switch (*s) {

      case 'v':                 /* for verbose output */
        VerboseOutput = 1;
        break;

      case 'c':                 /* for low-level control output */
        ControlOutput = 1;
        break;

      case 't':                 /* for printing whole byte throughput
rate */
        WholeBytesOutput = 1;
        break;

      case 'o':                 /* to get data in parallel sequential
order */
        readListFlags = HPSS_READ_SEQUENTIAL;
        break;

      case 'n':                 /* to specify max connections/no.
buffers */
        if (argc > 1 && (*(argv+1))[0] != '-') {
          MaxConnections = atoi((++argv)[0]);
          argc -= 1;
        }
```

```
    else
      badUsage = 1;
    break;

  case 's':                 /* to specify buffer size */
    if (argc > 1 && (*(argv+1))[0] != '-') {
      BufferSize = atobytes((++argv)[0]);
      argc -= 1;
    }
    else
      badUsage = 1;
    break;

  case 'x':                 /* to specify socket buffer sizes */
    if (argc > 1 && (*(argv+1))[0] != '-') {
      SbMax = atobytes((++argv)[0]);
      argc -= 1;
    }
    else
      badUsage = 1;
    break;

  case 'h':                 /* TCP hostname */
    if (argc > 1 && (*(argv+1))[0] != '-') {
      if (!HostList) {
        HostList = (tcphost_t *)malloc (sizeof(*HostList));
      }
```

```
            else {
             HostList = (tcphost_t *)realloc (HostList, sizeof(*HostList)
*
                                                (NumHosts + 1));
            }

            strncpy (HostList[NumHosts].hostname, (++argv)[0],
                     sizeof(HostList[NumHosts].hostname));

            /* Make sure it is a legitimate hostname */

            HostList[NumHosts].hostEntry =
              gethostbyname (HostList[NumHosts].hostname);

            if (!(HostList[NumHosts].hostEntry)) {
              fprintf (stderr, "Invalid hostname \"%s\"\n",
                       HostList[NumHosts].hostname);
              perror ("gethostbyname");
              badUsage = 1;
            }
            else {
              HostList[NumHosts].ipAddr =
                *((unsigned32*)(HostList[NumHosts].hostEntry-
>h_addr_list[0]));
              ++NumHosts;
            }

            argc -= 1;
```

```
    }
    else
      badUsage = 1;
    break;

  case 'p':                 /* transfer protocol */
    if (argc > 1) {
      ++argv;

      if (!strcmp(argv[0], "tcp")) {
        sink.Flags |= XFEROPT_IP;
      }
      else if (!strcmp(argv[0], "shm")) {
        sink.Flags |= XFEROPT_SHMEM;
      }
      else {
        printf ("Invalid transfer protocol - use tcp or shm\n");
        badUsage = 1;
      }
      argc -= 1;
    }
    else
      badUsage = 1;
    break;

  default:
    badUsage = 1;
```

```
        } /* end switch */

      } /* end for */

    } /* end while */


    if (argc != 1) badUsage = 1;


    if (badUsage) {

      printf("Usage:\n\n");

      printf("%s [-vco] [-n maxConnections] [-s bufferSize] [-h
  hostname]*\n",

             programName);

      printf("         [-p tcp|shm]* [-x sbmax] <path>\n\n");

      printf("-v\n");

      printf("\tPrints verbose output.\n\n");

      printf("-c\n");

      printf("\tPrints low-level control information.\n\n");

      printf("-t\n");

      printf("\tPrints throughput rate in whole bytes.\n\n");

      printf("-o\n");

      printf("\tInstructs HPSS to send data to this application in ");

      printf("sequential parallel\n");

      printf("\torder.\n\n");

      printf("-n maxConnections\n");

      printf("\tMaximum number of concurrent transfer threads that are ");

      printf("available for\n");

      printf("\tcommunicating simultaneously with HPSS Movers.  This ");

      printf("corresponds to the\n");
```

```
    printf("\tnumber of buffers allocated to receive HPSS data.  Default
");

    printf("is %d.\n\n", DEFAULT_MAX_CONNECTIONS);

    printf("-s bufferSize\n");

    printf("\tBuffer size used to receive data within each transfer ");

    printf("thread.  Values\n");

    printf("\tsuch as \"2mb\" can be specified.  Default is ");

    print_bytes (DEFAULT_BUFFER_SIZE);

    printf(".\n\n");

    printf("-h hostname\n");

    printf("\tSpecifies one or more hostnames associated with the
desired ");

    printf("network\n");

    printf("\tinterface(s) for TCP-based transfers (default is network
");

    printf("associated\n");

    printf("\twith default hostname).  If multiple hostnames are ");

    printf("specified, each\n");

    printf("\ttransfer thread will be assigned one of the network ");

    printf("interfaces in a\n");

    printf("\tround-robin fashion.\n\n");

    printf("-p tcp|shm\n");

    printf("\tBy default, TCP and shared memory transfers are
enabled.");

    printf("The -p option can be repeatedly used to specify which \n");

    printf("\twhich transfer options to make available. \n\n");

    printf("-x sbmax\n");

    printf("\tSets the initial upper limit on the TCP socket send/
receive ");

    printf("buffer sizes\n");
```

```
    printf("\tDefault is ");

    print_bytes (SbMax);

    printf(".\n\n");

    printf("<path>\n");

    printf("\tThe HPSS file to read.  Relative pathnames are resolved
");

    printf("from the\n");

    printf("\tperspective of the user's home directory within HPSS.\n");


    terminate (1);

  }


  /* Perform some sanity checks on values

   */

  if (MaxConnections > MAX_MAX_CONNECTIONS) {

    printf("Maximum limit on number of buffers is %d\n",
MAX_MAX_CONNECTIONS);

    terminate (1);

  }


  if (BufferSize > MAX_BUFFER_SIZE) {

    printf ("Maximum limit on buffer size is ");

    print_bytes (MAX_BUFFER_SIZE);

    printf ("\n");

    terminate (1);

  }


  /* If no transfer protocol(s) were specified, use all of them
```

```
   */
  if (!sink.Flags) {
    sink.Flags  = XFEROPT_IP | XFEROPT_SHMEM;
  }
  sink.Flags |= CONTROL_ADDR; /* optionally add "| HOLD_RESOURCES"
                               * to keep mover connections open until
                               * entire transfer completes
                               */


  /* If no hostname was specified, use the local default hostname for
TCP
   * transfers
   */
  if (!HostList) {
    HostList = (tcphost_t *) malloc (sizeof(*HostList));


    if (gethostname (HostList[0].hostname,
                     sizeof(HostList[0].hostname)) < 0) {
      perror ("gethostname");
      terminate (1);
    }


    HostList[0].hostEntry = gethostbyname (HostList[0].hostname);


    if (!(HostList[NumHosts].hostEntry)) {
      perror ("gethostbyname");
      terminate (1);
    }
```

```
  HostList[0].ipAddr =
    *((unsigned32*)(HostList[0].hostEntry->h_addr_list[0]));
  ++NumHosts;
}

/* Set up signal handling
 */
handle_signals();

/* Allocate the array of transfer/connection information
 */
Connections = (connection_info_t *) malloc (sizeof(Connections[0]) *
                                            MaxConnections);

memset (Connections, 0, sizeof(Connections[0]) * MaxConnections);



pthread_mutex_init (&GlobalVarMutex, pthread_mutexattr_default);

/* Open the HPSS file (argv[0] points to the HPSS pathname)
 */
FileDes = hpss_Open (argv[0], O_RDONLY, 0777, NULL, NULL, NULL);

if (FileDes < 0) {
  hpss_error ("hpss_Open", FileDes);
  terminate (FileDes);
}
```

```
  /* Get the file size by lseek'ing to the end of the file and seeing
what

   * position is returned.  Then lseek back to the beginning of the
file.

   */

  status = hpss_SetFileOffset (FileDes, cast64m(0), SEEK_END,

                               HPSS_SET_OFFSET_FORWARD,

                               &fileSize);

  if (status) {

    hpss_error ("hpss_SetFileOffset(end)", status);

    terminate (status);

  }


  if (VerboseOutput) {

    printf("File size is ");

    print_bytes64(fileSize);

    putchar('\n');

  }


  status = hpss_Lseek (FileDes, 0, SEEK_SET);


  if (status) {

    hpss_error ("hpss_Lseek(0)", status);

    terminate (status);

  }


  /* Create the local control socket which all Movers will initially
connect
```

```
 * to
 */
ControlSocket = socket (AF_INET, SOCK_STREAM, 0);

if (ControlSocket == -1) {
  perror ("socket");
  terminate (1);
}

if (ControlOutput) {
  printf ("Control socket is socket %d\n", ControlSocket);
}

(void)memset (&controlSocketAddr, 0, sizeof(controlSocketAddr));
controlSocketAddr.sin_family      = AF_INET;
controlSocketAddr.sin_addr.s_addr = INADDR_ANY;
controlSocketAddr.sin_port        = 0;

if (bind (ControlSocket, (const struct sockaddr*)&controlSocketAddr,
          sizeof(controlSocketAddr)) == -1) {
  perror ("bind");
  terminate (1);
}

tmp = sizeof (controlSocketAddr);

if (getsockname (ControlSocket,
```

```
                  (struct sockaddr *)&controlSocketAddr,
                  (size_t *)&tmp) == -1) {
    perror ("getsockname");
    terminate (1);
  }

  if (listen (ControlSocket, SOMAXCONN) == -1) {
    perror ("listen");
    terminate (1);
  }

  /* Start the thread to receive control connections from individual
Movers
   */
  pthread_create (&manageConnectionsThread, pthread_attr_default,
                 (pthread_startroutine_t) manage_mover_connections,
                 (pthread_addr_t)NULL);
  pthread_yield();

  /* Define source and sink descriptors and the IOD
   */
  src.SrcSinkAddr.Type = CLIENTFILE_ADDRESS;
  src.SrcSinkAddr.Addr_u.ClientFileAddr.FileDes = FileDes;

  RequestId = getpid();

  sink.SrcSinkAddr.Type = NET_ADDRESS;
  sink.SrcSinkAddr.Addr_u.NetAddr.SockTransferID = cast64m (RequestId);
```

```
  sink.SrcSinkAddr.Addr_u.NetAddr.SockAddr.addr  = HostList[0].ipAddr;

  sink.SrcSinkAddr.Addr_u.NetAddr.SockOffset     = cast64m (0);

  sink.SrcSinkAddr.Addr_u.NetAddr.SockAddr.port  =
controlSocketAddr.sin_port;

  sink.SrcSinkAddr.Addr_u.NetAddr.SockAddr.family=


controlSocketAddr.sin_family;



  iod.Function       = IOD_READ;

  iod.RequestID      = RequestId;

  iod.SrcDescLength  = 1;

  iod.SinkDescLength = 1;

  iod.SrcDescList    = &src;

  iod.SinkDescList   = &sink;



  if (ControlOutput) {

    printf("Client request id is %d\n", RequestId);

  }



  GapBytes = TotalBytesRead = bytesMoved = cast64m(0);



  startTime = get_current_timestamp();

  TransferStatus = HPSS_E_NOERROR;



  /* Loop as long as the total bytes moved plus all reported gaps are
less

   * than the total size of the file AND no transfer error has been

   * encountered

   */
```

```
while (lt64m(add64m(bytesMoved, GapBytes), fileSize) &&
       TransferStatus == HPSS_E_NOERROR) {

  /* Set the source/sink length to the number of bytes we want
   */
  src.Offset = sink.Offset = add64m(bytesMoved, GapBytes);
  src.Length = sink.Length = sub64m(fileSize, src.Offset);

  src.SrcSinkAddr.Addr_u.ClientFileAddr.FileOffset = src.Offset;

  if (ControlOutput) {
    printf ("Issuing hpss_ReadList call for ");
    print_bytes64 (sub64m(fileSize, add64m(bytesMoved, GapBytes)));
    printf ("\n");
  }

  memset (&ior,  0, sizeof(ior));

  status = hpss_ReadList (&iod, readListFlags, &ior);

  if (status) {
    hpss_error ("hpss_ReadList", status);

    if (ior.Status != HPSS_E_NOERROR) {
      hpss_error ("IOR status", ior.Status);
      printf ("Returned flags is 0x%x, bytes moved is ", ior.Flags);
      print_bytes64 (bytesMoved);
```

```
            putchar ('\n');

            terminate (1);

          }


          if (TransferStatus == HPSS_E_NOERROR) TransferStatus = status;

        }

        else {


          inc64m (bytesMoved, ior.SinkReplyList->BytesMoved);


          /* See if data transfer stopped at a gap (hole)

           */

          if (ior.Flags & IOR_GAPINFO_VALID) {


            if (VerboseOutput) {

              printf ("GAP encountered at offset ");

              print_bytes64

                          (ior.ReqSpecReply-
>ReqReply_s.ReqReply_u.GapInfo.Offset);

              printf (", length ");

              print_bytes64

                          (ior.ReqSpecReply-
>ReqReply_s.ReqReply_u.GapInfo.Length);

              putchar ('\n');

            }


            inc64m

               (GapBytes, ior.ReqSpecReply-
>ReqReply_s.ReqReply_u.GapInfo.Length);
```

```
        if (ior.ReqSpecReply) rpc_ss_client_free (ior.ReqSpecReply);
      }
      if (ior.SrcReplyList)  rpc_ss_client_free (ior.SrcReplyList);
      if (ior.SinkReplyList) rpc_ss_client_free (ior.SinkReplyList);
    }

  } /* end while */

  endTime = get_current_timestamp ();

  totalTime = diff_timestamps (startTime, endTime);

  /* Close the HPSS file
   */
  status = hpss_Close (FileDes);

  if (status < 0) {
    hpss_error ("hpss_Close", status);
  }

  /* Let's make sure that all data has actually been received before we
kill
   * any active transfer threads
   */
  pthread_mutex_lock (&GlobalVarMutex);

  if (neq64m (bytesMoved, TotalBytesRead)) {
```

```
struct timespec delay = { 1, 0 }; /* 1 second */

printf("bytesMoved is ");
print_bytes64(bytesMoved);
printf(", TotalBytesRead is ");
print_bytes64(TotalBytesRead);
printf("\n");

pthread_mutex_unlock (&GlobalVarMutex);

/* Wait for all transfer threads to complete before moving on
 */
for (tmp=0, i=0; i < MaxConnections; i++) {

  pthread_mutex_lock (&GlobalVarMutex);

  while (Connections[i].active) {

    pthread_mutex_unlock (&GlobalVarMutex);

    if ((VerboseOutput || ControlOutput) && !tmp) {
      printf ("Waiting on thread %d to complete...\n", i+1);
      tmp = 1;              /* only show the message once */
    }

    (void) pthread_delay_np (&delay);
```

```
      pthread_mutex_lock (&GlobalVarMutex);
    }
    pthread_mutex_unlock (&GlobalVarMutex);
  }
}

/* Print stats
 */
if (!eq64m (TotalBytesRead, cast64m(0))) {

  u_signed64 usecs64;
  unsigned32 throughput;

  usecs64    = add64m (mul64m (cast64m (totalTime.tv_sec), 1000000),
                       cast64m (totalTime.tv_usec));

  throughput = cast32m (div2x64m (mul64m (TotalBytesRead, 1000000),
                                  usecs64));

  if (WholeBytesOutput) {
    printf ("%ld\n", throughput);
  }
  else {
    print_bytes64 (TotalBytesRead);
    printf(" successfully read in %d.%06d sec -> ",
           totalTime.tv_sec, totalTime.tv_usec);
```

```
        print_bytes_per_second (throughput);

        putchar('\n');

        if (neqz64m (GapBytes)) {
          printf ("...");
          print_bytes64 (GapBytes);
          printf (" within gaps\n");
        }
      }
      fflush (stdout);
    }
    pthread_mutex_unlock (&GlobalVarMutex);

    /* Now cancel the manage_mover_connections thread
     */
    (void)pthread_cancel (manageConnectionsThread);
    (void)pthread_join   (manageConnectionsThread, &pthreadStatus);
    (void)pthread_detach (&manageConnectionsThread);

    terminate (0);
  }

  /
  *=================================================================
  ======
   * Function:
```

```
 *    manage_mover_connections - Accept socket connections from HPSS
Movers &

 *                               spawn a new thread to handle each Mover

 *                               connection and data transfer

 * Return Values:

 *    <none>


*=================================================================
====*/



void manage_mover_connections ()

{

  int moverSocketFd;            /* New Mover socket file descriptor */

  int index;                    /* Counters */

  int tmp;                      /* Temporary variable */


  struct sockaddr_in  socketAddr;




  /* Loop until this thread is cancelled

   */

  for (;;) {


    tmp = sizeof(socketAddr);


    while ((moverSocketFd =

            accept (ControlSocket, (struct sockaddr *)&socketAddr,

                    (size_t *)&tmp)) < 0) {
```

```
          if ((errno != EINTR) && (errno != EAGAIN)) {
            perror ("accept");
            TransferStatus = errno;
            break;
          }

        } /* end while */

        if (moverSocketFd < 0) break;

        if (ControlOutput) {
          printf ("Mover control connection accepted on control socket
%d\n",
                  moverSocketFd);
        }

        /* Find a connection/transfer thread that is free to accept this
         * connection.  If one is not free, sleep for a bit
         * and try again.
         */
        do {

          pthread_mutex_lock (&GlobalVarMutex);

          for (index = 0; index < MaxConnections; index++) {

            if (!Connections[index].active) {
              Connections[index].active = 1;
```

```
        Connections[index].controlSocketFd = moverSocketFd;
        break;
      }
    }
    pthread_mutex_unlock (&GlobalVarMutex);

    /* Sleep (without blocking the process) if no free buffer/thread
     * was found
     */
    if (index == MaxConnections) {
      struct timespec delay = { 0, 500000 };

      (void) pthread_delay_np (&delay);
    }

  } while (index == MaxConnections);

  socket_setoptions (moverSocketFd);

  /* Spawn a thread to handle this transfer request
   */
  pthread_create (&Connections[index].threadId, pthread_attr_default,
                  (pthread_startroutine_t) transfer_routine,
                  (pthread_addr_t) index);
  pthread_yield();

} /* end for */
```

```
  return;

}


/
*=====================================================================
======

 * Function:

 *   handle_signals - Routine to cause process to catch common signals
and

 *                    gracefully terminate


*=====================================================================
====*/


void handle_signals()

{

  pthread_t threadId;           /* Signal thread id */


  sigemptyset (&SigMask);

  sigaddset (&SigMask, SIGHUP);

  sigaddset (&SigMask, SIGINT);

  sigaddset (&SigMask, SIGQUIT);

  sigaddset (&SigMask, SIGTERM);


  (void) sigprocmask (SIG_SETMASK, &SigMask, (sigset_t *)NULL);


  /* Spawn a thread to catch signals

   */

  pthread_create (&threadId, pthread_attr_default,
```

```
                  (pthread_startroutine_t) signal_thread,

                  (pthread_addr_t) NULL);

  pthread_yield();

}



/
*=====================================================================
======

 * Function:

 *   signal_thread - Thread to catch signals and gracefully terminate
the

 *                   process by removing any allocated shared memory


*=====================================================================
====*/



void signal_thread()

{

  int index;

  int status = sigwait (&SigMask);



  /* Step through the connections and for any that are active, delete
the

   * shared memory segment if it exists

   */



  if (ControlOutput) printf("****** signal received ******\n");



  for (index=0; index < MaxConnections; index++) {
```

```
    if (Connections[index].active && Connections[index].shmId != -1) {

      if (ControlOutput)

        printf("Deleting shared memory for thread %d\n", index+1);

      shmctl (Connections[index].shmId, IPC_RMID, (struct shmid_ds
*)NULL);

    }

  }



  exit (status);

}



/
*=====================================================================
======

 * Function:

 *   transfer_routine - Retrieve data transfer using mover protocol

 *

 * Arguments:

 *   index  - Index into Connections array for this thread

 *

 * Return Values:

 *    <none>


*=====================================================================
====*/



void transfer_routine (int index)

{

  int                 status, tmp; /* Return, temporary values */
```

```
  int                 transferListenSocket; /* Socket listen descriptors
*/

  int                  transferSocketFd; /* Transfer accept socket */

  struct sockaddr_in  transferSocketAddr; /* Transfer socket address */

  int                  bytesReceived;

  initiator_msg_t     initMessage, initReply;

  initiator_ipaddr_t  ipAddr;   /* TCP socket address info */

  initiator_shmaddr_t shmAddr;  /* Shared memory address info */

  completion_msg_t    completionMessage;

  char                 *buffer;  /* Transfer data buffer */



  if (ControlOutput)

    printf("Thread %d - Started, using control socket %d\n", index+1,

           Connections[index].controlSocketFd);


  Connections[index].shmId = -1;

  transferListenSocket = transferSocketFd = -1;


  buffer = NULL;


  /* Loop until we reach a condition to discontinue talking with Mover

   */

  while (TransferStatus == HPSS_E_NOERROR) {


    /* Get the next transfer initiation message from the Mover.

     * HPSS_ECONN will be returned when the Mover is done.

     */
```

```
status = mvrprot_recv_initmsg (Connections[index].controlSocketFd,
                                &initMessage);

if (ControlOutput)
  printf("Thread %d - mvrprot_recv_initsg returned %ld\n",
         index+1, status);

if (status == HPSS_ECONN) {
  break;                    /* break out of the while loop */
}
else if (status != HPSS_E_NOERROR) {

  hpss_error ("mvrprot_recv_initmsg returned", status);
  TransferStatus = status;
  continue;
}

if (ControlOutput) {
  printf ("Thread %d - Mover ready to send ", index+1);
  print_bytes64 (initMessage.Length);
  printf (" at offset ");
  print_bytes64 (initMessage.Offset);
  printf (" via %s\n",
          initMessage.Type == NET_ADDRESS ? "TCP" :
          initMessage.Type == SHM_ADDRESS ? "SHM" :);
}
```

```
    /* Tell the Mover we will send the address next
     */
    initReply.Flags = MVRPROT_COMP_REPLY | MVRPROT_ADDR_FOLLOWS;

    /* Let's agree to use the transfer protocol selected by the Mover
and
     * let's accept the offset.  However, the number of bytes the Mover
can
     * transfer at one time is limited by our buffer size, so we tell
the
     * Mover how much of the data he has offerred that we are willing to
     * accept.
     */
    initReply.Type   = initMessage.Type;
    initReply.Offset = initMessage.Offset;

    if (gt64m (initMessage.Length, cast64m(BufferSize)))
      initReply.Length = cast64m(BufferSize);
    else
      initReply.Length = initMessage.Length;

    /* Send our response back to the Mover
     */
    status = mvrprot_send_initmsg (Connections[index].controlSocketFd,
                                   &initReply);

    if (status != HPSS_E_NOERROR) {
      hpss_error ("mvrprot_send_initmsg", status);
      TransferStatus = status;
```

```
        continue;
      }

      /* Based on the type of transfer protocol, allocate memory, send
address
       * information, and receive the data from the HPSS Mover
       */
      switch (initMessage.Type) {

      case SHM_ADDRESS:

        /* If we have not already created the shared memory segment for
this
         * thread, do it now
         */
        if (!buffer) {

          Connections[index].shmId = shmget (IPC_PRIVATE, BufferSize,
                                             S_IRWXU | S_IRWXG | S_IRWXO);
          if (Connections[index].shmId == -1) {
            perror ("shmget");
            TransferStatus = errno;
            continue;
          }

          buffer = shmat (Connections[index].shmId, NULL, 0);

          if (!buffer) {
```

```
        perror ("shmat");
        TransferStatus = errno;
        continue;
      }

      memset (&shmAddr, 0, sizeof(shmAddr));
      shmAddr.Flags             = HOLD_RESOURCES;
      shmAddr.ShmAddr.ShmID     = Connections[index].shmId;
    }

    /* Tell the Mover what our shared memory address is
     */
    status = mvrprot_send_shmaddr (Connections[index].controlSocketFd,
                                   &shmAddr);

    if (status != HPSS_E_NOERROR) {
      hpss_error ("mvrprot_send_shmaddr", status);
      TransferStatus = status;
      continue;
    }

    /* At this point, the Mover is moving the data into shared memory
     * and we wait for the completion message
     */
    break;

  case NET_ADDRESS:
```

```
      /* The first time through, allocate the memory buffer and data
transfer

       * socket

       */

      if (!buffer) {

        buffer = malloc (BufferSize);

        if (!buffer) {

          perror ("malloc");

          TransferStatus = errno;

          continue;

        }


        transferListenSocket = socket (AF_INET, SOCK_STREAM, 0);


        if (transferListenSocket == -1) {

          perror ("socket");

          TransferStatus = errno;

          continue;

        }


        if (ControlOutput) {

          printf ("Thread %d - Opened transfer listen socket %d\n",
index+1,

                  transferListenSocket);

        }


        (void)memset (&transferSocketAddr, 0,
sizeof(transferSocketAddr));
```

```
        transferSocketAddr.sin_family      = AF_INET;

        transferSocketAddr.sin_port        = 0;


        /* Select the hostname (IP address) in a round-robin fashion

         */

        pthread_mutex_lock (&GlobalVarMutex);


        transferSocketAddr.sin_addr.s_addr =
HostList[CurrentHost++].ipAddr;

        if (CurrentHost == NumHosts) CurrentHost = 0;


        pthread_mutex_unlock (&GlobalVarMutex);


        if (bind (transferListenSocket,

                  (const struct sockaddr*)&transferSocketAddr,

                  sizeof(transferSocketAddr)) == -1) {

          perror ("bind");

          return;

        }


        tmp = sizeof (transferSocketAddr);


        (void)memset (&transferSocketAddr, 0,
sizeof(transferSocketAddr));


        if (getsockname (transferListenSocket,

                         (struct sockaddr *)&transferSocketAddr,

                         (size_t *)&tmp) == -1) {
```

```
            perror ("getsockname");

            TransferStatus = errno;

            continue;

          }


          if (listen (transferListenSocket, SOMAXCONN) == -1) {

            perror ("listen");

            TransferStatus = errno;

            continue;

          }


          if (VerboseOutput) {

            printf ("Thread %d - Using TCP network address
%d.%d.%d.%d:%d\n",

                    index+1,

                    (transferSocketAddr.sin_addr.s_addr & 0xff000000) >>
24,

                    (transferSocketAddr.sin_addr.s_addr & 0x00ff0000) >>
16,

                    (transferSocketAddr.sin_addr.s_addr & 0x0000ff00) >>
8,

                    (transferSocketAddr.sin_addr.s_addr & 0x000000ff),

                    transferSocketAddr.sin_port);

          }


          memset (&ipAddr,  0, sizeof(ipAddr));

          ipAddr.IpAddr.SockTransferID  = cast64m (RequestId);

          ipAddr.IpAddr.SockAddr.family = transferSocketAddr.sin_family;

          ipAddr.IpAddr.SockAddr.addr   =
transferSocketAddr.sin_addr.s_addr;
```

```
  ipAddr.IpAddr.SockAddr.port   = transferSocketAddr.sin_port;
  ipAddr.IpAddr.SockOffset      = cast64m (0);
}

/* Tell the Mover what socket to send the data to
 */
status = mvrprot_send_ipaddr (Connections[index].controlSocketFd,
                              &ipAddr);

if (status != HPSS_E_NOERROR) {
  hpss_error ("mvrprot_send_ipaddr", status);
  TransferStatus = status;
  continue;
}

/* Wait for the new Mover socket connection, if you don't already
 * have one
 */
if (transferSocketFd == -1) {

  tmp = sizeof(transferSocketAddr);

  while ((transferSocketFd =
          accept (transferListenSocket,
                  (struct sockaddr *)&transferSocketAddr,
                  (size_t *)&tmp)) < 0) {
```

```
            if ((errno != EINTR) && (errno != EAGAIN)) {
              TransferStatus = errno;
              break;
            }
          } /* end while */

          if (ControlOutput)
            printf("Thread %d - accept received, new transfer socket is
%d\n",
                   index+1, transferSocketFd);

          socket_setoptions (transferSocketFd);

          if (transferSocketFd < 0) continue;

        }

        /* Receive the data from the Mover via our socket
         */
        status = mover_socket_recv_data (transferSocketFd, cast64m
(RequestId),
                                         initMessage.Offset, buffer,
                                         low32m (initReply.Length),
                                         &bytesReceived, 1);
        if (status <= 0) {
          hpss_error ("mover_socket_recv_data", status);
          TransferStatus = status;
        }
```

```
  break;

default:
  break;

} /* end switch */

/* Get a transfer completion message from the Mover
 */
status = mvrprot_recv_compmsg (Connections[index].controlSocketFd,
                               &completionMessage);

if (status != HPSS_E_NOERROR) {
  hpss_error ("mvrprot_recv_compmsg", status);
  TransferStatus = status;
  continue;
}

if (VerboseOutput) {
  printf ("Thread %d - ", index+1);

  print_bytes64 (completionMessage.BytesMoved);
  printf(" received at offset ");
  print_bytes64 (initMessage.Offset);
  printf(" via ");

  switch (initMessage.Type) {
```

```
        case SHM_ADDRESS:
          printf ("SHM\n");
          break;
        case NET_ADDRESS:
          printf ("TCP\n");
          break;
        default:
          break;
        }
      }

      pthread_mutex_lock (&GlobalVarMutex);
      inc64m (TotalBytesRead, completionMessage.BytesMoved);
      pthread_mutex_unlock (&GlobalVarMutex);

    } /* end while loop */

    if (ControlOutput) {
      printf("Closing down thread %d\n", index+1);
    }

    /* Clean up, based on the transfer protocol
     */
    switch (initMessage.Type) {

    case SHM_ADDRESS:
```

```
      /* Remove the shared memory segment if it got allocated
       */
      if (Connections[index].shmId != -1) {
        shmdt (buffer);
        shmctl (Connections[index].shmId, IPC_RMID, (struct shmid_ds
*)NULL);
        Connections[index].shmId = -1;
      }
      break;

    case NET_ADDRESS:
    default:

      /* Close down the TCP transfer socket if it got opened
       */
      if (transferSocketFd != -1) {
        (void) close (transferSocketFd);
      }

      /* Free the buffer if it was allocated
       */
      if (buffer) (void) free(buffer);

      break;

    } /* end switch */

    /* Close the control socket
```

```
    */

   (void) close (transferListenSocket);


   /* Close the control socket and mark this connection as not

    * active

    */

   pthread_mutex_lock (&GlobalVarMutex);


   (void) close (Connections[index].controlSocketFd);


   Connections[index].active = 0;


   pthread_mutex_unlock (&GlobalVarMutex);


   return;

}


/
*=================================================================
======

 * Function:

 *   socket_setoptions - Set socket options for a given socket
descriptor by

 *                       trying to set the send/receive buffer size to
SbMax.

 *                       If that fails, divide SbMax by 2 until a valid
value

 *                       is found.

 *

 * Arguments:
```

```
 *   socketFd        Socket file descriptor

 *

 * Return Values:

 *   <none>

*=====================================================================
====*/


void socket_setoptions (int socketFd)

{

  int bufferSize;

  int one = 1;


  while (SbMax > HPSS_SOCKET_BUF_MIN) {


    if ((setsockopt (socketFd, SOL_SOCKET, SO_SNDBUF, &SbMax,

                     sizeof(SbMax)) < 0) ||

        (setsockopt (socketFd, SOL_SOCKET, SO_RCVBUF, &SbMax,

                     sizeof(SbMax)) < 0))

      {

        SbMax = SbMax / 2;

        continue;

      }


    if (ControlOutput) {

      printf("Setting socket %d send/receive buffer size to ",
socketFd);

      print_bytes (SbMax);

      printf("\n");
```

```
          }

          break;

        }



        (void)setsockopt (socketFd, SOL_SOCKET, SO_KEEPALIVE, &one,
      sizeof(one));

        (void)setsockopt (socketFd, IPPROTO_TCP, TCP_NODELAY, &one,
      sizeof(one));



        return;

      }
```

Example 5: Directory Operations

This example demonstrates a simple application which calls HPSS directory functions. This program is passed the following argument:

argv[1] directory name

```
/*==============================================================================
 *
 *  Name:
 *     diropts.c - Tests directory operations for opening, reading, and
 *                 closing a directory.
 *
 *  Description:
 *     In this example, a starting directory is passed in argv[1].  This
 *     code example will recursivlely list all directories and indicate the
 *     type of object for each entry.  For files, the Class of Service and
 *     file size will be printed.  For symbolic links, the pathname to
 *     which the link points will be printed.
 *
 *------------------------------------------------------------------------------*/

#include <dirent.h>
#include "hpss_api.h"

/*==================================================================
```

```
 * print_indentation - Indent output based on an indentation level
 *==================================================================*/

void print_indentation (int indent_level)
{
  int i;

  for (i=0; i < indent_level; i++)
    printf ("   ");
}

/*==================================================================
 * list_directory - Recursively list contents of a directory
 *==================================================================*/

void list_directory (char *s, int indent_level)
{
  int rc, handle;
  struct dirent entry;
  char *path, *sympath;
  ns_Attrs_t *attrs;

  path = malloc (HPSS_MAX_PATH_NAME);
  attrs = (ns_Attrs_t *)malloc (sizeof(ns_Attrs_t));

  /* Open the directory
   */
  handle = hpss_Opendir (s);
  if (handle < 0) {
    printf ("Can't open \"%s\" (rc=%d)\n", s, handle);
    free (path);
    return;
  }

  /* Read all entries in the directory.  If an entry
   * is a directory, recursively list its contents.
   */
  do {

    rc = hpss_Readdir (handle, &entry);

    /* At the end of the directory if entry.d_namelen
     * is zero (or if we get an error).  Ignore "." and ".."
```

```
 * entries.
 */
if (rc) {
  print_indentation (indent_level);
  printf ("==> hpss_Readdir returned %d <==\n", rc);
}
else if (entry.d_namlen && strcmp(entry.d_name,".")
     && strcmp(entry.d_name,"..")) {
  print_indentation (indent_level);
  printf ("%s", entry.d_name);

  /* Determine type of file
   */
  sprintf (path, "%s/%s", s, entry.d_name);
  rc = hpss_GetListAttrs (path, attrs);

  if (!rc) {

    switch (attrs->Type) {

    case NS_DIRECTORY_OBJECT:

      printf (" (DIRECTORY)\n");
      list_directory (path, indent_level+1);
      break;

    case NS_FILE_OBJECT:
      printf (" (FILE, COS %lu, ", attrs->ClassOfService);
      print_u_signed64 (attrs->FileSize);
      printf (" bytes)\n");
      break;

    case NS_SYM_LINK_OBJECT:
      sympath = malloc(HPSS_MAX_PATH_NAME);

      rc = hpss_Readlink (path, sympath, HPSS_MAX_PATH_NAME);
      if (rc < 0)
        printf (" (SYM LINK, hpss_Readlink returned %d)\n",rc);
      else
        printf (" (SYM LINK to %s)\n", sympath);
      free(sympath);
      break;
```

```
        case NS_HARD_LINK_OBJECT:
          printf (" (HARD LINK, COS %lu, ", attrs->ClassOfService);
          print_u_signed64 (attrs->FileSize);
          printf (" bytes)\n");
          break;

        default:
          break;
        }
      }
      else {
        printf (" (FILE, hpss_GetListAttrs returned %d)\n", rc);
      }
      rc = 0;                   /* reset for test in while loop */
    }

  } while (!rc && entry.d_namlen);

  (void) hpss_Closedir (handle);

  free (path);
  return;
}

/*==================================================================
 * main
 *==================================================================*/

main (int argc, char *argv[])
{

  printf ("%s (DIRECTORY)\n", argv[1]);
  list_directory (argv[1], 1);

  exit(0);
}
```

Example 6: Get and Set File Attributes

This example demonstrates a simple application which calls HPSS get and set attributes functions.

```
/==========================================================================

 *
```

```
 *  Name:
 *      fileattrs.c - Tests the file get and set attributes functions.
 *
 *  Description:
 *      In this example, the attributes for a file will be retrieved and
 *      printed.  The comment fields and Class of Service attributes will be
 *      changed.  The attributes will then be retrieved and printed.
 *
 *-----------------------------------------------------------------------------*/

#include "hpss_api.h"
#include "u_signed64.h"

void print_attrs (char *title, hpss_fileattr_t *attrs)
{
  printf ("\n%s\n\n", title);
  printf ("       Current COS: %lu\n", attrs->BFSAttr.BfAttribMd.COSId);
  printf ("           New COS: %lu\n", attrs->BFSAttr.BfAttribMd.NewCOSId);
  printf ("            NS COS: %lu\n", attrs->NSAttr.ClassOfService);
  printf ("         File size: ");
  print_u_signed64 (attrs->BFSAttr.BfAttribMd.DataLen);
  printf ("\n");
  printf ("           Comment: %s\n", attrs->NSAttr.Comment);
  printf ("   Current Clients: %ld\n\n", attrs->BFSAttr.OpenCount);
}
/*=============================================================================
 * main
 *===========================================================================*/

main (int argc, char *argv[])
{
  int rc;
  hpss_fileattr_t attrs, modattrs;
  u_signed64 nsflags, bfsflags;

  /* Get initial file attributes
   */
  rc = hpss_FileGetAttributes ("/home/tyler/mhpcc.mail", &attrs);

  if (rc != 0) {
    printf ("hpss_FileGetAttributes returned %ld\n", rc);
    exit (1);
  }
```

```
  print_attrs ("Initial attributes...", &attrs);

  bzero ((char*)&modattrs, sizeof(modattrs));

  /* Change the comment field attached to the file
   * (a Name Server attribute)
   */
  strncpy (modattrs.NSAttr.Comment,
      "Performance analysis output from 091796 run",
      HPSS_MAX_COMMENT_LENGTH);

  nsflags = orbit64m (cast64m(0), ATTRINDEX_COMMENT);

  /* Change the COS (a BFS attribute)
   */
  modattrs.BFSAttr.BfAttribMd.COSId = atol(argv[1]);
  bfsflags = orbit64m (cast64m(0), BFS_SET_COS_ID);

  rc = hpss_FileSetAttributes ("/home/tyler/mhpcc.mail",
                 nsflags, bfsflags,
                 &modattrs, &attrs);

  if (rc) {
    printf ("hpss_FileSetAttributes returned %d\n", rc);
  }
  else {
    rc = hpss_FileGetAttributes ("/home/tyler/mhpcc.mail", &attrs);

    if (rc != 0) {
      printf ("hpss_FileGetAttributes returned %ld\n", rc);
      exit (1);
    }

    print_attrs ("After changing comment and COS id...", &attrs);
  }

  exit (0);
}
```

Example 7: Link Operations

This example demonstrates a simple application which calls HPSS link functions.

```
/*==============================================================================
 *
 *  Name:
 *      links.c - Tests link operations on a file.
 *
 *  Description:
 *      In this example, a file is created and written to.  A symbolic link
 *      and hard link are created to the file.  The HPSS stat and lstat
 *      functions are called for each of the 3 files, and the status is
 *      printed.  The files are then deleted.
 *
 *------------------------------------------------------------------------------*/

#include "hpss_api.h"

/*==============================================================================
 * stat_file
 *==============================================================================*/

void stat_file (char *path)
{
  int rc;
  struct stat buffer;

  rc = hpss_Stat (path, &buffer);

  if (rc < 0) {
    printf ("hpss_Stat on \"%s\" returned %d\n", path, rc);
  }
  else {
    printf ("hpss_Stat on \"%s\":\n", path);
    printf ("\t        Size: %ld\n", buffer.st_size);
    printf ("\t        Mode: %o\n", buffer.st_mode);
    printf ("\tNum of Links: %u\n\n", buffer.st_nlink);
  }
  return;
}

/*==============================================================================
 * lstat_file
 *==============================================================================*/
```

```
void lstat_file (char *path)
{
  int rc;
  struct stat buffer;

  rc = hpss_Lstat (path, &buffer);

  if (rc < 0) {
    printf ("hpss_Lstat on \"%s\" returned %d\n", path, rc);
  }
  else {
    printf ("hpss_Lstat on \"%s\":\n", path);
    printf ("\t        Size: %ld\n", buffer.st_size);
    printf ("\t        Mode: %o\n", buffer.st_mode);
    printf ("\tNum of Links: %u\n\n", buffer.st_nlink);
  }
  return;
}

/*==============================================================================
 * main
 *==============================================================================*/

main (int argc, char *argv[])
{
  int fd, rc;
  ssize_t bytes_written;
  char buffer[1024];

  /* Delete the testfiles, in case they exist (ignore errors)
   */
  (void) hpss_Unlink ("testfile");
  (void) hpss_Unlink ("testfile.symlink");
  (void) hpss_Unlink ("testfile.hardlink");

  /* Create the base file and write 1024 bytes to it.
   */
  fd = hpss_Open ("testfile", O_CREAT | O_WRONLY, 0644,
         NULL, NULL, NULL);

  if (fd < 0) {
    printf ("hpss_Open returned %ld\n", fd);
```

```
    exit (1);
  }

  bytes_written = hpss_Write (fd, buffer, sizeof(buffer));

  (void) hpss_Close (fd);

  if (bytes_written != sizeof(buffer)) {
    printf ("hpss_Write returned %ld\n", bytes_written);
    exit (1);
  }

  /* Create symbolic link "testfile.symlink" that points
   * to "testfile"
   */
  rc = hpss_Symlink ("testfile.symlink", "testfile");

  if (rc != 0) {
    printf ("hpss_Symlink returned %ld\n", rc);
    exit (1);
  }

  /* Create hard link "testfile.hardlink" that points
   * to "testfile"
   */
  rc = hpss_Link ("testfile", "testfile.hardlink");

  if (rc != 0) {
    printf ("hpss_Link returned %ld\n", rc);
    exit (1);
  }

  /* Now call hpss_Stat on each file
   */
  stat_file ("testfile");
  stat_file ("testfile.symlink");
  stat_file ("testfile.hardlink");

  printf("--------------------\n");

  /* Now call hpss_Lstat on each file
   */
  lstat_file ("testfile");
```

```
  lstat_file ("testfile.symlink");
  lstat_file ("testfile.hardlink");

  /* Delete test files (ignore errors)
   */
  (void) hpss_Unlink ("testfile");
  (void) hpss_Unlink ("testfile.symlink");
  (void) hpss_Unlink ("testfile.hardlink");

  exit (0);
}
```

Example 8: Get File System Information for a Class of Service

This example demonstrates a simple application which calls HPSS functions to get file system information for a specified Class of Service. This program is passed the following argument:

argv[1]	Class of Service id

```
/*==============================================================================
 *
 *  Name:
 *     statfs.c - Tests the capability to get file system information for
 *                a particular Class of Service.
 *
 *  Description:
 *     In this example, a Class of Service is passed in argv[1].  The total
 *     space, used space, and free space statistics are printed.
 *
 *------------------------------------------------------------------------------*/

#include "hpss_api.h"
#include "u_signed64.h"

main (int argc, char *argv[])
{
  int rc;
  struct statfs stats;
  u_signed64 total_space, used_space, free_space;

  rc = hpss_Statfs (atol(argv[1]), &stats);

  if (!rc) {
```

```
    total_space = mul64m (cast64m(stats.f_blocks),
             (unsigned32)stats.f_bsize);
    free_space = mul64m (cast64m(stats.f_bfree),
            (unsigned32)stats.f_bsize);

    used_space = sub64m (total_space, free_space);

    printf ("COS %d Stats:\n");
    printf ("   Total Bytes: ");
    print_u_signed64 (total_space);
    printf ("\n    Bytes Used: ");
    print_u_signed64 (used_space);
    printf ("\n    Bytes Free: ");
    print_u_signed64 (free_space);
    printf ("\n");

    /*
    printf ("Blocksize: %ld\n", stats.f_bsize);
    printf ("Blocks: %ld\n", stats.f_blocks);
    printf ("Free Blocks: %ld\n", stats.f_bfree);
    printf ("Files: %ld\n", stats.f_files);
    printf ("Free files: %ld\n", stats.f_ffree);
    printf ("Fsize: %ld\n", stats.f_fsize);
    printf ("Fname: %s\n", stats.f_fname);
    */
  }
}
```

Example 9: Get and Set the Client API Library Configuration

This example demonstrates a simple application which calls HPSS functions to get and set Client API configuration settings.

```
/*==============================================================================
 *
 *  Name:
 *     api_config.c - Tests the capability to get and set configuration
 *                    settings associated with the Client API library.
 *
 *  Description:
 *     In this example, the current Client API Library configuration settings
 *     are retrieved and printed.  The transfer type and host name are then
 *     changed.  The configuration settings are then retrieved and printed
 *     to show the new values.
```

```
 *
 *-----------------------------------------------------------------------------*/

#include "hpss_api.h"
#include "api_internal.h"

/*=================================================================
 * print_api_config
 *=================================================================*/

void print_api_config ( api_config_t *config )
{
  printf ("Flags: %lx\n", config->Flags);
  printf ("DebugValue: %ld\n", config->DebugValue);
  printf ("TransferType: ");

  switch (config->TransferType) {
  case API_TRANSFER_TCP:
    printf("TCP\n");
    break;
  default:
    break;
  }

  printf ("NumRetries: %ld\n", config->NumRetries);
  printf ("MaxConnections: %ld\n", config->MaxConnections);
  printf ("ServerName: %s\n", config->ServerName);
  printf ("HPNSServiceName: %s\n", config->HPNSServiceName);
  printf ("BFSServiceName: %s\n", config->BFSServiceName);
  printf ("PrincipalName: %s\n", config->PrincipalName);
  printf ("KeytabPath: %s\n", config->KeytabPath);
  printf ("DebugPath: %s\n", config->DebugPath);
  printf ("HostName: %s\n", config->HostName);

  return;
}

/*=================================================================================
 * main
 *=================================================================================*/

main (int argc, char *argv[])
{
```

```
  long rc;
  api_config_t config;

  rc = hpss_GetConfiguration (&config);

  if (!rc) {
    print_api_config(&config);

    printf ("\nNow setting xfer to TCP and host name\n");

    strcpy (config.HostName, "hpss-f");
    config.TransferType = API_TRANSFER_TCP;

    rc = hpss_SetConfiguration (&config);

    if (!rc)
      rc = hpss_GetConfiguration (&config);

      if (!rc)
        print_api_config(&config);

    else
      printf("hpss_SetConfiguration returned %ld\n", rc);
  }
}
```

Appendix B Makefile Example

The following makefiles give examples of how HPSS Client API applications should be built on all the different supported platforms. The example is in the form of a makefile system that was designed to allow the user to supply the platform name as an argument to the make utility. The example consists of a main makefile and several include makefiles that contain the platform specific compiler directives. The header of the main makefile demonstrates how the multi-platform make is invoked.

Multi-Platform Makefile Example

B.1 Main Makefile:

```
#==============================================================================
# Makefile -
#  This is an example makefile that can be used for building an HPSS
# client API application on a number of different platforms. The client
# program (‘client_program’) is built by running the make utility in
# the following manner:
#
# $ make PLATFORM=<platform> all
#
# Where platform is the name of the platform that the client
# application is targeted for..
#
# The following platform files are supplied with this example:
#    aix42-43.mk - AIX 4.2 and 4.3 systems
#    aix42-nodce.mk - Non-DCE AIX 4.2 systems
#    aix43-nodce.mk - Non-DCE AIX 4.3 systems
#    solaris26.mk - Solaris 2.6 systems
#    solaris26-nodce.mk - Non-DCE Solaris 2.6 systems
#    irix64-nodce.mk - Non-DCE IRIX 6.4 systems
#
#==============================================================================
```

```
include $(PLATFORM).mk

INCLUDES=-I. -I$(PLATFORM_ROOT_DIR)/include \
     -I$(PLATFORM_ROOT_DIR)/include/dmapi/dmg

LIBS=$(PLATFORM_ROOT_DIR)/lib/libhpss.a $(PLATFORM_LIBS)

CFLAGS=$(PLATFORM_CFLAGS) $(INCLUDES) $(PLATFORM_INCLUDES)

LDFLAGS=$(PLATFORM_LDFLAGS)

CC=$(PLATFORM_CC)

.c.o:;@echo “  Compiling $< ...”
    $(CC) $(CFLAGS) -c $<

PROGRAM=client_program

all:   $(PROGRAM)

$(PROGRAM):$$@.o
       @echo “  Linking $@ ...”
       @$(CC) $(LDFLAGS) -o $@ $@.o $(LIBS)

clean:;
       @echo “  Removing $@ ...”
       @rm -f $(PROGRAM) $(PROGRAM).o
```

B.2 Include Makefile for AIX 4.2 and 4.3:

```
#==============================================================================
#    aix42-43.mk - AIX 4.2 and 4.3 systems
#==============================================================================

PLATFORM_ROOT_DIR= /opt/hpss
PLATFORM_CC= xlc_r4
PLATFORM_LIBS=-ldce -lEncSfs -lEncina
PLATFORM_INCLUDES=-I/usr/lpp/encina/include
PLATFORM_CFLAGS= \
    -D_THREAD_SAFE -D_AIX32_THREADS=1 -D_AES_SOURCE \
```

```
    -D_AIX41 -D_AIX -D_AIX32 -D_IBMR2 \
    -I/usr/include/dce
```

B.3 Include Makefile for Non-DCE AIX 4.2:

```
#==============================================================================
#    aix42-nodce.mk - Non-DCE AIX 4.2 systems
#==============================================================================

PLATFORM_ROOT_DIR= /opt/hpss_nodce
PLATFORM_CC= xlc_r
PLATFORM_LIBS= -lpthreads
PLATFORM_CFLAGS= \
    -DNO_DCE -DIDLBASE_H -Dnbase_v0_0_included \
    -Dhpss_idl_types_v0_0_included -DTHREADS_ENABLED \
    -DTHREAD_SAFE -D_AIX -D_IBMR2
```

B.4 Include Makefile for Non-DCE AIX 4.3:

```
#==============================================================================
#    aix43-nodce.mk - Non-DCE AIX 4.3 systems
#==============================================================================

PLATFORM_ROOT_DIR= /opt/hpss_nodce
PLATFORM_CC= xlc_r7
PLATFORM_LIBS= -lpthreads
PLATFORM_CFLAGS= \
    -DNO_DCE -DIDLBASE_H -Dnbase_v0_0_included \
    -Dhpss_idl_types_v0_0_included -DTHREADS_ENABLED \
    -DTHREAD_SAFE -D_AIX -D_IBMR2
```

B.5 Include Makefile for Non-DCE Solaris 2.6:

```
#==============================================================================
```

```
#    solaris26-nodce.mk - Non-DCE Solaris 2.6 systems
#==============================================================================

PLATFORM_ROOT_DIR= /opt/hpss4112_sun_ndc
PLATFORM_CC= ucbcc -mt -v -D_THREAD_SAFE -z muldefs
PLATFORM_LIBS= -lpthread -lnsl -lxnet -lsocket
PLATFORM_CFLAGS= \
    -DNO_DCE -DIDLBASE_H -Dnbase_v0_0_included \
    -Dhpss_idl_types_v0_0_included -DTHREADS_ENABLED \
    -DTHREAD_SAFE -D_XOPEN_SOURCE -D_XOPEN_SOURCE_EXTENDED=1 \
     -D__EXTENSIONS__
```

B.6 Include Makefile for Solaris 2.6:

```
#==============================================================================
#    solaris26.mk - Solaris 2.6 systems
#==============================================================================

PLATFORM_ROOT_DIR= /opt/hpss
PLATFORM_CC= ucbcc -mt -v -D_THREAD_SAFE -I/usr/include/dce -z muldefs
PLATFORM_LIBS= -lnsl -lxnet -lsocket -ldce -lEncina
PLATFORM_INCLUDES= -I/opt/encina/include
PLATFORM_CFLAGS= \
    -D_XOPEN_SOURCE -D_XOPEN_SOURCE_EXTENDED=1 -D__EXTENSIONS__
```

B.7 Include Makefile for Non-DCE IRIX 6.4:

```
#==============================================================================
#    irix64-nodce.mk - Non-DCE IRIX 6.4 systems
#==============================================================================

PLATFORM_ROOT_DIR= /opt/hpss
PLATFORM_CC= cc
PLATFORM_LIBS= -lpthread
PLATFORM_CFLAGS= \
    -DNO_DCE -DIDLBASE_H -Dnbase_v0_0_included \
    -Dhpss_idl_types_v0_0_included -DTHREADS_ENABLED \
```

```
-DTHREAD_SAFE -mips4 -woff 1171,1116,1233,1047,1183,1204,1110
```

Appendix C Notes

This section provides example IOD scenarios.

Usage of IOD/IOR

The IOD is designed to allow lists of source and sink descriptors to accommodate random data access within a single request. For large data transfers, these lists may be extremely long. The *stripeaddress* structure is intended to allow a shorthand notation to be used for describing data that is striped across a number of data addresses. Currently, these addresses will be either device addresses or network (IP) addresses.

The *stripeaddress* structure defines the amount of contiguous data that is written to each element in a stripe group (the *BlockSize* field), the number of elements in the stripe group (the *StripeWidth* field), and addressing information describing the elements of the stripe group (the *AddrListLength* and *StripeAddrList* fields).

The addressing information will be used in two ways. The first is by the initiator of a request, to specify the addressing information for each element in the stripe group. This information will then be passed to the responder who will use that information, along with the block size and stripe group width, to determine which element to contact for each piece of the transfer. In this case, the *AddrListLength* field will be equal to the *StripeWidth* field. The second way in which the addressing will be used is by the responding Storage Server to describe to a Mover the part of the transfer for which it is responsible. The block size and stripe group width information (along with the offset in the source/sink descriptor) allows the Mover to calculate for which parts of the overall transfer it has responsibility. In this case, the *AddrListLength* field will be equal to 1 (one).

As an example, consider a 100 MB read request from a client which will stripe the data across 2 I/O nodes, with a stripe block size of 4 MB. The I/O nodes will be *NodeA* and *NodeB*; the file in question *Bfid1*, at offset 0. The client would build an IOD which would look like:

```
IOD:
   RequestID:Req1
   Function: READ
   SrcDescLength:1
   SinkDescLength:1
   SrcDescList:
      0:
```

```
            Offset:  0
            Length:100 MB
            SrcSinkAddress:
               Type:  FILEADDRESS
               FileAddr:
                  BitfileID:Bfid1
                  BitfileOffset:0
      SinkDescList:
         0:
            Offset:0
            Length:100 MB
            SrcSinkAddress:
               Type:  STRIPEADDRESS
               StripeAddr:
                  BlockSize:4MB
                  StripeWidth:2
                  AddrListLength:2
                  StripeAddrList:
                     0:
                       Type:NETADDRESS
                         NetAddr:
                            SockTransferID:<transfer id>
                            SockAddr:<addr for NodeA>
                            SockOffset:0
                     1:
                       Type:NETADDRESS
                         NetAddr:
                            SockTransferID:<transfer id>
                            SockAddr:<addr for NodeB>
                            SockOffset:0
```

The client sends the IOD to the Bitfile Server. The Bitfile Server maps the bitfile addressing information into logical segment information. For simplicity, assume that the data falls in one logical segment (an example of splitting the source descriptor will come when we get down to the physical volume addresses, below). The 100 MB is stored in *Lseg1*, beginning at offset 0. The update IOD would look like:

```
   IOD:
      RequestID:       Req1
      Function:        READ
      SrcDescLength:   1
      SinkDescLength:  1
      SrcDescList:
```

```
        0:
            Offset:    0
            Length:    100 MB
            SrcSinkAddr:
                Type:  LSEGADDRESS
                LsegAddr:
                    LSegID:Lseg1
                    LSegOffset:0
    SinkDescList:
            <Same as SinkDescList received from Client>
```

The Bitfile Server sends the IOD on to the Storage Server. The Storage Server maps the logical segment information in the source descriptor into virtual volume addressing information. The data is stored on virtual volume *Vvol 1*, beginning at offset 0. The updated source descriptor would look like:

```
    SrcDescList:
        0:
            Offset:      0
            Length:      100 MB
            SrcSinkAddr:
                Type:        VVOLADDRESS
                VVolAddr:
                    VVolID:         Vvol1
                    VVolOffset:     0
```

The storage server then maps the virtual volume information into physical volume addressing information. Assume *Vvol1* is striped across 3 physical volumes: *PVol1*, *PVol2* and *PVol3*; each beginning at offset 64 KB (which could be the first addressable by after a volume label on tape), with stripe block size 1MB. The source descriptor would then look like:

```
SrcDescList:
    0:
        Offset:      0
        Length:      100 MB
        SrcSinkAddr:
            Type:  STRIPEADDRESS
            StripeAddr:
                BlockSize:      1 MB
                StripeWidth:    3
                AddrListLength: 3
                StripeAddrList:
                    0:
                        Type:PVOLADDRESS
                        PVolAddr:
```

```
                PVolName:     PVol1
                PVolOffset:   64 KB
            1:
                Type:PVOLADDRESS
                PVolAddr:
                PVolName:     PVol2
                PVolOffset:   64 KB
            2:
                Type:PVOLADDRESS
                PVolAddr:
                PVolName:     PVol3
                PVolOffset:   64 KB
```

The Storage Server would then, after mounting removable media, map the physical volume information into device addressing information. Assume the physical volumes are mounted on devices *Dev1*, *Dev2*, and *Dev3*, respectively and the position for each device is 64KB from the beginning of the tape. The Movers associated with the three devices are *Mvr1*, *Mvr2*, and *Mvr3*, respectively. An IOD would be built for each Mover, every field would be the same as in the IOD received by the storage server with the exception of the *SrcDescLength* and the *SrcDescList*. Note that the *SrcDescLength* field may be the same as was passed to the Storage Server or it may be different (e.g., the Storage Server may include an extra source descriptor to one or more of the Movers if the request does not start on a stripe boundary, so that the remaining descriptor(s) can refer to a stripe aligned piece of data). The source descriptors would look like:

```
For Mvr1:
   SrcDescLength:      1
   SrcDescList:
      0:
         Offset:      0
         Length:      34 MB
         SrcSinkAddr:
            Type:  STRIPEADDRESS
            StripeAddr:
               BlockSize:      1 MB
               StripeWidth:    3
               AddrListLength: 1
               StripeAddrList:
                  0:
                     Type:  DEVADDRESS
                     DevAddr:
                       DeviceID:Dev1
                       DevicePosition:
                         Whence:                0
                         Granularity:bytes
```

```
                        Offset:                 64KB
                        AbsolutePosition:0
For Mvr2:
   SrcDescLength:      1
   SrcDescList:
      0:
         Offset:      1 MB
         Length:      3 MB
         SrcSinkAddr:
            Type:  STRIPEADDRESS
            StripeAddr:
               BlockSize:      1 MB
               StripeWidth:    3
               AddrListLength: 1
               StripeAddrList:
                  0:
                     Type:  DEVADDRESS
                     DevAddr:
                       DeviceID:        Dev2
                       DevicePosition:
                         Whence:                 0
                         Granularity:bytes
                         Offset:                 64KB
                         AbsolutePosition:0
For Mvr3:
   SrcDescLength:      1
   SrcDescList:
      0:
         Offset:      2 MB
         Length:      33 MB
         SrcSinkAddr:
            Type:  STRIPEADDRESS
            StripeAddr:
               BlockSize:      1 MB
               StripeWidth:    3
               AddrListLength: 1
               StripeAddrList:
                  0:
                     Type:  DEVADDRESS
                     DevAddr:
```

```
DeviceID:          Dev3
DevicePosition:
  Whence:                0
  Granularity:bytes
  Offset:                64KB
  AbsolutePosition:0
```

Given the information in the source descriptor describing the part of the transfer for which it is responsible, the Mover uses the *Offset* field in *SrcDescList[0]* to determine the offset into the transfer into which its part of the data begins, and uses the *BlockSize* and *StripeWidth* fields in *StripeAddr* to determine the stride at which the blocks are allocated to it - e.g., *Mvr1* begins at offset 1 MB into the transfer and then is responsible to 1 MB blocks with a 3 MB stride (1 MB *BlockSize* * 3 *StripeWidth*). The Mover uses the information contained in the sink descriptor to perform a similar calculation to determine which client Mover to contact and what the format of the parallel data tag should be for the appropriate part of the transfer.

Each Mover then builds an IOR to return status to the Storage Server. Assuming the data transfer completed successfully, the IORs would look like:

```
For Mvr1:
   RequestID:x
   Flags:       COMPLETED
   Status:      0
   DevSpecReply:   NULL
   SrcReplyLength:  1
   SinkReplyLength: 1
   SrcReplyList:
      0:
         Flags:       COMPLETED | POSITIONVALID
         Status:      0
         BytesMoved:  34 MB
         Position:
            Whence:         0
            Granularity:    Bytes
            Offset:         64 KB
            AbsolutePosition:0
   SinkReplyList:
      0:
         Flags:    COMPLETED
         Status:      0
         BytesMoved:  34 MB

For Mvr2:
```

```
   RequestID:          x
   Flags:              COMPLETED
   Status:             0
   DevSpecReply:       NULL
   SrcReplyLength:     1
   SinkReplyLength:    1
   SrcReplyList:
      0:
         Flags:       COMPLETED | POSITIONVALID
         Status:      0
         BytesMoved:  33 MB
         Position:
            Whence:         0
            Granularity:    Bytes
            Offset:         64 KB
            AbsolutePosition:0
   SinkReplyList:
      0:
         Flags:       COMPLETED
         Status:      0
         BytesMoved:  33 MB

For Mvr3:
   RequestID:          x
   Flags:              COMPLETED
   Status:             0
   DevSpecReply:       NULL
   SrcReplyLength:     1
   SinkReplyLength:    1
   SrcReplyList:
      0:
         Flags:       COMPLETED | POSITIONVALID
         Status:      0
         BytesMoved:  33 MB
         Position:
            Whence:         0
            Granularity:    Bytes
            Offset:         64 KB
            AbsolutePosition:0
   SinkReplyList:
```

```
        0:
            Flags:        COMPLETED
            Status:       0
            BytesMoved:   33 MB
```

The Storage Server receives the IORs and maps the status information into physical volume status, virtual volume status, and eventually logical volume status. The IOR returned to the Bitfile Server would look like:

```
RequestID:          Req1
Flags:              COMPLETED
Status:             0
DevSpecReply:       NULL
SrcReplyLength:     1
SinkReplyLength:    1
SrcReplyList:
    0:
        Flags:        COMPLETED
        Status:       0
        BytesMoved:   100 MB
SinkReplyList:
    0:
        Flags:        COMPLETED
        Status:       0
        BytesMoved:   100 MB
```

The Bitfile Server receives the IOR and maps the source reply list into bitfile information that the client will understand. The IOR returned to the client would look as follows. Since we had only one logical segment, the IOR structures are actually identical.

```
RequestID:          Req1
Flags:              COMPLETED
Status:             0
DevSpecReply:       NULL
SrcReplyLength:     1
SinkReplyLength:    1
SrcReplyList:
    0:
        Flags:        COMPLETED
        Status:       0
        BytesMoved:   100 MB
```

```
SinkReplyList:
   0:
      Flags:       COMPLETED
      Status:      0
```

```
BytesMoved:  100 MB
```

Usage of IOD/IOR with Mover Protocol

For Release 3, IOD/IOR modifications are made to:

- Support disk devices (addition of **DEVICE_CLEAR** device specific request).
- Support intermediate (Mover) replies, containing listen port addressing information.
- Support specification of the use of a Mover-to-Mover data transfer control protocol. This protocol supports dynamic data transport selection and Mover-to-Mover flow control over a potentially very large data transfer.

The modification to support disk devices was not extensive. Some of the currently used fields defined in the IOD/IOR, notably *BlocksBetweenTapeMarks* will be unused when making requests that involve disk device addresses. The only addition is a device specific command, **DEVICE_CLEAR**, which can be used to request that a section of the disk media be zeroed out.

To support an intermediate reply made from a Mover after it has put out listen ports for the control dialogue in the Mover-to-Mover protocol (mentioned below), it was necessary to add a *ReplyAddr* field to the request specific information in the IOD. Note that it is not a restriction on the intermediate reply that it be used with the Mover-to-Mover protocol, but that is the primary use.

To support the Mover-to-Mover protocol, an additional bit value is added to the IOD *Flags* field, **CONTROL_ADDR**, which indicates that the data transfer should use the Mover-to-Mover protocol, and indicates that the addressing information present in this source/sink descriptor is the peer Mover's control listen port addressing information.

A brief example follows to illustrate the use of the additional capabilities for the Mover-to-Mover protocol:

The **BFS** decides to move data from a single tape to a single disk (for simplicity's sake). The tape is controlled by the storage server **tape_ss** and the Mover **tape_mvr**. The disk is controlled by the storage server **disk_ss** and the Mover **disk_mvr**.

1. **BFS** creates a listen port and spawns a task to wait for the Mover intermediate replies (note that in the general case, the **BFS** may receive many such replies).
2. **BFS** builds a WRITE IOD, specifying **REPLYWHENREADY** in the IOD *Flags* field. The IOD contains a NULL source descriptor list. **BFS** sends the IOD to **disk_ss**.
3. **disk_ss** translates the sink descriptor list and forwards the IOD to **disk_mvr**.
4. **disk_mvr** puts out a listen, places the address in an IOR and replies to the address specified in the request specific information of the IOD (which corresponds to the listen put out by **BFS** in step (1)). The src/sink descriptors returned will specify **CONTROL_ADDR** in the *Flags* field to indicate that they represent control ports (i.e., Mover protocol messages will be expected). Also included will be indication of **disk_mvr's** data transfer options.
5. **BFS** receives the Mover listen port information, and uses it in the sink descriptor list of a READ IOD, which is sent to **tape_ss**.
6. **tape_ss** translates the source descriptor and forwards the request to **tape_mvr**.

7. **tape_mvr** connects to the listen port put out by **disk_mvr** in step (4), and sends indication of the transport option (IP, etc.) selected, and the offset and length of the first piece of the transfer to be sent.

8. **disk_mvr** responds with data port listen addresses, maximum length which it can handle before more control protocol dialogue must take place, and transfer identifiers, if necessary (these will likely be included in the data port addresses).

9. **tape_mvr** connects to data ports and sends data (with tag information, if necessary).

10. Steps (7) through (9) are repeated until the end of the transfer.

11. **tape_mvr** replies to **tape_ss**, which in turn replies to **BFS**. **BFS** drops the Mover reply connection established with **disk_mvr** in step (4), to indicate that the request is complete. Note that the disk Movers may have already completed, if their part of the transfer has completed.

12. **disk_mvr** responds to **disk_ss**, which in turn responds to **BFS**.

Appendix D Acronyms

ACL	Access Control List
ACLS	Automated Cartridge System Library Software (Science Technology Corporation)
AIX	Advanced Interactive Executive
API	Application Program Interface
BFS	Bitfile Server
CDS	Cell Directory Server
DCE	Distributed Computing Environment
DFS	Distributed File System
EFS	External File System
FTP	File Transfer Protocol
gid	Group Identifier
HIPPI	High Performance Parallel Interface
HPSS	High Performance Storage System
IBM	International Business Machines Corporation
ID	Identifier
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input / Output
IOD	I/O Descriptor
IOR	I/O Reply
IP	Internet Protocol
LaRC	Langley Research Center
LANL	Los Alamos National Laboratory
LLNL	Lawrence Livermore National Laboratory
MPI-IO	Message Passing Interface - Input / Output
NASA	National Aeronautics and Space Administration
NFS	Network File System
NS	Name Server
ORNL	Oak Ridge National Laboratory
OSF	Open Software Foundation
PFTP	Parallel File Transfer Protcol
POSIX	Portable Operating System Interface for computer environments (an IEEE operating system standard)
RISC	Reduced Instruction Set Computer
SFS	Structured File Server
SNL	Sandia National Laboratories
SP	Scalable Processor
TCP	Transmission Control Protocol
uid	User Identifier
VV	Virtual Volume
XFS	XFS is not an acronym. It is the full name of a filesystem from SGI. In this document, we

are referring to the Open Source version of XFS available for Linux.

Appendix E References

1. *File Transfer Protocol, RFC-0959,* October 1985.

2. *IEEE Std 1003.1-1990 Standard for Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Programming Interface (API).*

3. *HPSS Error Messages Manual,* May 2002.

4. *HPSS Programmer's Reference Guide, Volume 2,* May 2002.

5. *HPSS Installation Guide,* May 2002.

6. *HPSS Management Guide,* May 2002.

7. *HPSS User's Guide,* May 2002.

8. *Institute of Electrical and Electronics Engineers (IEEE) Mass Storage System Reference Model (MSSRM) (Version 5).*

9. *Network File System Specification, RFC-1094,* DDN Network Information Center, SRI International, Menlo Park, Ca.

10. *OSF DCE User's Guide amd Reference,* Prentice Hall, Englewood Cliffs, N. J.

11. *P1003.1a Draft Revision to Information Technology - POSIX Part 1: System Application Program Interface (API) [C Language].*

Appendix F MPI-IO Programming Examples

Coding examples for opening, setting views, reading, writing, and closing HPSS MPI-IO files are provided below. Both collective and noncollective read and write examples are included.

Sample Makefile for Example Code

The following Makefile may be used to compile and load example code, where the source to be compiled is generically named **mpio_example.c**. The Makefile may be used for either AIX or Solaris, and for any of the three host MPIs available across these systems, by selecting the appropriate settings for PLATFORM and MPI_HOST.

```
# Makefile for MPI-IO examples

PLATFORM       = AIX
#PLATFORM      = AIX_SMP
#PLATFORM      = SOLARIS

MPI_HOST       = MPICH
#MPI_HOST      = IBM_POE
#MPI_HOST      = SUN_HPC

MPI_MPICH      = /usr/local/mpi
MPI_IBM_POE    = /usr/lpp/ppe.poe
MPI_SUN_HPC    = /opt/SUNWhpc

MPI_PATH       = $(MPI_$(MPI_HOST))

ENCINA_AIX     = /usr/lpp/encina
ENCINA_AIX_SMP = /usr/lpp/encina
ENCINA_SOLARIS = /opt/encina
ENCINA_PATH    = $(ENCINA_$(PLATFORM))
```

```
HPSS          = /opt/hpss

INCLUDES_DCE_on = -I/usr/include/dce -I$(ENCINA_PATH)/include
INCLUDES_DCE_off =
INCLUDES      = -I$(HPSS)/include -I$(MPI_PATH)/include \
                  $(INCLUDES_DCE_$(DCE_SUPPORT))

FLAGS_MPICH   = -D_THREAD_SAFE
FLAGS_IBM_SMP = -D_THREAD_SAFE
FLAGS_SUN_HPC = -mt -D_THREAD_SAFE

FLAGS_AIX     = -D_AIX_PTHREADS_D7 -D_AES_SOURCE \
                  -D_AIX32_THREADS=1 -DAIX_PROD
FLAGS_AIX_SMP = $(FLAGS_AIX)
FLAGS_SOLARIS = -D_XOPEN_SOURCE -D_XOPEN_SOURCE_EXTENDED=1 -D__EXTENSIONS__

FLAGS_DCE_on  =
FLAGS_DCE_off = -DNO_DCE

FLAGS         = $(FLAGS_$(MPI_HOST)) $(FLAGS_$(PLATFORM)) \
                  $(FLAGS_DCE_$(DCE_SUPPORT))

MPCC_MPICH    = $(MPI_PATH)/bin/mpicc
MPCC_IBM_POE  = $(MPI_PATH)/bin/mpcc_r
MPCC_SUN_HPC  = $(MPI_PATH)/bin/tmcc
MPCC          = $(MPCC_$(MPI_HOST))

CFLAGS        = $(INCLUDES) $(FLAGS)

LIB_DIRS_DCE_on   = -L$(HPSS)/lib -L$(ENCINA_PATH)/lib -L$(MPI_PATH)/lib
LIB_DIRS_DCE_off  = -L$(HPSS)/lib -L$(MPI_PATH)/lib
LIB_DIRS          = $(LIB_DIRS_DCE_$(DCE_SUPPORT))

HPSSMPIO_LIBS     = -lmpioapi -lhpss

LIBS_AIX_DCE_on   = -ldcepthreads -lpthreads_compat -lpthreads
LIBS_AIX_DCE_off  = -lpthreads_compat -lpthreads -lm
LIBS_AIX          = $(LIBS_AIX_DCE_$(DCE_SUPPORT))
LIBS_AIX_SMP      = $(LIBS_AIX)

LIBS_SOLARIS_DCE_on= -lnsl -lsocket -z muldefs
LIBS_SOLARIS_DCE_off= -lnsl -lsocket -lm -z muldefs
LIBS_SOLARIS      = $(LIBS_SOLARIS_DCE_$(DCE_SUPPORT))
```

```
HOST_LIBS         = $(LIBS_$(PLATFORM))

DCE_LIBS_on       = -lEncina -lEncClient -ldce
DCE_LIBS_off      =
DCE_LIBS          = $(DCE_LIBS_$(DCE_SUPPORT))

MPI_LIB_MPICH     = -lmpich
MPI_LIB_IBM_POE   = -lmpi_r
MPI_LIB_SUN_HPC   = -lmpi_mt
MPI_LIB           = $(MPI_LIB_$(MPI_HOST))

LIBS          = $(LIB_DIRS) \
                  $(HPSSMPIO_LIBS) \
                  $(HOST_LIBS) \
                  $(DCE_LIBS) \
                  $(MPI_LIB)

.c.o:;@echo Compiling $< ...
   @$(MPCC) $(CFLAGS) -c $<

all: mpio_example

mpio_example:     mpio_example.o
   @echo Loading $@
   @$(MPCC) mpio_example.o $(LIBS) -o $@

mpio_example.o:   mpio_example.c
```

Executing MPI-IO applications

The resulting executable, **mpio_example** in the Makefile above, may be run using the host-appropriate MPI command, which is **mpirun** for MPICH, **poe** for IBM's POE, and **mprun** for SUN's HPC. For example, assuming the application is to be executed on 4 tasks, and that it expects the argument:

argv[1] HPSS file name

the following commands could be used:

For MPICH:

mpirun -np 4 mpio_example /hpsshome/mpiouser/filename

For IBM POE:

```
poe mpio_example /hpsshome/mpiouser/filename -procs 4
```

For Sun HPC:

```
mprun -np 4 mpio_example /hpsshome/mpiouser/filename
```

Please refer to the MPI host-specific man pages and documentation for information on the other command line arguments that may be used with each MPI.

Example 1: Basic file I/O

```
/* Basic MPI-IO example */

#include <mpio.h>
#include <stdio.h>

#define MAX_DATA_SIZE0x00100000/* 2 ** 20 = 1M */

main(int argc, char ** argv)
{
    char * filename;
    char   buffer[MAX_DATA_SIZE];
    int    myrank;
    int    nprocs;
    int    amode;
    MPI_Offsetmyoffset;
    MPI_Filefh;
    MPI_Statusstatus;
    int    rc;
    int    i;

    if (argc < 2) {
        printf("Usage:%s <hpss_filename>\n", argv[0]);
        exit(-1);
    }

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    printf("MPI-IO example to open/write/read/close a file\n");

    filename = argv[1];
```

```
amode = MPI_MODE_CREATE | MPI_MODE_RDWR | MPI_MODE_DELETE_ON_CLOSE;

rc = MPI_File_open(MPI_COMM_WORLD, filename, amode, MPI_INFO_NULL,
           &fh);

if (rc != MPI_SUCCESS) {
    printf("Couldn't open file %s\n", filename);
    MPI_Abort(MPI_COMM_WORLD, rc);
}

memset(buffer, (char)myrank, MAX_DATA_SIZE);

printf("Writing to file\n");

myoffset = myrank * MAX_DATA_SIZE;

rc = MPI_File_write_at(fh, myoffset, buffer, MAX_DATA_SIZE,
           MPI_BYTE, &status);

if (rc != MPI_SUCCESS) {
    printf("Error writing to file: %d\n", rc);
    MPI_Abort(MPI_COMM_WORLD, rc);
}

memset(buffer, -1, MAX_DATA_SIZE);

printf("Reading from file\n");

rc = MPI_File_read_at(fh, myoffset, buffer, MAX_DATA_SIZE,
           MPI_BYTE, &status);

if (rc != MPI_SUCCESS) {
    printf("Error reading from file: %d\n", rc);
    MPI_Abort(MPI_COMM_WORLD, rc);
}

for (i = 0; i < MAX_DATA_SIZE; i++) {
    if ((int)buffer[i] != myrank) {
        printf("Error in data read from file\n");
        break;
    }
}
```

```
    rc = MPI_File_close(&fh);

    if (rc != MPI_SUCCESS) {
        printf("Error closing file: %d\n", rc);
    }

    MPI_Finalize();

    printf("All done\n");
}
```

Example 2: File views and collective I/O

```
/* Using file views and collective I/O in MPI-IO */

#include <mpio.h>
#include <stdio.h>

#define MAX_DATA_SIZE0x00100000/* 2 ** 20 = 1M */

main(int argc, char ** argv)
{
    char * filename;
    int    buffer[MAX_DATA_SIZE];
    int    myrank;
    int    nprocs;
    int    amode;
    MPI_Offsetmyoffset;
    MPI_Filefh;
    MPI_Statusstatus;
    MPI_Datatypeintblock;
    int    rc;
    int    i;

    if (argc < 2) {
        printf("Usage:%s <hpss_filename>\n", argv[0]);
        exit(-1);
    }

    filename = argv[1];

    printf("MPI-IO example using collective I/O for data access.\n");
```

```
MPI_Init(&argc, &argv);

MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

amode = MPI_MODE_CREATE | MPI_MODE_RDWR | MPI_MODE_DELETE_ON_CLOSE;

rc = MPI_File_open(MPI_COMM_WORLD, filename, amode, MPI_INFO_NULL,
           &fh);

if (rc != MPI_SUCCESS) {
    printf("Couldn't open file %s\n", filename);
    MPI_Abort(MPI_COMM_WORLD, rc);
}

/* Set file view to tiling of MPI_INTs */

rc = MPI_File_set_view(fh, (MPI_Offset)0, MPI_INT, MPI_INT,
               "native", MPI_INFO_NULL);

if (rc != MPI_SUCCESS) {
    printf("Couldn't set file view: %d\n", rc);
    MPI_Abort(MPI_COMM_WORLD, rc);
}

for (i = 0; i < MAX_DATA_SIZE; i++)
    buffer[i] = myrank;

printf("Writing to file\n");

myoffset = myrank * MAX_DATA_SIZE * sizeof(int);

rc = MPI_File_write_at_all(fh, myoffset, buffer, MAX_DATA_SIZE,
               MPI_INT, &status);

if (rc != MPI_SUCCESS) {
    printf("Error writing to file: %d\n", rc);
    MPI_Abort(MPI_COMM_WORLD, rc);
}

/* Reset file view to tiling of contiguous block of ints */
```

```
rc = MPI_Type_contiguous(MAX_DATA_SIZE, MPI_INT, &intblock);

if (rc != MPI_SUCCESS) {
    printf("Couldn't construct file type: %d\n", rc);
    MPI_Abort(MPI_COMM_WORLD, rc);
}

rc = MPI_Type_commit(&intblock);

if (rc != MPI_SUCCESS) {
    printf("Couldn't commit file type: %d\n", rc);
    MPI_Abort(MPI_COMM_WORLD, rc);
}

rc = MPI_File_set_view(fh, (MPI_Offset)0, MPI_INT, intblock,
               "native", MPI_INFO_NULL);

if (rc != MPI_SUCCESS) {
    printf("Couldn't set file view: %d\n", rc);
    MPI_Abort(MPI_COMM_WORLD, rc);
}

memset(buffer, -1, MAX_DATA_SIZE * sizeof(int));

/* Note that offset is still in terms of etypes*/

printf("Reading from file\n");

rc = MPI_File_read_at_all(fh, myoffset, buffer, 1, intblock, &status);

if (rc != MPI_SUCCESS) {
    printf("Error reading from file: %d\n", rc);
    MPI_Abort(MPI_COMM_WORLD, rc);
}

for (i = 0; i < MAX_DATA_SIZE; i++) {
    if (buffer[i] != myrank) {
        printf("Error in data read from file\n");
        break;
    }
}

rc = MPI_File_close(&fh);
```

```
    if (rc != MPI_SUCCESS) {
        printf("Error closing file: %d\n", rc);
    }

    MPI_Finalize();

    printf("All done\n");
}
```

Example 3: I/O using file pointers

```
/* Using independent file pointers in MPI-IO */

#include <mpio.h>
#include <stdio.h>

#define MAX_DATA_SIZE0x00100000/* 2 ** 20 = 1M */

main(int argc, char ** argv)
{
    char * filename;
    int    buffer[MAX_DATA_SIZE];
    int    myrank;
    int    nprocs;
    int    amode;
    int    blocksize;
    MPI_Filefh;
    MPI_Datatypeintblock;
    MPI_Statusstatus;
    MPI_Offsetmydisp;
    MPI_Offsetposition;
    MPI_Offsetbyte_offset;
    MPI_Offsetwritten_size;
    MPI_Offsetexpected_size;
    int    rc;
    int    i;

    if (argc < 2) {
        printf("Usage:%s <hpss_filename>\n", argv[0]);
        exit(-1);
    }
```

```
MPI_Init(&argc, &argv);

MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

printf("MPI-IO example using file pointers for data access\n");

filename = argv[1];
amode = MPI_MODE_CREATE | MPI_MODE_RDWR | MPI_MODE_DELETE_ON_CLOSE;

rc = MPI_File_open(MPI_COMM_WORLD, filename, amode, MPI_INFO_NULL,
          &fh);

if (rc != MPI_SUCCESS) {
   printf("Couldn't open file %s\n", filename);
   MPI_Abort(MPI_COMM_WORLD, rc);
}

/*
 *  Set view so that data each task writes will be written at the
 *  right offsets in the file:  two blocks per task, with a stride
 *  between them to skip over data from other tasks.  Use the
 *  file displacement of the view to set the position of each
 *  task's first byte.
 */

blocksize = MAX_DATA_SIZE / 2;/* in etypes */

rc = MPI_Type_vector(2, blocksize, nprocs * blocksize,
          MPI_INT, &intblock);

if (rc != MPI_SUCCESS) {
   printf("Couldn't construct file type\n");
   MPI_Abort(MPI_COMM_WORLD, rc);
}

rc = MPI_Type_commit(&intblock);
if (rc != MPI_SUCCESS) {
   printf("Couldn't commit file type\n");
   MPI_Abort(MPI_COMM_WORLD, rc);
}
```

```
mydisp = myrank * blocksize * sizeof(int);  /* in bytes */

rc = MPI_File_set_view(fh, mydisp, MPI_INT, intblock,
           “native”, MPI_INFO_NULL);

for (i = 0; i < MAX_DATA_SIZE; i++)
    buffer[i] = myrank;

printf(“Writing to file\n”);

rc = MPI_File_write(fh, buffer, MAX_DATA_SIZE,
           MPI_INT, &status);

if (rc != MPI_SUCCESS) {
    printf(“Error writing to file: %d\n”, rc);
    MPI_Abort(MPI_COMM_WORLD, rc);
}

/*
 *  The file position per view should be the same:  each
 *  task thinks it has written MAX_DATA_SIZE ints.
 */

rc = MPI_File_get_position(fh, &position);

if (rc != MPI_SUCCESS) {
    printf(“Error getting file position: %d\n”, rc);
    MPI_Abort(MPI_COMM_WORLD, rc);
}

if (position != MAX_DATA_SIZE) {
    printf(“File position is %lld, but expected MAX_DATA_SIZE\n”,
        position);
}

/* After all tasks have written their data, check file size */

rc = MPI_File_sync(fh);

if (rc != MPI_SUCCESS) {
    printf(“Error syncing file: %d\n”, rc);
    MPI_Abort(MPI_COMM_WORLD, rc);
}
```

```
rc = MPI_File_get_size(fh, &written_size);

expected_size = nprocs * MAX_DATA_SIZE * sizeof(int);

if (written_size != expected_size) {
    printf("Written size is %lld, but expected %lld\n",
        written_size, expected_size);
}

/*
 *  Seek back to the beginning of the file; check position and
 *  byte offset after seek.
 */

MPI_File_seek(fh, 0, MPI_SEEK_SET);

rc = MPI_File_get_position(fh, &position);

if (rc != MPI_SUCCESS) {
    printf("Error getting file position: %d\n", rc);
    MPI_Abort(MPI_COMM_WORLD, rc);
}

if (position != 0) {
    printf("File position is %lld, but expected 0\n", position);
}

/*
 *  Converting position to byte offset should give us the file
 *  displacement used, per task.
 */

rc = MPI_File_get_byte_offset(fh, position, &byte_offset);

if (rc != MPI_SUCCESS) {
    printf("Error converting position to byte_offset: %d\n", rc);
    MPI_Abort(MPI_COMM_WORLD, rc);
}

if (byte_offset != mydisp) {
    printf("Byte offset is %lld, but expected %lld\n",
        byte_offset, mydisp);
```

```
    }

    /* Read the data back, and check it */

    memset(buffer, -1, MAX_DATA_SIZE * sizeof(int));

    printf("Reading from file\n");

    rc = MPI_File_read(fh, buffer, MAX_DATA_SIZE,
               MPI_INT, &status);

    if (rc != MPI_SUCCESS) {
        printf("Error reading from file: %d\n", rc);
        MPI_Abort(MPI_COMM_WORLD, rc);
    }

    for (i = 0; i < MAX_DATA_SIZE; i++) {
        if (buffer[i] != myrank) {
            printf("Error in data read from file\n");
            break;
        }
    }

    rc = MPI_File_close(&fh);

    if (rc != MPI_SUCCESS) {
        printf("Error closing file: %d\n", rc);
    }

    MPI_Finalize();

    printf("All done\n");
}
```

Example 4: Enabling cached I/O

```
/* Using automatic caching in HPSS MPI-IO */

#include <mpio.h>
#include <stdio.h>

#define MAX_DATA_SIZE0x00100000/* 2 ** 20 = 1M */
```

```
main(int argc, char ** argv)
{
    char * filename;
    int    buffer[MAX_DATA_SIZE];
    int *  bufptr;
    int    myrank;
    int    nprocs;
    int    amode;
    MPI_Offsetmyoffset;
    MPI_Offsetoffset;
    MPI_Filefh;
    MPI_Statusstatus;
    MPI_Datatypeintblock;
    int    blen;
    int    disp;
    int    rc;
    int    i;

    if (argc < 2) {
        printf("Usage:%s <hpss_filename>\n", argv[0]);
        exit(-1);
    }

    filename = argv[1];

    printf("MPI-IO example enabling cached I/O for data access.\n");

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    amode = MPI_MODE_CREATE | MPI_MODE_RDWR | MPI_MODE_UNIQUE_OPEN;

    rc = MPI_File_open(MPI_COMM_WORLD, filename, amode, MPI_INFO_NULL,
               &fh);

    if (rc != MPI_SUCCESS) {
        printf("Couldn't open file %s\n", filename);
        MPI_Abort(MPI_COMM_WORLD, rc);
    }
```

```
/* Set file view to tiling of blocks of MPI_INTs */

blen = MAX_DATA_SIZE;
disp = MAX_DATA_SIZE * myrank;
rc = MPI_Type_indexed(1, &blen, &disp, MPI_INT, &intblock);

if (rc != MPI_SUCCESS) {
    printf("Couldn't create file type: %d\n", rc);
    MPI_Abort(MPI_COMM_WORLD, rc);
}

rc = MPI_Type_commit(&intblock);

if (rc != MPI_SUCCESS) {
    printf("Couldn't commit file type: %d\n", rc);
    MPI_Abort(MPI_COMM_WORLD, rc);
}

rc = MPI_File_set_view(fh, (MPI_Offset)0, MPI_INT, intblock,
               "native", MPI_INFO_NULL);

if (rc != MPI_SUCCESS) {
    printf("Couldn't set file view: %d\n", rc);
    MPI_Abort(MPI_COMM_WORLD, rc);
}

for (i = 0; i < MAX_DATA_SIZE; i++)
    buffer[i] = myrank;

myoffset = myrank * MAX_DATA_SIZE * sizeof(int);

/*
 *  For illustration, we do data accesses in smaller amounts
 *  than optimal for the file type.  Small accesses are not
 *  efficient for HPSS, either, so caching will effectively
 *  postpone writing to HPSS until the cache is full or the
 *  file is closed.
 */

printf("Writing to file\n");

offset = myoffset;
bufptr = buffer;
```

```
for (i = 0; i < MAX_DATA_SIZE / 512; i++) {
    rc = MPI_File_write_at_all(fh, offset, bufptr, 512,
                   MPI_INT, &status);

    if (rc != MPI_SUCCESS) {
        printf("Error writing to file: %d\n", rc);
        MPI_Abort(MPI_COMM_WORLD, rc);
    }

    offset += 512;
    bufptr += 512;
}

memset(buffer, -1, MAX_DATA_SIZE * sizeof(int));

printf("Reading from file\n");

offset = myoffset;
bufptr = buffer;
for (i = 0; i < MAX_DATA_SIZE / 256; i++) {
    rc = MPI_File_read_at_all(fh, offset, bufptr, 256,
                   MPI_INT, &status);

    if (rc != MPI_SUCCESS) {
        printf("Error reading from file: %d\n", rc);
        MPI_Abort(MPI_COMM_WORLD, rc);
    }

    offset += 256;
    bufptr += 256;
}

for (i = 0; i < MAX_DATA_SIZE; i++) {
    if (buffer[i] != myrank) {
        printf("Error in data read from file\n");
        break;
    }
}

/* Closing the file will cause cache flush */

printf("Closing the file\n");
```

```
    rc = MPI_File_close(&fh);

    if (rc != MPI_SUCCESS) {
        printf("Error closing file: %d\n", rc);
    }

    MPI_Finalize();

    printf("All done\n");
}
```

Example 5: Nonblocking I/O

```
/* Using nonblocking I/O in MPI-IO */

#include <mpio.h>
#include <stdio.h>

#define MAX_DATA_SIZE0x00100000/* 2 ** 20 = 1M */

int nreps;

void compute_results(
    int *result_buffer,
    int *done
)
{
    static int count = 0;

    /*
     *  Compute results to place in result_buffer.
     *  Return done = 1 when no more results.
     */

    printf("Computing results\n");

    count++;
    if (count >= nreps) *done = 1;
}

main(
    int argc,
```

```
    char ** argv
)
{
    char * filename;
    int    buffer[MAX_DATA_SIZE];
    int    myrank;
    int    nprocs;
    int    amode;
    MPI_Filefh;
    MPI_Statusstatus;
    MPI_Requestrequest;
    int    done;
    int    rc;
    int    i;

    if (argc < 3) {
        printf("Usage:%s <hpss_filename> #reps\n", argv[0]);
        exit(-1);
    }

    filename = argv[1];
    nreps = atol(argv[2]);

    printf("MPI-IO example using nonblocking I/O for data access.\n");

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    amode = MPI_MODE_CREATE | MPI_MODE_RDWR;

    rc = MPI_File_open(MPI_COMM_WORLD, filename, amode, MPI_INFO_NULL,
              &fh);

    if (rc != MPI_SUCCESS) {
        printf("Couldn't open file %s\n", filename);
        MPI_Abort(MPI_COMM_WORLD, rc);
    }

    /* Set file view to tiling of MPI_INTs */

    rc = MPI_File_set_view(fh, (MPI_Offset)0, MPI_INT, MPI_INT,
```

```
                “native”, MPI_INFO_NULL);

if (rc != MPI_SUCCESS) {
    printf(“Couldn’t set file view: %d\n”, rc);
    MPI_Abort(MPI_COMM_WORLD, rc);
}

/*
 *  Common use of nonblocking I/O would be to overlap I/O with
 *  computation.  Imagine a loop that computes, writes results,
 *  and repeats:  some variation on the basic outline below.
 */

done = 0;

compute_results(buffer, &done);

while (!done) {

    printf(“Writing to file\n”);

    rc = MPI_File_iwrite(fh, buffer, MAX_DATA_SIZE,
                MPI_INT, &request);

    if (rc != MPI_SUCCESS) {
        printf(“Error writing to file: %d\n”, rc);
        MPI_Abort(MPI_COMM_WORLD, rc);
    }

    compute_results(buffer, &done);

    MPI_Wait(&request, &status);
}

rc = MPI_File_close(&fh);

if (rc != MPI_SUCCESS) {
    printf(“Error closing file: %d\n”, rc);
}

MPI_Finalize();

printf(“All done\n”);
```

```
}
```

Index

Numerics

64-bit Arithmetic Library
- add64_3m, 21, 230
- add64m, 21, 230
- and64m, 21, 231
- bld64m, 21, 232
- cast64m, 21, 232
- Components, 21
- Constraints, 22
- div_2xcl64m, 21
- div_cl64m, 21, 234, 235
- div2x64m, 21, 233
- div64m, 21, 233
- eq64m, 22, 236
- eqz64m, 22, 236
- ge64m, 22, 237
- gt64m, 22, 237
- high32m, 22, 238
- le64m, 22, 239
- Libraries, 23
- low32m, 22, 239
- lt64m, 22, 240
- mod2x64m, 22, 241
- mod64m, 22, 241
- mul64m, 22, 242
- neq64m, 22
- neqz64m, 22, 243
- not64m, 22, 243
- or64m, 22, 244
- Purpose, 21
- shl64m, 22, 244
- shr64m, 22, 245
- sub64_3m, 22, 246
- sub64m, 22, 246

A

Account Validation Functions
- av_site_AcctNameToIdx, 386
- av_site_Initialize, 389
- av_site_Shutdown, 390

av_site_ValidateAccount, 391
av_site_ValidateChacct, 394
av_site_ValidateChown, 397
av_site_ValidateCreate, 400
av_siteAcctIdxToName, 383
acct_rec_t, 156
ACL Functions
hacl_ConvertACLToHACL, 404
hacl_ConvertHACLPermsToPerms, 406
hacl_ConvertHACLToACL, 404
hacl_ConvertHACLToString, 405
hacl_ConvertHACLTypeToType, 406
hacl_ConvertPermsToHACLPerms, 407
hacl_ConvertStringsToHACL, 408
hacl_ConvertTypeToHACLType, 409
hacl_DeleteHACL, 410
hacl_GetHACL, 411
hacl_SetHACL, 412
hacl_SortHACL, 413
hacl_UpdateHACL, 413
add64_3m, 21, 230
add64m, 21, 229
address_t 173
and64m, 21, 231
API Name Specification 161
api_config_t, 156
api_namespec_t 161
av_site_AcctIdxToName, 383
av_site_AcctNameToIdx 386
av_site_Initialize, 389
av_site_Shutdown, 390
av_site_ValidateAccount, 391
av_site_ValidateChacct, 394
av_site_ValidateChown, 397
av_site_ValidateCreate, 400

B

bf_attrib_md_t, 149
bf_attrib_t, 147
bf_vv_attrib_t, 153
bf_xattrib_t, 148
bfs_callback_addr_t 162
bfs_owner_rec_t, 151
bfs_sc_attrib_t, 152
bfs_stats_t, 155
Bitfile Callback Address 162

Bits
NS Fileset Attribute 161
NS Object Attribute 161
bld64m, 21, 232

C

cast64m, 21, 232
class of service, 25
Client API
components, 13
constraints, 16
environment variables (also see Environment Variables), 16
libraries (also see Libraries), 16
Client API Functions 31
Client API, 27
collective functions, 249
completion_msg_t, 214
COS, 25

D

DCE User Accounts, 28
dce_login, 28
directory operations, 528, 579
displacement, 252
div_2xcl64m, 21, 235
div_cl64m, 21, 234
div2x64m, 21, 233
div64m, 21, 233

E

e64m, 239
end of file, 252
Environment Variables
HPSS_BUSY_DELAY, 17, 25
HPSS_BUSYRETRIES, 17
HPSS_DEBUG, 17
HPSS_DEBUGPATH, 17
HPSS_DESC_NAME, 17
HPSS_DISABLE_CROSS_CELL, 18
HPSS_DMAP_WRITE_UPDATES, 18
HPSS_GKTOTAL_DELAY, 18
HPSS_HOSTNAME, 16
HPSS_KTAB_PATH, 16
HPSS_LIMITED_RETRIES, 18
HPSS_LOGGING_PORT, 21
HPSS_LOGGING_TYPE, 21
HPSS_LS_NAME, 16, 25

HPSS_MAX_CONN, 16
HPSS_MNT_DISABLE_JUNCTIONS, 18
HPSS_NDCG_NAME, 20
HPSS_NDCG_SERVERS, 21
HPSS_NDCG_TCP_PORT, 20
HPSS_NDGC_SERVERS, 25
HPSS_NFS_DISABLE_JUNCTIONS, 18
HPSS_NUMRETRIES, 17
HPSS_PRINCIPAL, 17
HPSS_REGISTRY_SITE_NAME, 18
HPSS_RETRY_STAGE_INP, 17
HPSS_REUSE_CONNECTIONS, 17
HPSS_SERVER_NAME, 17
HPSS_TOTAL_DELAY, 18
HPSS_TRANSFER_TYPE, 17
HPSS_USE_PORT_RANGE, 17
MPIO_DEBUG, 25
MPIO_KEYTAB_PATH, 25
MPIO_LOGIN_NAME, 25
eq64m, 22, 236
eqz64m, 22, 235
etype, 252

F

file family, 26
file handle, 252
file hints, 330
file pointer, 252
filetype, 252

G

Gatekeeper Functions
gk_site_Close, 360
gk_site_Create, 362
gk_site_CreateComplete, 366
gk_site_CreateStats, 366
gk_site_GetMonitorTypes, 369
gk_site_Init, 370
gk_site_Open, 372
gk_site_OpenStats, 373
gk_site_PassThru, 374
gk_site_ReadSitePolicy, 376
gk_site_Shutdown, 378
gk_site_stage, 379
gk_site_StageComplete, 380
gk_site_StageStats, 381
ge64m, 22, 237

get and set file attributes, 531
get and set the client API library configuration, 538
get file system information for a COS, 537
gk_site_Close, 360
gk_site_Create, 362
gk_site_CreateComplete, 366
gk_site_CreateStats, 366
gk_site_GetMonitorTypes, 369
gk_site_Init, 370
gk_site_Open, 372
gk_site_OpenStats, 373
gk_site_PassThru, 374
gk_site_ReadSitePolicy, 376
gk_site_Shutdown, 378
gk_site_StageComplete, 380
gk_site_StageStats, 381
gk_site_State, 379
gt64m, 22, 237

H

hacl_acl_entry_t, 415
hacl_acl_t, 415
hacl_ConvertACLToHACL, 404
hacl_ConvertHACLPermsToPerms, 406
hacl_ConvertHACLToACL, 404
hacl_ConvertHACLToString, 405
hacl_ConvertStringsToHACL, 408
hacl_ConvertTypeToHACLType, 409
hacl_DeleteHACL, 410
hacl_GetHACL, 411
hacl_SetHACL, 412
hacl_SortHACL, 413
hacl_UpdateHACL, 413
HACL-style ACL
    hacl_acl_t, 415
HACL-style ACL Entry
    hacl_acl_entry_t, 415
halc_ConvertHACLTypeToType, 406
halc_ConvertPermsToHACLPerms, 407
high32m, 22, 238
HPSS
    overview, 13
HPSS Directory Entry 163
HPSS Security User Credentials 163
hpss_Access, 15, 32
hpss_AcctCodeToName, 33

hpss_AcctNameToCode, 34
hpss_AsynchStatus, 68
HPSS_BUSY_DELAY, 17, 25
HPSS_BUSY_RETRIES, 17
hpss_Chacct, 15, 35
hpss_ChacctByName, 36
hpss_Chdir, 15, 37
hpss_Chmod, 15, 38, 330
hpss_Chown, 15, 39
hpss_Chroot, 15, 40
hpss_ClientAPIInit, 15
hpss_ClientAPIReset, 15, 41
hpss_Close, 15, 42, 422, 424, 536
hpss_Closedir, 15, 43, 531
hpss_ConvertIdsToNames, 43
hpss_ConvertNamesToIds, 45
hpss_cos_hints_t, 132
hpss_cos_md_t 136
hpss_cos_md_t, 136
hpss_cos_priorities_t, 134
hpss_Create, 15, 46
HPSS_DEBUG, 17
HPSS_DEBUGPATH, 17
hpss_DeleteACL, 15
hpss_DeleteAcl, 47
HPSS_DESC_NAME, 17
hpss_dirent_t 163
HPSS_DISABLE_CROSS_CELL, 18
HPSS_DMAP_WRITE_UPDATES, 18, 20
hpss_Fclear, 15, 49
hpss_FclearOffset, 50
hpss_fileattr_t, 138
hpss_FileGetAttributes, 15, 50, 532, 533
hpss_FileGetXAttributes, 15, 51
hpss_FileSetAttributes, 15, 54, 60, 533
hpss_FilesetCreate, 15, 56
hpss_FilesetDelete, 57
hpss_FilesetDeletes, 15
hpss_FilesetGetAttributes, 15, 58
hpss_FilesetListAll, 15, 59
hpss_FilesetSetAttributes, 15
hpss_Fpreallocate, 15
hpss_Fstat, 15, 64
hpss_Ftruncate, 15, 64
hpss_GetAcct, 15, 65
hpss_GetAcctName, 66

hpss_GetACL, 15, 67
hpss_GetBFSStats, 15, 69
hpss_GetConfiguration, 15, 70, 540
hpss_Getcwd, 15, 71
hpss_GetJunctions, 72
hpss_GetListAttrs, 15, 73, 530
HPSS_GKTOTAL_DELAY, 18
hpss_global_fsent_t 160
HPSS_HOSTNAME, 16
hpss_JunctionCreate, 15, 74
hpss_JunctionDelete, 15, 75
HPSS_KTAB_PATH, 16
HPSS_LIMITED_RETRIES, 18
hpss_Link, 15, 76, 536
hpss_LoadDefaultThreadState, 15, 78
hpss_LoadThreadState, 15, 77
HPSS_LOGGING_PORT, 21
HPSS_LOGGING_TYPE, 21
HPSS_LS_NAME, 16, 20, 25
hpss_Lseek, 15, 79
hpss_LseekOffset, 15
hpss_Lstat, 15, 80, 535
HPSS_MAX_CONN, 16
hpss_Migrate, 15, 81
hpss_Mkdir, 15, 82
HPSS_MNT_DISABLE_JUNCTIONS, 18
HPSS_NDCG_NAME, 20
HPSS_NDCG_SERVERS, 21, 25
HPSS_NDCG_TCP_PORT, 20
HPSS_NFS_DISABLE_JUNCTIONS, 18
HPSS_NUMRETRIES, 17
hpss_Open, 15, 26, 84, 420, 423, 535
hpss_OpenBitfile, 15, 86
hpss_Opendir, 15, 87, 529
HPSS_PRINCIPAL, 17
hpss_Purge, 15, 88
hpss_PurgeLock, 15, 89
hpss_PurgeLoginContext, 15, 90
hpss_PVRetrievals, 20, 131
hpss_Read, 15, 91, 422, 424
hpss_ReadAttrs, 15, 92
hpss_Readdir, 15, 94, 529
hpss_Readlink, 15, 95, 530
hpss_ReadList, 15, 96, 189, 417, 476, 541, 575
    example, 476
HPSS_REGISTRY_SITE_NAME, 18

hpss_Rename, 15, 98
hpss_ReopenBitfile, 15, 99
HPSS_RETRY_STAGE_INP, 17
HPSS_REUSE_CONNECTIONS, 17
hpss_Rewinddir, 15, 100
hpss_Rmdir, 15, 101
HPSS_SERVER_NAME, 17, 20
hpss_SetAcct, 15, 104
hpss_SetAcctByName, 104
hpss_SetACL, 15, 102
hpss_SetBFSStats, 15, 106
hpss_SetConfiguration, 15, 106, 540
hpss_SetCOSByHints, 107
hpss_SetFileOffset, 15, 108
hpss_SetLoginContext, 15, 110
hpss_SiteIdToName, 111
hpss_SiteNameToId, 112
hpss_Stage, 15, 113
hpss_StageCallBack, 114
hpss_Stat, 15, 115, 534
hpss_Statfs, 15, 116, 537
hpss_Statvfs, 117
hpss_Symlink, 15, 118, 536
hpss_ThreadCleanUp, 15, 31, 119
HPSS_TOTAL_DELAY, 18
HPSS_TRANSFER_TYPE, 17, 20
hpss_Truncate, 15, 120
hpss_Umask, 15, 121
hpss_Unlink, 15, 122, 535, 537
hpss_UpdateACL, 15, 123
HPSS_USE_PORT_RANGE, 17
hpss_Utime, 15, 125
hpss_Write, 15, 126, 418, 421, 536
hpss_WriteList ,
    example, 424
hpss_WriteList, 15, 127, 189, 417, 424, 541, 571
hpss_xfileattr_t 139
hpss_XLoadThreadState, 129
hpssuser, 27
hsec_UserCred_t 163

I

I/O Descriptor
    components, 167
    purpose, 167
I/O Reply

components, 168
purpose, 167
individual file pointer, 252
initiator_ipaddr_t, 215
initiator_msg_t, 213
initiator_shmaddr_t, 216
IOD_t, 169
IOR_t, 182

K

kdestroy, 29
kinit, 29
klist, 29

L

le64m, 22
Libraries
libdce.a, 16, 20
libEncina.a, 16, 20
libhpss.a, 16, 20, 24
libmpioapi.a, 24
link operations, 533
low32m, 22
lshpss, 26
lt64m, 22, 240

M

makefile
sample, 417, 541, 563
Math Library Data Definitions
u_signed64, 247
unsigned32, 248
mod2x64m, 22, 241
mod64m, 22, 240
Mover Protocol Data Structures
completion message, 214
Initiator Message, 213
shared memory address, 216
TCP/IP address, 215
Mover Protocol Functions
mvrprot_recv_compmsg, 211
mvrprot_recv_initmsg, 207
mvrprot_recv_ipaddr, 208
mvrprot_recv_shmaddr, 210
mvrprot_send_compmsg, 212
mvrprot_send_initmsg, 208
mvrprot_send_ipaddr, 209

mvrprot_send_shmaddr, 211
Mover Socket Functions
mover_socket_get_buffer, 193
mover_socket_get_buffer_timeout, 194
mover_socket_recv_data, 196
mover_socket_recv_data_timeout, 197
mover_socket_send_buffer, 190
mover_socket_send_buffer_timeout, 191
mover_socket_send_buffer_timeout_size, 192
mover_socket_send_requested_data, 198
mover_socket_send_requested_data_timeout, 199
mover_socket_send_requested_data_timeout_size, 201
mover_socket_waitfor_data, 202
mover_socket_waitfor_data_timeout, 203
mover_waitfor_requests, 204
mover_waitfor_requests_timeout, 206
mover to mover protocol, 424, 476
mover_socket_get_buffer, 193
mover_socket_get_buffer_timeout, 194
mover_socket_recv_data, 195
mover_socket_recv_data_timeout, 197
mover_socket_send_buffer, 190
mover_socket_send_buffer_timeout, 191
mover_socket_send_buffer_timeout_size, 192
mover_socket_send_requested_data, 198
mover_socket_send_requested_data_timeout, 199
mover_socket_send_requested_data_timeout_size, 200
mover_socket_waitfor_data, 202
mover_socket_waitfor_data_timeout, 203
mover_waitfor_requests, 204
mover_waitfor_requests_timeout, 205
MPI_File_call_errhandler, 24, 329
MPI_File_close, 23, 251, 252, 255
MPI_File_create_errhandler, 24, 323, 327
MPI_File_delete, 23, 256, 323, 331
MPI_File_get_amode, 23, 261
MPI_File_get_atomicity, 24, 321
MPI_File_get_byte_offset, 24, 288
MPI_File_get_errhandler, 24, 328
MPI_File_get_group, 23, 260
MPI_File_get_info, 23, 262, 330, 331
MPI_File_get_nthkey, 336
MPI_File_get_position, 24, 287
MPI_File_get_position_shared, 24, 299
MPI_File_get_size, 23, 259
MPI_File_get_type_extent, 24, 316, 318

MPI_File_get_valuelen, 335
MPI_File_get_view, 23, 265
MPI_File_iread, 24
MPI_File_iread_at, 23
MPI_File_iread_shared, 24, 292
MPI_File_iwrite, 24
MPI_File_iwrite_at, 23
MPI_File_iwrite_shared, 24, 293
MPI_File_open, 23, 249, 250, 251, 252, 253, 323, 331
MPI_File_preallocate, 23, 251, 258
MPI_File_read, 24
MPI_File_read_all, 24
MPI_File_read_all_begin, 24, 305
MPI_File_read_all_end, 24, 306
MPI_File_read_at, 23
MPI_File_read_at_all_begin, 23, 300
MPI_File_read_at_all_end, 23, 302
MPI_File_read_ordered, 24, 295
MPI_File_read_ordered_begin, 24, 310
MPI_File_read_ordered_end, 24, 311
MPI_File_read_share, 289
MPI_File_read_shared, 24
MPI_File_seek, 24, 286
MPI_File_seek_shared, 24, 251, 298
MPI_File_set_atomicity, 24, 251, 321
MPI_File_set_errhandler, 24, 323, 327
MPI_File_set_info, 23, 261, 331
MPI_File_set_size, 23, 251, 257
MPI_File_set_view, 23, 249, 251, 263, 315, 324, 331
MPI_File_sync 322
MPI_File_sync, 24, 251
MPI_File_write, 24
MPI_File_write_all, 24
MPI_File_write_all_begin, 24, 307
MPI_File_Write_all_end, 24
MPI_File_write_all_end, 309
MPI_File_write_at, 23
MPI_File_write_at_all, 23
MPI_File_write_at_all_begin, 23, 303
MPI_File_write_at_all_end, 23, 304
MPI_File_write_ordered, 24, 296
MPI_File_write_ordered_begin, 24, 312
MPI_File_write_ordered_end, 24, 314
MPI_File_write_shared, 24, 290
MPI_Info, 331
MPI_Info_create, 24, 331, 332

MPI_Info_delete, 24, 333
MPI_Info_dup, 24, 337
MPI_Info_free, 24, 331, 338
MPI_Info_get, 24, 331, 334
MPI_Info_get_nkeys, 24, 336
MPI_Info_get_nthkey, 24
MPI_Info_get_valuelen, 24
MPI_Info_set, 24, 331, 332
MPI_Offset 352
MPI_Offset, 352
MPI_Register_datarep, 24, 318
MPI_Test, 250, 326
MPI_Wait, 250, 326
MPI-IO access, 27
MPI-IO API
    collective functions, 249
    Components, 23
    Constraints, 24
    Environment Variables, 25
    Libraries, 24
    noncollective functions, 249
    Purpose, 23
MPI-IO file, 252
MPI-IO Standard Data Definitions
    MPI_Offset, 352
MPIO_DEBUG, 25
MPIO_KEYTAB_PATH, 25
MPIO_LOGIN_NAME, 25
mul64m, 22, 242
mvrprot_recv_compmsg, 211
mvrprot_recv_initmsg, 207
mvrprot_recv_ipaddr, 208
mvrprot_recv_shmaddr, 210
mvrprot_send_compmsg, 212
mvrprot_send_initmsg, 208
mvrprot_send_ipaddr, 209
mvrprot_send_shmaddr, 210

N

neq64m, 22
neqz64m, 22, 242
netopt_entry_t, 226
netopt_FindEntry, 225
netopt_GetWriteSize, 225
Network Options Data Definitions
    entry, 226

Network Options Functions
    netopt_FindEntry, 225
    netopt_GetWriteSize, 226
noncollective functions, 249
Non-DCE Client API
    components, 20
    constraints, 20
    environment variables (also see Environment Variables), 20
    libraries (also see Libraries), 20
not64m, 22, 243
ns_ACLConfArray_t, 159
ns_ACLEntry_t, 159
ns_Attrs_t, 139
ns_DirEntry_t, 147
ns_FilesetAttrBits_t 161
ns_FilesetAttrs_t, 144
ns_ObjHandle_t, 145

O

offset, 252
or64m, 22, 244
ow32m, 239

P

Parallel Data Transfer Data Definitions
    header, 223
Parallel Data Transfer Functions
    pdata_recv_hdr, 217
    pdata_recv_hdr_timeout, 218
    pdata_send_hdr, 218
    pdata_send_hdr_and_data, 220
    pdata_send_hdr_and_data_timeout, 221
    pdata_send_hdr_and_data_timeout_size, 222
    pdata_send_hdr_timeout, 219
participating process, 249
pdata_hdr_t, 223
pdata_recv_hdr, 217
pdata_recv_hdr_timeout, 217
pdata_send_hdr, 218
pdata_send_hdr_and_data, 220
pdata_send_hdr_and_data_timeout, 221
pdata_send_hdr_and_data_timeout_size, 222
pdata_send_hdr_timeout, 219
Purge Lock Flag 161
purgelock_flag_t 161
pv_list_element_t, 154
pv_list_t, 154

S

SecPWent_t 164
Security Password Entry 164
shared file pointer, 252
shl64m, 22, 244
shr64m, 22, 245
size, 252
srcsinkdesc_t 172
srcsinkreply_t, 186
storage class, 26
storage hierarchy, 26
Structures
    account record, 156
    address, 173
    API configuration, 156
    bitfile metadata attributes, 149
    bitfile owner record, 151
    bitfile server statistics 155
    bitfile server storage class attributes 152
    bitfile server virtual volume attributes 153
    bitfile volatile and extended metadata attributes, 148
    bitfile volatile and metadata attributes, 147
    COS metadata, 136
    COS priorities, 134
    Extended File Attribute 139
    file attribute, 138
    file creation hint, 132
    Fileset attributes, 144
    Global Fileset Entry 160
    I/O descriptor, 169
    I/O reply, 182
    NS ACL conformant array, 159
    NS ACL entry, 159
    NS attribute, 139
    NS directory entry, 147
    NS object handle, 145
    source/sink descriptor, 172
    source/sink reply, 186
    storage server physical volume attributes 154
    storage server physical volume attributes conformant array 154
sub64_3m, 22, 246
sub64m, 22, 246

U

u_signed64, 247
unsigned32, 248

User IDs, 27

V

view, 252