

---

# *HPSS*

## *Programmer's Reference Guide, Volume 1*

**High Performance Storage System  
Release 4.1.1**

*November 1999 (Revision 2)*

## HPSS Programmer's Reference, Volume 1

© 1992-1999 International Business Machines Corporation, The Regents of the University of California, Sandia Corporation, Lockheed Martin Energy Research Corporation, and NASA Langley 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 4.1.1

November 1999 (Revision 2)

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

Preface .....	ix
<b>Chapter 1. Overview.....</b>	<b>1-1</b>
1.1. Client API.....	1-1
1.1.1. Purpose .....	1-1
1.1.2. Components .....	1-1
1.1.3. Constraints.....	1-3
1.1.4. Libraries .....	1-3
1.1.5. Environment Variables.....	1-3
1.2. Supplemental Data Transfer Functions .....	1-5
1.2.1. Purpose .....	1-5
1.2.2. Components .....	1-6
1.2.2.1. IPI-3 Data Transfer .....	1-6
1.2.2.2. Mover Socket (Parallel TCP/IP Data Transfer).....	1-6
1.2.2.3. Mover Protocol.....	1-6
1.2.2.4. Parallel Data Transfer .....	1-6
1.2.2.5. Network Options .....	1-6
1.2.3. Constraints.....	1-6
1.2.4. Libraries .....	1-7
1.3. Non-DCE Client API.....	1-7
1.3.1. Purpose .....	1-7
1.3.2. Components .....	1-7
1.3.3. Constraints.....	1-7
1.3.4. Libraries .....	1-8
1.3.5. Environment Variables.....	1-8
1.4. 64-bit Arithmetic Library .....	1-9
1.4.1. Purpose .....	1-9
1.4.2. Components .....	1-9
1.4.3. Constraints.....	1-10
1.4.4. Libraries .....	1-10
1.5. MPI-IO API .....	1-10
1.5.1. Purpose .....	1-11
1.5.2. Components .....	1-11
1.5.3. Constraints.....	1-12
1.5.4. Libraries .....	1-12
1.5.5. Environment Variables.....	1-13
1.6. Storage Concepts .....	1-14
1.6.1. Class of Service .....	1-14
1.6.2. Storage Class .....	1-15

# Programmer's Reference Guide Vol. 1

1.6.3. Storage Hierarchy.....	1-15
1.6.4. File Family .....	1-15
1.7. User IDs .....	1-16
1.8. DCE User Accounts.....	1-16
<b>Chapter 2. Client API Functions.....</b>	<b>2-1</b>
2.1. API Interfaces.....	2-1
2.1.1. hpss_Access.....	2-3
2.1.2. hpss_Chacct.....	2-5
2.1.3. hpss_Chdir.....	2-6
2.1.4. hpss_Chmod.....	2-7
2.1.5. hpss_Chown.....	2-9
2.1.6. hpss_Chroot.....	2-11
2.1.7. hpss_ClientAPIReset.....	2-13
2.1.8. hpss_Close .....	2-14
2.1.9. hpss_Closedir .....	2-15
2.1.10. hpss_Create .....	2-16
2.1.11. hpss_DeleteACL .....	2-18
2.1.12. hpss_Fclear .....	2-20
2.1.13. hpss_FclearOffset.....	2-21
2.1.14. hpss_FileGetAttributes .....	2-22
2.1.15. hpss_FileGetXAttributes.....	2-23
2.1.16. hpss_FileSetAttributes .....	2-25
2.1.17. hpss_FilesetCreate .....	2-28
2.1.18. hpss_FilesetDelete .....	2-30
2.1.19. hpss_FilesetGetAttributes .....	2-31
2.1.20. hpss_FilesetListAll.....	2-33
2.1.21. hpss_FilesetSetAttributes .....	2-35
2.1.22. hpss_Fstat .....	2-38
2.1.23. hpss_Ftruncate .....	2-39
2.1.24. hpss_GetAcct.....	2-40
2.1.25. hpss_GetACL .....	2-41
2.1.26. hpss_GetBFSSStats .....	2-43
2.1.27. hpss_GetConfiguration.....	2-44
2.1.28. hpss_Getcwd .....	2-45
2.1.29. hpss_GetListAttrs .....	2-46
2.1.30. hpss_JunctionCreate.....	2-48
2.1.31. hpss_JunctionDelete.....	2-50
2.1.32. hpss_Link.....	2-51
2.1.33. hpss_LoadThreadState.....	2-53
2.1.34. hpss_LoadDefaultThreadState .....	2-54
2.1.35. hpss_Lseek.....	2-55
2.1.36. hpss_Lstat .....	2-57
2.1.37. hpss_Migrate .....	2-59

## Programmer's Reference Guide Volume 1

2.1.38. hpss_Mkdir.....	2-61
2.1.39. hpss_Open.....	2-63
2.1.40. hpss_OpenBitfile.....	2-66
2.1.41. hpss_Opendir.....	2-68
2.1.42. hpss_Purge.....	2-70
2.1.43. hpss_PurgeLock.....	2-72
2.1.44. hpss_PurgeLoginContext.....	2-73
2.1.45. hpss_Read.....	2-74
2.1.46. hpss_ReadAttrs.....	2-76
2.1.47. hpss_Readdir.....	2-78
2.1.48. hpss_Readlink.....	2-80
2.1.49. hpss_ReadList.....	2-82
2.1.50. hpss_Rename.....	2-85
2.1.51. hpss_ReopenBitfile.....	2-87
2.1.52. hpss_Rewinddir.....	2-89
2.1.53. hpss_Rmdir.....	2-90
2.1.54. hpss_SetACL.....	2-92
2.1.55. hpss_SetAcct.....	2-94
2.1.56. hpss_SetBFSStats.....	2-95
2.1.57. hpss_SetConfiguration.....	2-96
2.1.58. hpss_SetCOSByHints.....	2-97
2.1.59. hpss_SetFileOffset.....	2-99
2.1.60. hpss_SetLoginContext.....	2-101
2.1.61. hpss_Stage.....	2-102
2.1.62. hpss_StageCallBack.....	2-104
2.1.63. hpss_Stat.....	2-106
2.1.64. hpss_Statfs.....	2-108
2.1.65. hpss_Symlink.....	2-109
2.1.66. hpss_ThreadCleanUp.....	2-111
2.1.67. hpss_Truncate.....	2-112
2.1.68. hpss_Umask.....	2-114
2.1.69. hpss_Unlink.....	2-115
2.1.70. hpss_UpdateACL.....	2-117
2.1.71. hpss_Utime.....	2-119
2.1.72. hpss_Write.....	2-121
2.1.73. hpss_WriteList.....	2-123
2.1.74. hpss_XLoadThreadState.....	2-125
2.1.75. free_ior_mem.....	2-127
2.2. Non-DCE Client API Specific Interfaces.....	2-128
2.2.1. hpss_PVRetrievals.....	2-129
2.3. Data Definitions.....	2-130
2.3.1. File Creation Hint Structure - hpss_cos_hints_t.....	2-130
2.3.2. Class of Service Priorities - hpss_cos_priorities_t.....	2-132
2.3.3. Class of Service Metadata Structure - hpss_cos_md_t.....	2-133
2.3.4. File Attribute Structure - hpss_fileattr_t.....	2-136

## Programmer's Reference Guide Vol. 1

2.3.5. Extended File Attribute Structure - <code>hpss_xfileattr_t</code> .....	2-136
2.3.6. Name Server Attribute Structure - <code>ns_Attrs_t</code> .....	2-137
2.3.7. Name Server Fileset Attributes Structure - <code>ns_FilesetAttrs_t</code> .....	2-142
2.3.8. Name Server Object Handle Structure - <code>ns_ObjHandle_t</code> .....	2-144
2.3.9. Name Server Directory Entry - <code>ns_DirEntry_t</code> .....	2-145
2.3.10. Bitfile Volatile and Metadata Attributes - <code>bf_attrib_t</code> .....	2-146
2.3.11. Bitfile Volatile and Metadata Extended Attributes - <code>bf_xattrib_t</code> .....	2-146
2.3.12. Bitfile Metadata Attributes - <code>bf_attrib_md_t</code> .....	2-147
2.3.13. Bitfile Owner Record - <code>bfs_owner_rec_t</code> .....	2-150
2.3.14. Bitfile Server Storage Class Attributes - <code>bf_sc_attrib_t</code> .....	2-151
2.3.15. Bitfile Server Virtual Volume Attributes - <code>bf_vv_attrib_t</code> .....	2-152
2.3.16. Storage Server Physical Volume Attributes - <code>pv_list_element_t</code> .....	2-153
2.3.17. Storage Server Physical Volume Attributes Conformant Array - <code>pv_list_t</code> .....	2-153
2.3.18. Bitfile Server Statistics - <code>bfs_stats_t</code> .....	2-154
2.3.19. Account Record - <code>acct_rec_t</code> .....	2-154
2.3.20. API Configuration Structure - <code>api_config_t</code> .....	2-155
2.3.21. Name Server ACL Conformant Array - <code>ns_ACLConfArray_t</code> .....	2-158
2.3.22. Name Server Access Control List Entry - <code>ns_ACLEntry_t</code> .....	2-158
2.3.23. Global Fileset Entry Structure - <code>hpss_global_fsent_t</code> .....	2-159
2.3.24. Name Server Fileset Attribute Bits - <code>ns_FilesetAttrBits_t</code> .....	2-160
2.3.25. Name Server Object Attribute Bits - <code>ns_FilesetAttrBits_t</code> .....	2-160
2.3.26. Purge Lock Flag - <code>purgelock_flag_t</code> .....	2-160
<b>Chapter 3. I/O Descriptor (IOD) and I/O Reply (IOR) .....</b>	<b>3-1</b>
3.1. I/O Descriptor Purpose.....	3-1
3.2. I/O Reply Purpose .....	3-1
3.3. I/O Descriptor Components.....	3-1
3.4. I/O Reply Components.....	3-2
3.5. Data Definitions.....	3-3
3.5.1. I/O Descriptor (IOD) - <code>IOD_t</code> .....	3-3
3.5.2. Source/Sink Descriptor - <code>srcsinkdesc_t</code> .....	3-6
3.5.3. Address Structure - <code>address_t</code> .....	3-8
3.5.4. I/O Reply (IOR) - <code>IOR_t</code> .....	3-17
3.5.5. Source/Sink Reply - <code>srcsinkreply_t</code> .....	3-21
<b>Chapter 4. Supplemental Data Transfer Functions.....</b>	<b>4-1</b>
4.1. API Functions.....	4-1
4.1.1. IPI-3 Data Transfer Library Functions.....	4-2
4.1.1.1. <code>ipi3_data3_open</code> .....	4-2
4.1.1.2. <code>ipi3_data3_close</code> .....	4-3
4.1.1.3. <code>ipi3_data3_read</code> .....	4-4
4.1.1.4. <code>ipi3_data3_write</code> .....	4-6

4.1.1.5. ipi3_data3_complete .....	4-8
4.1.1.6. ipi3_data3_cancel.....	4-9
4.1.2. IPI-3 Data Transfer Library Data Definitions.....	4-10
4.1.2.1. IPI-3 Interface Address Structure - IPI3_INTERFACE_STRUCT.....	4-10
4.1.3. Mover Socket (Parallel TCP/IP Data Transfer) Functions.....	4-11
4.1.3.1. mover_socket_send_buffer .....	4-11
4.1.3.2. mover_socket_send_buffer_timeout.....	4-13
4.1.3.3. mover_socket_send_buffer_timeout_size.....	4-15
4.1.3.4. mover_socket_get_buffer .....	4-17
4.1.3.5. mover_socket_get_buffer_timeout.....	4-19
4.1.3.6. mover_socket_recv_data .....	4-21
4.1.3.7. mover_socket_recv_data_timeout .....	4-23
4.1.3.8. mover_socket_send_requested_data.....	4-25
4.1.3.9. mover_socket_send_requested_data_timeout.....	4-27
4.1.3.10. mover_socket_send_requested_data_timeout_size .....	4-29
4.1.3.11. mover_waitfor_data.....	4-31
4.1.3.12. mover_waitfor_data_timeout .....	4-33
4.1.3.13. mover_waitfor_requests.....	4-35
4.1.3.14. mover_waitfor_requests_timeout .....	4-37
4.1.4. Mover Protocol APIs.....	4-39
4.1.4.1. mvrprot_recv_initmsg .....	4-39
4.1.4.2. mvrprot_send_initmsg.....	4-40
4.1.4.3. mvrprot_recv_ipaddr.....	4-41
4.1.4.4. mvrprot_send_ipaddr.....	4-42
4.1.4.5. mvrprot_recv_ipi3addr.....	4-43
4.1.4.6. mvrprot_send_ipi3addr.....	4-44
4.1.4.7. mvrprot_recv_shmaddr.....	4-45
4.1.4.8. mvrprot_send_shmaddr.....	4-46
4.1.4.9. mvrprot_recv_compmsg .....	4-47
4.1.4.10. mvrprot_send_compmsg.....	4-48
4.1.5. Mover Protocol Data Structures.....	4-49
4.1.5.1. Mover Protocol Initiator Message Structure - initiator_msg_t .....	4-49
4.1.5.2. Mover Protocol Completion Msg Structure - completion_msg_t .....	4-51
4.1.5.3. Mover Protocol TCP/IP Address Structure - initiator_ipaddr_t.....	4-52
4.1.5.4. Mover Protocol IPI-3 Address Structure - initiator_ipi3addr_t.....	4-53
4.1.5.5. Mover Protocol Shm Address Structure - initiator_shmaddr_t.....	4-54
4.1.6. Parallel Data Transfer Functions.....	4-55
4.1.6.1. pdata_recv_hdr .....	4-55
4.1.6.2. pdata_recv_hdr_timeout .....	4-56

## Programmer's Reference Guide Vol. 1

4.1.6.3. pdata_send_hdr .....	4-57
4.1.6.4. pdata_send_hdr_timeout.....	4-58
4.1.6.5. pdata_send_hdr_and_data.....	4-59
4.1.6.6. pdata_send_hdr_and_data_timeout.....	4-60
4.1.6.7. pdata_send_hdr_and_data_timeout_size .....	4-62
4.1.7. Parallel Data Transfer Data Definitions .....	4-64
4.1.7.1. Parallel Data Transfer Header - pdata_hdr_t .....	4-64
4.1.8. Network Options Functions.....	4-65
4.1.8.1. netopt_FindEntry .....	4-65
4.1.8.2. netopt_GetWriteSize .....	4-66
4.1.9. Network Options Data Definitions.....	4-67
4.1.9.1. Network Options Entry - netopt_entry_t.....	4-67
<b>Chapter 5. Math Library .....</b>	<b>5-1</b>
5.1. API Interfaces.....	5-1
5.1.1. add64m.....	5-2
5.1.2. add64_3m.....	5-3
5.1.3. and64m.....	5-4
5.1.4. bld64m.....	5-5
5.1.5. cast64m.....	5-6
5.1.6. div64m.....	5-7
5.1.7. div2x64m.....	5-8
5.1.8. div_cl64m.....	5-9
5.1.9. div_2xcl64m.....	5-10
5.1.10. eqz64m .....	5-11
5.1.11. eq64m .....	5-12
5.1.12. ge64m .....	5-13
5.1.13. gt64m.....	5-14
5.1.14. high32m .....	5-15
5.1.15. le64m .....	5-16
5.1.16. low32m.....	5-17
5.1.17. lt64m.....	5-18
5.1.18. mod64m .....	5-19
5.1.19. mod2x64m .....	5-20
5.1.20. mul64m .....	5-21
5.1.21. neqz64m.....	5-22
5.1.22. not64m.....	5-23
5.1.23. or64m.....	5-24
5.1.24. shl64m .....	5-25
5.1.25. shr64m.....	5-26
5.1.26. sub64m .....	5-27
5.1.27. sub64_3m .....	5-28

5.2. Data Definitions.....	5-29
5.2.1. u_signed64.....	5-29
5.2.2. unsigned32.....	5-29
<b>Chapter 6. MPI-IO API Functions .....</b>	<b>6-1</b>
6.1. Application Programming Interfaces.....	6-1
6.1.1. File Manipulation .....	6-3
6.1.1.1. MPI_File_open.....	6-5
6.1.1.2. MPI_File_close.....	6-8
6.1.1.3. MPI_File_delete.....	6-9
6.1.1.4. MPI_File_set_size.....	6-11
6.1.1.5. MPI_File_preallocate.....	6-13
6.1.1.6. MPI_File_get_size.....	6-15
6.1.1.7. MPI_File_get_group.....	6-16
6.1.1.8. MPI_File_get_amode.....	6-17
6.1.1.9. MPI_File_set_info.....	6-18
6.1.1.10. MPI_File_get_info.....	6-19
6.1.1.11. MPI_File_set_view.....	6-20
6.1.1.12. MPI_File_get_view.....	6-22
6.1.2. File Access.....	6-24
6.1.2.1. MPI_File_read_at.....	6-26
6.1.2.2. MPI_File_read_at_all.....	6-28
6.1.2.3. MPI_File_write_at.....	6-30
6.1.2.4. MPI_File_write_at_all.....	6-32
6.1.2.5. MPI_File_iread_at.....	6-34
6.1.2.6. MPI_File_iwrite_at.....	6-36
6.1.2.7. MPI_File_read.....	6-38
6.1.2.8. MPI_File_read_all.....	6-40
6.1.2.9. MPI_File_write.....	6-42
6.1.2.10. MPI_File_write_all.....	6-44
6.1.2.11. MPI_File_iread.....	6-46
6.1.2.12. MPI_File_iwrite.....	6-48
6.1.2.13. MPI_File_seek.....	6-50
6.1.2.14. MPI_File_get_position.....	6-52
6.1.2.15. MPI_File_get_byte_offset.....	6-53
6.1.2.16. MPI_File_read_shared.....	6-54
6.1.2.17. MPI_File_write_shared.....	6-56
6.1.2.18. MPI_File_iread_shared.....	6-58
6.1.2.19. MPI_File_iwrite_shared.....	6-60

## Programmer's Reference Guide Vol. 1

6.1.2.20. MPI_File_read_ordered .....	6-62
6.1.2.21. MPI_File_write_ordered.....	6-64
6.1.2.22. MPI_File_seek_shared.....	6-66
6.1.2.23. MPI_File_get_position_shared .....	6-68
6.1.2.24. MPI_File_read_at_all_begin.....	6-69
6.1.2.25. MPI_File_read_at_all_end.....	6-71
6.1.2.26. MPI_File_write_at_all_begin .....	6-72
6.1.2.27. MPI_File_write_at_all_end.....	6-74
6.1.2.28. MPI_File_read_all_begin.....	6-76
6.1.2.29. MPI_File_read_all_end .....	6-78
6.1.2.30. MPI_File_write_all_begin.....	6-79
6.1.2.31. MPI_File_write_all_end.....	6-81
6.1.2.32. MPI_File_read_ordered_begin .....	6-83
6.1.2.33. MPI_File_read_ordered_end.....	6-85
6.1.2.34. MPI_File_write_ordered_begin.....	6-86
6.1.2.35. MPI_File_write_ordered_end .....	6-88
6.1.3. File Interoperability.....	6-90
6.1.3.1. MPI_File_get_type_extent .....	6-93
6.1.3.2. MPI_Register_datarep.....	6-94
6.1.4. File Consistency.....	6-96
6.1.4.1. MPI_File_set_atomicsity.....	6-97
6.1.4.2. MPI_File_get_atomicsity .....	6-98
6.1.4.3. MPI_File_sync .....	6-99
6.1.5. Error Handling.....	6-100
6.1.5.1. MPI_File_create_errhandler.....	6-104
6.1.5.2. MPI_File_set_errhandler .....	6-105
6.1.5.3. MPI_File_get_errhandler.....	6-106
6.1.5.4. MPI_File_call_errhandler .....	6-107
6.1.6. File Hints.....	6-108
6.1.6.1. MPI_Info_create .....	6-110
6.1.6.2. MPI_Info_set.....	6-111
6.1.6.3. MPI_Info_delete.....	6-112
6.1.6.4. MPI_Info_get.....	6-113
6.1.6.5. MPI_Info_get_valuelen.....	6-114
6.1.6.6. MPI_Info_get_nkeys.....	6-115
6.1.6.7. MPI_Info_get_nthkey.....	6-116
6.1.6.8. MPI_Info_dup .....	6-117
6.1.6.9. MPI_Info_free.....	6-118
6.2. Data Definitions.....	6-119
6.2.1. MPI-IO Standard Data Definitions .....	6-119

## Programmer's Reference Guide Volume 1

6.2.2. MPI Offset.....	6-119
<b>Appendix A - Programming Examples.....</b>	<b>A-1</b>
<b>Appendix B - Makefile Example.....</b>	<b>B-1</b>
<b>Appendix C - Notes.....</b>	<b>C-1</b>
<b>Appendix D - Acronyms.....</b>	<b>D-1</b>
<b>Appendix E - References.....</b>	<b>3</b>

# Programmer's Reference Guide Vol. 1

## Preface

This *High Performance Storage System (HPSS) Programmer's Reference Guide, Volume 1, Release 4.1.1*, documents client function calls which are provided by HPSS. It is designed for application programmers.

The document provides the programming reference for HPSS Release 4.1.1. In particular, the HPSS Client Application Programming Interfaces (APIs), 64-bit arithmetic library calls, and Message Passing Interface - Input / Output (MPI-IO) 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 Administration Guide* for a description 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 (NFS V2), or DFS, 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, constraints, and required libraries.

<i>Chapter 2: Client API</i>	This chapter defines the Client API specifications and associated data definitions.
<i>Chapter 3: I/O Descriptor (IOD) and I/O Reply (IOR)</i>	This chapter describes the I/O Descriptor (IOD) and I/O Reply (IOR) Structures.
<i>Chapter 4: Supplemental Data Transfer Functions</i>	This chapter describes a set of support APIs to facilitate data transfers. Applications using <code>hpss_ReadList</code> and <code>hpss_WriteList</code> are potential users of these functions.
<i>Chapter 5: Math Library</i>	This chapter defines the 64-bit arithmetic library API specifications and associated data definitions.
<i>Chapter 6: MPI-IO</i>	This chapter defines the MPI-IO specifications and associated data definitions.
<i>Appendix A: Programming Examples</i>	This appendix provides some example code for reading and writing HPSS files.
<i>Appendix B: Makefile Examples</i>	This appendix provides example Makefiles for building client applications on different platforms.
<i>Appendix B: Notes</i>	This appendix provides notes on IOD usage.
<i>Appendix C: Acronyms</i>	This appendix provides a list of acronyms used in this document.
<i>Appendix D: References</i>	This appendix lists documents cited in the text as well as other reference materials.

## Typographic and Keying Conventions

This document uses the following typographic conventions:

<b>Bold</b>	<b>Bold</b> words or characters represent system elements that you must use literally, such as functions, commands or keywords.
<i>Italic</i>	<i>Italic</i> 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

## Chapter 1. Overview

---

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\_LseekOffset**, **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\_Chacct**, **hpss\_Chmod**, **hpss\_Chown**, **hpss\_DeleteACL**, **hpss\_Fclear**, **hpss\_FileGetAttributes**, **hpss\_FileGetXAttributes**, **hpss\_FileSetAttributes**, **hpss\_Fpreallocate**, **hpss\_Fstat**, **hpss\_Ftruncate**, **hpss\_GetAcct**, **hpss\_GetACL**, **hpss\_GetListAttrs**, **hpss\_Lstat**, **hpss\_Migrate**, **hpss\_Purge**, **hpss\_SetACL**, **hpss\_SetAcct**, **hpss\_Stage**, **hpss\_Stat**, **hpss\_Statfs**, **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 are two important internal routines: **hpss\_ClientAPIInit**, and **hpss\_ClientAPIReset**. These APIs 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_GetBFSSStats` and `hpss_SetBFSSStats`.

### 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

Two Client API libraries are provided. Applications issuing HPSS Client API calls must link one of the following libraries:

<b>libhpss.a</b>	HPSS client library
<b>libhpss_ipi.a</b>	Same as <code>libhpss.a</code> , but also provides HPSS IPI-3 support. The HPSS IPI-3 support library <code>libhpssipi3.a</code> must also be linked. In addition, it is required to include the IPI-3 library by specifying <code>-lipi3</code> when compiling / linking the application. Also, for IPI-3 transfers, the <code>HPSS_TRANSFER_TYPE</code> environment variable must be set to <code>IPI3</code> . Refer to section 6.6.

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. `/usr/lpp/hpss/config/hpss_env`) for the environment variable settings.

One environment variable is used to control the HPSS system that the Client API attempts to contact. If the default values are not used for the Location Server name, the following environment variable can be

## Chapter 1. Overview

---

used to specify non-default name selected for the server. Note: you will need to check with your HPSS administrator to determine the server name if defaults are not being used.

**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.

Another environment variable is used to control the number of connections that are supported by the Client API within a single client process.

**HPSS\_MAX\_CONN** identifies the integer value that will be used as the maximum number of allowed connections. The default value of zero indicates using the default supported by the HPSS connection management software, which is currently 150.

One other environment variable is used to control the name of the file containing the DCE security keys necessary for successfully initializing the Client API.

**HPSS\_KTAB\_PATH** names the file (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. Valid values are either "TCP" for TCP/IP transfers or "IPI3" for IPI-3 transfers over HIPPI. The default value is "TCP". Note that the Client API library (`libhpss.a` or `libhpss_ipi.a`) must be linked by the application for the transfer to be performed via IPI-3 over HIPPI.

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** environment variable is used to place a descriptive name in any HPSS mes-

sages 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 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 DMAP file system while writing to a file that is mirror in HPSS. The default value is 20.

## **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. IPI-3 Data Transfer

The purpose of the IPI-3 data transfer library is to provide an interface to send and receive data when using IPI-3 as the data transfer protocol.

#### 1.2.2.2. 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).

#### 1.2.2.3. 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.

#### 1.2.2.4. 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.

#### 1.2.2.5. 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 non-IPI version Client API (See section 1.1.4.).

<b>libhpss.a</b>	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 ceiling
- **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
- **add64\_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 libhps 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. These processes can read and write data from a single file in parallel using HPSS's third-party transfer facilities. The interface also lets applications specify discontinuous 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**, and **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 an MPI message passing library. That is, MPI-IO is currently layered over 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.

### 1.5.4. Libraries

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

**libmpioapi.a**          MPI-IO library

In addition, the following HPSS libraries must be linked:

For non-IPI-3 applications:

**libhpss.a** HPSS utilities library (unsigned64)

For IPI-3 applications:

**libhpss\_ipi.a** HPSS utilities library (unsigned64)

**libhpssipi3.a**

**libipi3.a**

These libraries in turn require that the following are linked:

**libmpi.a**

**libEncina.a**

**libEncClient.a**

**libdce.a**

Lastly, there are platform-specific libraries on which the preceding libraries depend, and you must also include these libraries.

For AIX 4.2:

**libdcephthreads.a**

**libpthreads.a**

For AIX 4.3:

**libdcephthreads.a**

**libpthreads\_compat.a**

**libpthreads.a**

For Solaris:

**libnsl.a**

**libsocket.a**

Note: the libraries should be loaded in the order indicated.

### 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 spec-

ify an appropriate site-dependent value for:

<b>HPSS_LS_NAME</b>	HPSS Location 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.

In order to provide distributed multiprocess applications access to a user's DCE credentials cache, which is needed to authenticate a user to the HPSS servers, each user should create a keytab file. This keytab file will contain the user's DCE login name and encrypted password, which can be used to authenticate the user from each of the processes at initialization time. The keytab file must be accessible to all the processes (i.e., must be located on a file system common to all the processors on which the processes will execute).

To create the keytab file, the user must use the `rgy_edit` command interactively and after the prompt enter:

```
rgy_edit=>kta -p login_name -f keytab_path
```

The user will be prompted for a password, and asked to verify the password by retyping it. The permissions for the keytab file should be set to disallow access by the world (e.g., '640' is recommended). Note that this keytab file only needs to be created once for each environment, but it would have to be updated for any password changes for the user.

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

<b>MPIO_LOGIN_NAME</b>	<i>login_name</i>
<b>MPIO_KEYTAB_PATH</b>	<i>keytab_path</i>

where *login\_name* is as used to create the keytab file, and *keytab\_path* is the path name of the keytab file.

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

<b>MPIO_DEBUG</b>	<i>any_nonzero_integer_value</i>
-------------------	----------------------------------

## **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. DCE User Accounts**

As mentioned in the previous section, the Client API and MPI-IO requires the user be logged into DCE.

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.
ENOCCONNECT	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

<i>Path</i>	Points to the path name of the file for which client accessibility is being checked.
<i>Amode</i>	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 <i>Amode</i> are denied, or search permission is denied on a component of the path prefix.
EFAULT	The <i>Path</i> parameter is a NULL pointer.
EINVAL	An invalid value was specified for <i>Amode</i> .
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> argument points to an empty string.
ENOTDIR	A component of the <i>Path</i> prefix is not a directory.

#### See also

`hpss_Chmod`, `hpss_FileSetAttributes`.

## Chapter 2.

---

### Notes

None.

### 2.1.2. 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    Account);      /* IN */
```

#### Description

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

#### Parameters

<i>Path</i>	Names the file for which the account code is being changed.
<i>Account</i>	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 <i>Path</i> parameter is a NULL pointer.
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> argument points to an empty string.
ENOTDIR	A component of the <i>Path</i> 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_GetAcct`, `hpss_SetAcct`, `hpss_Chown`, `hpss_SetFileAttributes`.

#### Notes

None.

### 2.1.3. `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

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

#### See also

`hpss_Getcwd`, `hpss_Chroot`.

#### Notes

None.

### 2.1.4. 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

<i>Path</i>	Points to the path name of the file for which the file mode is being changed.
<i>Mode</i>	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 <i>Path</i> parameter is a NULL pointer.
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> argument points to an empty string.
ENOTDIR	A component of the <i>Path</i> 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**.

## Chapter 2.

---

### Notes

None.

### 2.1.5. 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

<i>Path</i>	Names the file for which the owner and group owner are being changed.
<i>Owner</i>	Specifies the new value for the owner of the file.
<i>Group</i>	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 <i>Path</i> parameter is a NULL pointer.
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> argument points to an empty string.
ENOTDIR	A component of the <i>Path</i> prefix is not a directory.
EPERM	The client does not have the appropriate privileges to perform the operation.

## Chapter 2.

---

### See also

`hpss_Chmod`, `hpss_Stat`, `hpss_FileGetAttributes`, `hpss_FileSetAttributes`.

### Notes

None.

## 2.1.6. 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

<i>Path</i>	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 <i>Path</i> parameter is a NULL pointer.
EINVAL	This call was made from the nonglobal Client API library.
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> argument points to an empty string.
ENOTDIR	A component of the <i>Path</i> 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.

### 2.1.7. **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.

### 2.1.8. `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.

### 2.1.9. `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.

### 2.1.10. `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

<i>Path</i>	Names the file to be opened or created.
<i>Mode</i>	Specifies the file mode used for determining the mode for the created file.
<i>HintsIn</i>	Points to an <code>hpss_cos_hints_t</code> structure which provides allocation hints to HPSS as to the expected structure or access of the file. This argument may be a NULL pointer.
<i>HintsPri</i>	Points to an <code>hpss_cos_priorities_t</code> structure which provides the relative priorities associated with the fields contained in the <i>HintsIn</i> structure. This argument may be a NULL pointer.
<i>HintsOut</i>	Points to an <code>hpss_cos_hints_t</code> 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 <i>Path</i> 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 <i>HintsIn</i> parameter is invalid.
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> 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).

### 2.1.11. `hpss_DeleteACL`

#### Purpose

Removes entries from the Access Control List of a file.

#### Synopsis

```
#include "hpss_api.h"
```

```
int  
hpss_DeleteACL(  
    char          *Path,          /* 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

<i>Path</i>	Names the file for which the <i>ACL</i> is being removed.
<i>ACL</i>	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 <i>Path</i> or <i>ACL</i> parameter is a NULL pointer.
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> argument points to an empty string.
ENOTDIR	A component of the <i>Path</i> 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.

### 2.1.12. **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**

<i>Fildes</i>	Specifies the file descriptor identifying the open file for which part is to be cleared.
<i>Length</i>	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.

### 2.1.13. `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

<i>Fildes</i>	Specifies the file descriptor identifying the open file of the part to clear.
<i>Offset</i>	Specifies where to begin clearing the file.
<i>Length</i>	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 <i>Length</i> or <i>Offset</i> argument is invalid.

#### See also

`hpss_Fclear`.

#### Notes

None.

### 2.1.14. `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

<i>Path</i>	Points to the path name of the file being queried.
<i>AttrOut</i>	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 <i>Path</i> or <i>AttrOut</i> parameter is a NULL pointer.
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> argument points to an empty string.
ENOTDIR	A component of the <i>Path</i> prefix is not a directory.

#### See also

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

#### Notes

None.

## 2.1.15. `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 *AttrOut*.

### Parameters

<i>Path</i>	Points to the pathname of the file being queried.
<i>Flags</i>	Specifies the flag that indicates the behavior of the call. The acceptable values are: <ul style="list-style-type: none"> <li>API_GET_STATS_FOR_LEVEL - Returns bitfile attributes at the storage level specified by the <i>StorageLevel</i> argument.</li> <li>API_GET_STATS_FOR_1STLEVEL - Returns bitfile attributes at the first storage level whether or not it contains bitfile data.</li> <li>API_GET_STATS_FOR_OPTIMIZE - Returns only <i>StripeWidth</i> and <i>OptimumAccessSize</i> for storage level zero.</li> <li>API_GET_STATS_ALL_LEVELS - Returns bitfile attributes across all storage class levels.</li> </ul>
<i>StorageLevel</i>	Specifies the specific storage level to query when the <code>API_GET_STATS_FOR_LEVEL</code> flag is used.
<i>AttrOut</i>	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 <i>Path</i> or <i>AttrOut</i> parameter is a NULL pointer.
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> 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.SCAAttrib[i].NumberOfVVs;j++)
    {
        if (AttrOut.BFSAttr.SCAAttrib[i].VVAttrib[j].PVLlist != NULL)
        {
            free(AttrOut.BFSAttr.SCAAttrib[i].VVAttrib[j].PVLlist);
        }
    }
}
```

## 2.1.16. 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

<i>Path</i>	Points to the name of the file for which attribute values are to be changed.
<i>NSSelFlags</i>	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

*BFSelFlags* 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

*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 <i>Path</i> , <i>AttrIn</i> or <i>AttrOut</i> parameter is a NULL pointer.
EINVAL	An attribute value or selection flag is invalid.
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> argument points to an empty string.
ENOSPC	Resources could not be allocated to satisfy the request.
ENOTDIR	A component of the <i>Path</i> 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.

### 2.1.17. 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

<i>NameServer</i>	Points to the name server uuid to be used for the create.
<i>FilesetAttrBits</i>	Specifies which fileset attributes are to be set.
<i>FilesetAttrs</i>	Points to the fileset attributes to be set.
<i>ObjectAttrBits</i>	Specifies which object attributes are to be set.
<i>ObjectAttrs</i>	Points to the object attributes to be set.
<i>RetFilesetAttrBits</i>	Specifies which fileset attributes were set.
<i>RetObjectAttrBits,</i>	Specifies which object attributes were set.
<i>RetFilesetAttrs</i>	Points to the fileset attributes that were set.
<i>RetObjectAttrs</i>	Points to the object attributes that were set.
<i>FilesetHandle</i>	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.

### 2.1.18. `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

<i>Name</i>	Specifies the name of the fileset to be deleted.
<i>FilesetId</i>	Specifies the id of the fileset to be deleted.
<i>FilesetHandle</i>	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 than 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.

## 2.1.19. hpss\_FilesetGetAttributes

### Purpose

Get attributes for an HPSS fileset.

### Synopsis

```
#include "hpss_api.h"
```

```
int
```

```
hpss_FileGetSetAttributes(
    char          Name,          /* IN */
    u_signed64   *FilesetId,    /* IN */
    ns_ObjHandle_t *FilesetHandle, /* 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.

### Parameters

<i>Name</i>	Specifies the name of the fileset to retrieve attributes for.
<i>FilesetId</i>	Specifies the id of the fileset to retrieve attributes for.
<i>FilesetHandle</i>	Specifies the handle of the fileset to retrieve attributes for.
<i>FilesetAttrBits</i>	Specifies the fileset attribute bits that specify the fileset attributes to retrieve.
<i>FilesetAttrs</i>	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 than 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.

## 2.1.20. `hpss_FilesetListAll`

### Purpose

Obtain a list of all the HPSS filesets.

### Synopsis

```
#include "hpss_api.h"
```

```
int
hpss_FilesetListAll(
    u_signed64          OffsetIn,          /* IN */
    unsigned long      Entries,          /* IN */
    unsigned long      *End,              /* OUT */
    u_signed64         *OffsetOut,        /* OUT */
    hpss_global_fsent_t *FSentPtr;        /* OUT */
```

### Description

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

### Parameters

<i>OffsetIn</i>	Specifies 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 provided the value returned in <i>OffsetOut</i> .
<i>Entries</i>	Specifies the size of the entry buffer, <i>FSentPtr</i> , in <code>hpss_global_fsent_t</code> entries.
<i>End</i>	Points to an area to contain indication of whether the last fileset entry is included in the returned list.
<i>OffsetOut</i>	Points to area to contain offset of the next fileset entry following those returned by this call.
<i>FSentPtr</i>	Points to area to contain returned fileset entries.

### 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	The <i>End</i> , <i>OffsetOut</i> or <i>DirenPtr</i> parameter is a NULL pointer.
--------	---

### See also

`hpss_FilesetCreate`.

### Notes

This API is used by setting `OffsetIn` to the starting point for the lookup (usually zero). 'Entries' is the number of filesets entries for which you have allocate space. `OffsetOut` is the point that the lookup as at when it accumulated the specified number of entries. This is typically used to specify the new starting offset (`OffsetIn`). `End` is a flag indicating that the end of the list was encountered before the all the entries were accumulated.

### 2.1.21. 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.

#### Parameters

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

<i>Fildes</i>	Specifies file descriptor identifying file to be queried.
<i>Length</i>	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.

### 2.1.22. **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

<i>Fildes</i>	Specifies the file descriptor identifying the file to be queried.
<i>Buf</i>	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 <i>Buf</i> 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.

### 2.1.23. **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

<i>Fildes</i>	Specifies the file descriptor identifying the open file for which the length is to be set.
<i>Length</i>	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.

### 2.1.24. **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. If the value returned in *RetDefAcct* is **HPSS\_ACCT\_USE\_UID** (currently defined as -1), this indicates that the client is configured for Unix-style accounting.

#### **Parameters**

<i>RetDefAcct</i>	Points to an area that will contain the default account code.
<i>RetCurAcct</i>	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 <i>RetDefAcct</i> or <i>RetCurAcct</i> parameter is a NULL pointer.
--------	---

#### **See also**

**hpss\_SetAcct**, **hpss\_Chacct**.

#### **Notes**

None.

### 2.1.25. hpss\_GetACL

#### Purpose

Query the Access Control List of a file.

#### Synopsis

```
#include "hpss_api.h"

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

#### Description

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

#### Parameters

<i>Path</i>	Names the file for which the <i>ACL</i> is being queried.
<i>ACL</i>	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

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



## 2.1.27. **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

<i>ConfigOut</i>	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 <i>ConfigOut</i> parameter is a NULL pointer.
--------	---

### See also

**hpss\_SetConfiguration.**

### Notes

None.

## 2.1.28. `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

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

### 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 <i>Buf</i> parameter is a NULL pointer.
EINVAL	The <i>Size</i> argument is zero.
ERANGE	The <i>Size</i> 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.

### 2.1.29. `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

<i>Path</i>	Points to the path name of the file being queried.
<i>AttrOut</i>	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 <i>Path</i> or <i>AttrOut</i> parameter is a NULL pointer.
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> argument points to an empty string.
ENOTDIR	A component of the <i>Path</i> 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.

### 2.1.30. `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

<i>Path</i>	Specifies path name of the new junction.
<i>SourceHandle</i>	Points to the directory or fileset handle that is used for the source of the new junction.
<i>JunctionHandle</i>	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 <i>Path</i> or <i>SourceHandle</i> parameter is a NULL pointer.
ENAMETOOLONG	The length of the <i>Path</i> argument exceeds the system imposed limit, or a component of the pathname exceeds the system imposed limit.
ENOENT	The <i>Path</i> argument points to an empty string.
EEXIST	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 <i>SourceHandle</i> parameter doesn't point to a directory handle.

**See also**

**hpss\_FilesetCreate, hpss\_JunctionDelete.**

**Notes**

None.

### 2.1.31. **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**

EENOENT	The <i>Path</i> parameter is an empty string or doesn't refer to an existing object
EFAULT	The <i>Path</i> parameter is NULL.
EINVAL	The <i>Path</i> 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.

### 2.1.32. 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

<i>Existing</i>	Specifies the path name of the existing file to which the link is to be created.
<i>New</i>	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 <i>New</i> already exists.
EFAULT	The <i>Existing</i> or <i>New</i> parameter is a NULL pointer.
EPERM	The object specified by <i>Existing</i> is a directory.
EMLINK	The number of links to the file named by <i>Existing</i> would exceed the system-imposed limit.
ENAMETOOLONG	The length of the <i>Existing</i> or <i>New</i> 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.

### 2.1.33. 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

<i>UserID</i>	Specifies the user ID for the user whose credentials are to be loaded.
<i>Umask</i>	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

<code>ENOENT</code>	Credentials for the specified user could not be obtained.
---------------------	---

#### See also

`hpss_XLoadThreadState`, `hpss_LoadDefaultThreadState`

#### Notes

None.

### 2.1.34. `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

<i>UserID</i>	Specifies the user ID for the user whose credentials are to be loaded.
<i>Umask</i>	Specifies the new file/directory creation mask.
<i>ClientFullName</i>	Specifies the client full qualified 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

<code>ENOENT</code>	Credentials for the specified user could not be obtained.
---------------------	---

#### See also

`hpss_LoadThreadState`, `hpss_XLoadThreadState`

#### Notes

None.

### 2.1.35. `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

<i>Fildes</i>	Specifies the open file handle for which the file offset is to be set.
<i>Offset</i>	Specifies the number of bytes to be used in calculating the new file offset - dependent on the value of <i>Whence</i> as to the final effect on the new file offset.
<i>Whence</i>	Specifies how to interpret the <i>Offset</i> parameter in setting the file pointer associated with the <i>Fildes</i> parameter. Values for the <i>Whence</i> parameter are as follows:  SEEK_SET - file offset set to <i>Offset</i> .  SEEK_CUR - file offset set to current offset plus <i>Offset</i> .  SEEK_END - file offset set to current end of file plus <i>Offset</i> .

#### 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 <i>Whence</i> 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.

### 2.1.36. `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

<i>Path</i>	Points to the path name of the file being queried.
<i>Buf</i>	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

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

## Chapter 2.

---

### Notes

None.

### 2.1.37. 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 */
    unsigned long     SrcLevel,         /* IN */
    unsigned long     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

<i>Fildes</i>	Specifies the file descriptor corresponding to the file to be migrated.
<i>SrcLevel</i>	Identifies the level in the storage hierarchy from which the data is to be migrated.
<i>Flags</i>	Controls the behavior of the migrate request. Anticipated values include:  BFS_MIGRATE_ALL - migrate entire file (required).
<i>RetBytesMigrated</i>	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 <i>RetBytesMigrated</i> parameter is a NULL pointer.
EINVAL	The <i>Flags</i> 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.

### 2.1.38. 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

<i>Path</i>	Specifies the path name to be used for the newly created directory.
<i>Mode</i>	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 <i>Path</i> 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 <i>Path</i> 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 <i>Path</i> argument points to an empty string.
ENOTDIR	A component of the <i>Path</i> prefix is not a directory.

### See also

`hpss_Umask`, `hpss_Rmdir`.

### Notes

None.

## 2.1.39. 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

<i>Path</i>	Names the file to be opened or created.
<i>Oflag</i>	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.
	<ul style="list-style-type: none"> <li>O_RDONLY</li> <li>O_WRONLY</li> <li>O_RDWR</li> <li>O_APPEND</li> <li>O_CREAT</li> <li>O_EXCL</li> <li>O_TRUNC</li> </ul>
<i>Mode</i>	Specifies the file mode for a file that is created as a result of <code>O_CREAT</code> .
<i>HintsIn</i>	Points to an <b>hpss_cos_hints_t</b> 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.
<i>HintsPri</i>	Points to an <b>hpss_cos_priorities_t</b> 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.

### 2.1.40. 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

<i>BitFileID</i>	Points to bitfile identifier.
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
<i>Ucred</i>	Points to client's user credentials.
<i>AuthzTicket</i>	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 <i>BitFileID</i> or <i>AuthzTicket</i> parameter is a NULL pointer.
EINPROGRESS	The file is currently being staged. The open should be retried at a later time.

EINVAL	<i>Oflag</i> 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**

None.

### 2.1.41. **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 <i>Dirname</i> parameter is a NULL pointer.
EMFILE	The open file table is already full.
ENAMETOOLONG	The length of the <i>Dirname</i> 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 <i>Dirname</i> argument points to an empty string.
ENOTDIR	A component of <i>Dirname</i> 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.

### 2.1.42. **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 */
    unsigned long     StorageLevel,    /* IN */
    unsigned long     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**

<i>Fildes</i>	Specifies the file descriptor corresponding to the file to be purged.
<i>Offset</i>	Specifies the offset of the start of the data to be purged. Currently must be 0.
<i>Length</i>	Specifies the length of the data to be purged. Currently must be 0.
<i>StorageLevel</i>	Identifies the level in the storage hierarchy from which the data is to be purged.
<i>Flags</i>	Controls the behavior of the purge request. Valid values include: BFS_PURGE_ALL - purge the entire file (required).
<i>RetBytesPurged</i>	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 <i>RetBytesPurged</i> parameter is a NULL pointer.
EINVAL	The <i>Flags</i> , <i>Offset</i> or <i>Length</i> 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.

### 2.1.43. **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**

<i>Fildes</i>	Specifies the file descriptor corresponding to the file to be locked/unlocked.
<i>Flag</i>	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.

## 2.1.44. `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

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

### See also

`hpss_SetLoginContext`.

### Notes

None.

### 2.1.45. **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

<i>Fildes</i>	Specifies the open file handle associated with the file from which data is to be read.
<i>Buf</i>	Point to a buffer where the data is to be placed.
<i>Nbyte</i>	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 <i>Buf</i> 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.

### 2.1.46. `hpss_ReadAttrs`

#### Purpose

Read directory entries and optionally return entry attributes.

#### Synopsis

```
#include "hpss_api.h"

int
hpss_ReadAttrs(
    int                Dirdes,           /* IN */
    unsigned long     OffsetIn,         /* IN */
    unsigned long     BufferSize,       /* IN */
    unsigned long     GetAttributes,    /* IN */
    unsigned long     *End,             /* OUT */
    unsigned long     *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

<i>Dirdes</i>	Specifies the open directory stream handle corresponding to the directory being read.
<i>OffsetIn</i>	Specifies the starting directory offset. If zero, the list will be from the beginning of the directory.
<i>BufferSize</i>	Specifies the size of the buffer pointed to by <i>DirentPtr</i> , in bytes. The maximum number of entries that fit in the buffer will be returned, until the end of the directory is reached.
<i>GetAttributes</i>	Indicates, if nonzero, that attributes will be returned with the directory entries.
<i>End</i>	Points to an area that will contain an indication of whether the last entry is returned in the current list.
<i>OffsetOut</i>	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.
<i>DirentPtr</i>	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 <i>DirentPtr</i> , <i>End</i> or <i>OffsetOut</i> parameter is a NULL pointer.
EINVAL	The <i>BufferSize</i> 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**.

### 2.1.47. **hpss\_Readdir**

#### Purpose

Read a directory entry.

#### Synopsis

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

int
hpss_Readdir(
    int                Dirdes,          /* IN */
    struct dirent     *DirentOut);     /* OUT */
```

#### Description

The **hpss\_Readdir** function returns a structure, *DirentOut*, representing the directory entry at the current position in the open directory stream. Reference POSIX.1 for more detailed information.

#### Parameters

<i>Dirdes</i>	Specifies the open directory stream handle corresponding to the directory being read.
<i>DirentOut</i>	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 <i>DirentOut</i> 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 *DirentOut* structure.

### 2.1.48. **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 */
    unsigned long 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

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

#### 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 <i>Path</i> or <i>Contents</i> parameter is a NULL pointer.
EINVAL	The specified file is not a symbolic link.
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> prefix is not a directory.
ERANGE	The size of the <i>Contents</i> 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.

## 2.1.49. 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 */
    unsigned long  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

<i>IODPtr-&gt;Function</i>	Specifies the IOD function type. Set to <b>IOD_READ</b> .
<i>IODPtr-&gt;SrcDescLength</i>	Specifies the number of entries in the source descriptor list.
<i>IODPtr-&gt;SinkDescLength</i>	Specifies the number of entries in the sink descriptor list.
<i>IODPtr-&gt;SrcDescList</i>	Specifies a list of descriptors specifying the parts of the file to be read.
<i>IODPtr-&gt;SinkDescList</i>	Specifies a list of descriptors containing the destinations of the data.
<i>Flags</i>	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).
<i>IORPtr-&gt;RequestID</i>	Specifies request identifier assigned to this request by the Client API.
<i>IORPtr-&gt;Flags</i>	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 encountered during the read.
<i>IORPtr-&gt;ReqSpecReply</i>	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 <i>IORPtr-&gt;Flags</i> field.
<i>IORPtr-&gt;SrcReplyLength</i>	Specifies the number of source reply descriptors.
<i>IORPtr-&gt;SinkReplyLength</i>	Specifies the number of sink reply descriptors.
<i>IORPtr-&gt;SrcReplyList</i>	Specifies a list of descriptors containing the data source results.
<i>IORPtr-&gt;SinkReplyList</i>	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 <i>Flags</i> 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.

## 2.1.50. 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

<i>Old</i>	Specifies the path name that currently names the file or directory.
<i>New</i>	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 <i>Old</i> or <i>New</i> denies write permission, or write permission is required and denied for a directory pointed to by the <i>Old</i> or <i>New</i> arguments.
EFAULT	The <i>Old</i> or <i>New</i> parameter is a NULL pointer.
EISDIR	The <i>New</i> argument points to a directory, and the <i>Old</i> argument points to a file that is not a directory.
EMLINK	The file named by <i>Old</i> is a directory, and the link count of the parent directory of <i>New</i> already contains the maximum allowed number of links.
ENAMETOOLONG	The length of the <i>Old</i> or <i>New</i> 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 <i>Old</i> or <i>New</i> 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.

## 2.1.51. 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); /* OUT */
```

### Description

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

### Parameters

<i>Fildes</i>	Specifies the file handle for the currently open file.
<i>BitFileID</i>	Points to the bitfile ID of the file to be reopened.
<i>Oflags</i>	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
<i>Ucred</i>	Points to client's user credentials.
<i>AuthzTicket</i>	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	<i>Fildes</i> does not refer to an open file.
EBUSY	The open file descriptor is in use by another thread.

EFAULT	The <i>BitFileID</i> or <i>AuthzTicket</i> 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.

## 2.1.52. `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

<i>Dirdes</i>	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_Openidir`, `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.

### 2.1.53. **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 <i>Path</i> parameter is a NULL pointer.
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> argument points to an empty string.
ENOTDIR	A component of the <i>Path</i> 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.

### 2.1.54. **hpss\_SetACL**

#### **Purpose**

Set the Access Control List of a file.

#### **Synopsis**

```
#include "hpss_api.h"

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

#### **Description**

The **hpss\_SetACL** function replaces the access control list for the file named by *Path*.

#### **Parameters**

<i>Path</i>	Names the file for which the <i>ACL</i> is being replaced.
<i>ACL</i>	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 <i>Path</i> or <i>ACL</i> parameter is a NULL pointer.
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> argument points to an empty string.
ENOTDIR	A component of the <i>Path</i> 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.

### 2.1.55. **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**

<i>NewCurAcct</i>	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\_GetAcct**, **hpss\_Chacct**.

#### **Notes**

None.



### 2.1.57. **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**

<i>ConfigIn</i>	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 <i>ConfigIn</i> parameter is a NULL pointer.
EINVAL	Invalid configuration attribute value setting.

#### **See also**

**hpss\_GetConfiguration, hpss\_ClientAPIReset.**

#### **Notes**

None.

## 2.1.58. `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

<i>Filedes</i>	Specifies the open file handle for which the Class of Service is to be set.
<i>Flags</i>	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.
<i>HintsPtr</i>	Points to a structure that provides attribute values for selection of a new Class of Service or storage segment size.
<i>PrioPtr</i>	Points to a structure that contains relative priorities for the attribute values indicated by the <i>HintsPtr</i> parameter.
<i>COSPtr</i>	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 <i>HinstPtr</i> , <i>PrioPtr</i> , or <i>COSPtr</i> 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 <i>Path</i> prefix is not a directory.

### See also

**hpss\_Open, hpss\_OpenBitfile.**

### Notes

None.

## 2.1.59. `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

<i>Fildes</i>	Specifies the open file handle for which the file offset is to be set.
<i>OffsetIn</i>	Specifies the number of bytes to be used in calculating the new file offset - dependent on the value of <i>Whence</i> and <i>Direction</i> as to the final effect on the new file offset.
<i>Whence</i>	Specifies how to interpret the <i>OffsetIn</i> parameter in setting the file pointer associated with the <i>Fildes</i> parameter. Values for the <i>Whence</i> parameter are as follows:  SEEK_SET - file offset set to <i>OffsetIn</i> . SEEK_CUR - file offset set to current offset plus <i>OffsetIn</i> . SEEK_END - file offset set to current end of file plus <i>OffsetIn</i> .
<i>Direction</i>	Specifies whether <i>OffsetIn</i> should be used to move forward or backward in the file.  HPSS_SET_OFFSET_FORWARD - consider <i>OffsetIn</i> as being a nonnegative number. HPSS_SET_OFFSET_BACKWARD - consider <i>OffsetIn</i> as being a negative number.
<i>OffsetOut</i>	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	<i>OffsetOut</i> is a NULL pointer.
EINVAL	The <i>Whence</i> or <i>Direction</i> 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.

## 2.1.60. hpss\_SetLoginContext

### Purpose

Establish a security context for an application.

### Syntax

```
#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

<i>PrincipalName</i>	Specifies the server's principal name.
<i>KeytabName</i>	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

<code>HPSS_E_NOERROR</code>	Successful completion
<code>HPSS_EINVAL</code>	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`.

### 2.1.61. 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 */
    unsigned long      StorageLevel,    /* IN */
    unsigned long      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

<i>Fildes</i>	Specifies the file descriptor, identifying the file to be staged.
<i>Offset</i>	Specifies the offset of the start of the data to be staged.
<i>Length</i>	Specifies the length of the data to be staged.
<i>StorageLevel</i>	Identifies the level in the storage hierarchy to which the data is to be staged. Currently, the only supported value is 0.
<i>Flags</i>	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 <i>Flags</i> , <i>Offset</i> or <i>Length</i> 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.

### 2.1.62. 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 */
    unsigned long       StorageLevel,   /* IN */
    unsigned long       Flags);        /* IN */
```

#### 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

<i>Path</i>	Specifies the pathname of the file to be staged.
<i>Offset</i>	Specifies the offset of the start of the data to be staged.
<i>Length</i>	Specifies the amount of the data to be staged.
<i>StorageLevel</i>	Identifies the level in the storage hierarchy to which the data is to be staged.
<i>Flags</i>	Controls the behavior of the stage request. Valid values include:  BFS_STAGE_ALL - stage entire 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

EACCESS	Search permission is denied for a component of the path prefix.
EFAULT	The <i>Path</i> parameter is a NULL pointer.
EINVAL	The value of the <i>Offset</i> parameter is beyond the end of the file or the <i>StorageLevel</i> parameter is invalid for the storage hierarchy.

ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> argument points to an empty string.
ENOTDIR	A component of the <i>Path</i> prefix is not a directory.

**See also**

**hpss\_Migrate, hpss\_Purge, hpss\_Stage.**

**Notes**

None.

### 2.1.63. 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

<i>Path</i>	Points to the path name of the file being queried.
<i>Buf</i>	Points to a <b>stat</b> 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 <i>Path</i> or <i>Buf</i> parameter is a NULL pointer.
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> argument points to an empty string.
ENOTDIR	A component of the <i>Path</i> 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.

### 2.1.64. 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

<i>COSId</i>	Specifies the identifier for the Class of Service that is being queried.
<i>StatfsBuffer</i>	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 <i>StatfsBuffer</i> parameter is a NULL pointer.
EINVAL	The specified Class of Service does not exist.

#### See also

None.

#### Notes

None.

## 2.1.65. `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

<i>Contents</i>	Specifies the path name to which the symbolic link will point.
<i>Path</i>	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 <i>Path</i> or <i>Contents</i> parameter is a NULL pointer.
EEXIST	The specified file already exists.
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> prefix is not a directory.

### See also

`hpss_Readlink`.

## Chapter 2.

---

### Notes

None.

## 2.1.66. `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

<i>ThreadID</i>	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.

### 2.1.67. **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**

<i>Path</i>	Specifies the path name of the file to be truncated.
<i>Length</i>	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 <i>Path</i> parameter is a NULL pointer.
EINVAL	<i>Path</i> specifies a directory.
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> prefix is not a directory.

#### **See also**

**hpss\_Ftruncate, hpss\_Fclear, hpss\_FileSetAttributes.**

**Notes**

None.

### 2.1.68. **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.

## 2.1.69. `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 <i>Path</i> parameter is a NULL pointer.
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> argument points to an empty string.
ENOTDIR	A component of the <i>Path</i> prefix is not a directory.
EPERM	The file named by <i>Path</i> 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.

## 2.1.70. hpss\_UpdateACL

### Purpose

Update entries in the Access Control List of a file.

### Synopsis

```
#include "hpss_api.h"
```

```
int
hpss_UpdateACL(
    char                *Path,           /* IN */
    unsigned long       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

<i>Path</i>	Names the file for which the <i>ACL</i> is being updated.
<i>Options</i>	Defines a bit vector containing bits which control the behavior of <b>ns_UpdateACL</b> while calculating the MASK_OBJ. See the Notes below for more information.
<i>ACL</i>	Points to the <i>ACL</i> 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 <i>Path</i> or <i>ACL</i> parameter is a NULL pointer.
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> argument points to an empty string.
ENOTDIR	A component of the <i>Path</i> 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.

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 constants:

#### `DONT_CALCULATE_MASK`

Specifies that a new mask should not be calculated. This option is useful only for objects that are required to recalculate a new mask after they are modified. If a modify operation creates a mask that unintentionally adds permissions to an existing ACL entry, the modify causing the mask recalculation will abort with an error unless you specify either the `CALCULATE_MASK_IGNORE_ERRORS` or `DONT_CALCULATE_MASK` option.

#### `CALCULATE_MASK_IGNORE_ERRORS`

Creates or modifies the object's `MASK_OBJ` type 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 is set even if it grants permissions previously masked out. It is recommended that you use this option only if not specifying it results in an error. This option is useful only for objects that support the `MASK_OBJ` type and are required to recalculate a new mask after they are modified.

If a modify operation creates a mask that unintentionally adds permissions to an existing ACL entry, the modify causing the mask recalculation will abort with an error unless you specify either the `CALCULATE_MASK_IGNORE_ERRORS` or `DONT_CALCULATE_MASK` option.

#### `PURGE_MASK_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 is calculated as a result of adding or modifying an ACL entry.

If an update operation creates a `MASK_OBJ` that unintentionally adds permissions to an existing `acl` entry, the modify causing the `MASK_OBJ` recalculation will abort with an error unless you specify either the `CALCULATE_MASK_IGNORE_ERRORS` or `DONT_CALCULATE_MASK` option.

## 2.1.71. 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

<i>Path</i>	Names the file for which times are being changed.
<i>Times</i>	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 <i>Path</i> parameter is a NULL pointer.
ENAMETOOLONG	The length of the <i>Path</i> 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 <i>Path</i> argument points to an empty string.
ENOTDIR	A component of the <i>Path</i> 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.

## 2.1.72. `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

<i>Fildes</i>	Specifies the open file handle associated with the file to which data is to be written.
<i>Buf</i>	Points to a buffer where the data is to be found.
<i>Nbyte</i>	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 <i>Buf</i> 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.

## Chapter 2.

---

### See also

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

### Notes

None.

### 2.1.73. 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 */
    unsigned long  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

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

<i>IORPtr-&gt;SrcReplyList</i>	Specifies a list of descriptors specifying the data source results.
<i>IORPtr-&gt;SinkReplyList</i>	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 <i>Flags</i> 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

## 2.1.74. 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

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

## 2.1.75. 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

<i>IORPtr</i>	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 addition 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\_PVRetrievals** 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

<i>PVName</i>	The physical volume name.
<i>MountCntSinceService</i>	The number of mounts since last service.
<i>MountCntSinceMaint</i>	The number of mounts since last maintenance.
<i>NumReads</i>	The number of reads.
<i>NumWrites</i>	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;
} 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.

### 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;
} 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

### **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];
    u_signed64      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_attr_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;
    hpssioid_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     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 Bit file 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 identifies 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

## Chapter 2.

---

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.

### TimeOfMetadataUpdate

Specifies the last time the metadata was updated.

### Type

Specifies the 'type' of the object:

DIRECTORY_OBJECT	directory
FILE_OBJECT	file
JUNCTION_OBJECT	junction
SYM_LINK_OBJECT	symbolic link
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          RegisterBitMap;
    unsigned32         ClassOfService;
    unsigned32         FamilyId;
    ns_ObjHandle_t     FilesetHandle;
    u_signed64         FilesetId;
    char               FilesetName[NS_FS_MAX_FS_NAME_LENGTH];
    unsigned32         FilesetType;
    uuid_t             GatewayUUID;
    unsigned32         StateFlags;
    unsigned32         SubSystemId;
    byte               UserData[NS_FS_MAX_USER_DATA];
    signed32           DirectoryCount;
    signed32           FileConunt;
    signed32           HardLinkCount;
    signed32           JunctionCount;
    signed32           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.

#### FilesetHandle

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 meta data for the object.)

#### FileId

If the Type field specifies a hardlink this is the RSN of the SFS record containing the meta data 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\_ROOT    Handle 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 valid 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 current 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;
    cos_t               COSId;
    cos_t               NewCOSId;
    acct_rec_t          Acct;
    unsigned32          Flags;
    unsigned32          StorageSegMult;
    bfs_owner_rec_t     OwnerRec;
    u_signed64          RegisterBitmap;
    security_t          Security;
} 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\_attr struct will be set to indicate which reverse map entries have changed.

### Security

This field is not currently used.

### **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;
```

**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.

## Chapter 2.

---

### 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:

<b>BFS_BFATTRS_LEVEL_IS_DISK</b>	This is a disk storage level.
<b>BFS_BFATTRS_LEVEL_IS_TAPE</b>	This is a tape storage level.
<b>BFS_BFATTRS_DATAEXIST_AT_LEVEL</b>	Data for the bitfile exists at this level.
<b>BFS_BFATTRS_ADDITIONAL_VV_EXIST</b>	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_attr_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_attr {
    hpsoid_t      VVID;
    signed32     RelPosition;
    pv_list_t    *PVList;
} bf_vv_attr_t;
```

### VVID

Specifies the virtual volume identifier.

### RelPosition

Specifies the relative start position of first bitfile segment on this virtual volume.

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[1];
} 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:

## Chapter 2.

---

<b>API_INIT_HSEC</b>	Client API should perform HPSS security initialization.
<b>API_INIT_HSEC</b>	Client API should perform HPSS security initialization.
<b>API_INIT_TRPC</b>	Client API should perform TRPC initialization.
<b>API_ENABLE_LOGGING</b>	If logging compiled into Client API library, perform HPSS logging on errors.
<b>API_GLOBAL_FILETABLE</b>	Indicates whether Client API was built using global or per-thread file table (this flag is informational only - it cannot be set hpss_SetConfiguration).
<b>API_USE_ENV</b>	Modify configuration based on environment variables.

### 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. Valid values include:

<b>API_TRANSFER_TCP</b>	Use TCP/IP
<b>API_TRANSFER_IPI3</b>	Use IPI-3 over HIPPI

### 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 or 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[1];
} 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 {
    unsigned char     EntryType;
    unsigned char     Perms;
    unsigned16        ExpirationDate;
    unsigned32        EntryId;
    unsigned32        Location;
} 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.

### 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\_AttrBits\_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;
```

## **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:

SEQUENTIAL_IO	When set, the caller wishes to process the data sequentially in the order requested. If this bit is not set, HPSS is free to send the data out of order in an effort to optimize the resource utilization. Note that this flag is currently unused.
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.
LAST_IN_XFER	For read and write requests via IPI-3 transfers, indicates that the IOD contains the last byte in the client transfer.

### 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. This has consequences for IPI-3 transfers that may involve third party data movement directly between a device and client.
XFEROPT_IP	When set, indicates that the client is capable of transferring data using TCP/IP.
XFEROPT_IPI3	When set, indicates that the client is capable of transferring data using IPI-3 over HIPPI.
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.

**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:

```
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;
    } Addr_u;
} address_t;
```

### Type

This field indicates the type of address contained in Addr\_u. Values are:

NET_ADDRESS	TCP/IP
IPI_ADDRESS	IPI-3
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

### 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 {
    unsigned32SockTransferID;
    struct {
        signed32    addr;
        signed32    port;
        signed32    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.

### 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.

### IPIAddr

This field contains IPI-3 addressing information. It will be used for clients or devices which are using IPI-3 data transfer protocols. The structure has the following format:

```
typedef struct ipiaddress {
    unsigned32IPI3TransferID;
    struct {
        signed16    Interface;
        char        Name[32];
    }IPI3Addr;
    u_signed64IPI3Offset;
} ipiaddress_t;
```

### IPI3TransferID

This field contains the transfer ID which will be used in the IPI-3 header for the data described by this source/sink descriptor.

### IPI3Addr

This field contains the IPI-3 addressing information necessary to perform the data transfer. The subfields include:

#### Interface

This field contains the IPI-3 interface to be used for the transfer (e.g., "IPI3\_HIPPI").

#### Name

This field contains the name of the client machine (used to look up the correct i-field in the IPI-3 configuration).

### IPI3Offset

This field contains the offset to be used in the IPI-3 header for the data described by this source/sink descriptor.

### 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 `hpsd_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.

### 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. This has consequences for IPI-3 transfers that may involve third party data movement directly between a device and client.
XFEROPT_IP	When set, indicates that the client is capable of transferring data using TCP/IP.
XFEROPT_IPI3	When set, indicates that the client is capable of transferring data using IPI-3 over HIPPI.
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 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;
    } 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.

### 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



### 4.1.1.2. ipi3\_data3\_close

#### Purpose

Close an open IPI-3 transfer end point.

#### Synopsis

```
#include "ipi3defs.h"  
#include "ipi3rc.h"
```

```
int  
ipi3_data3_close(  
    int    TransferDescriptor);    /* IN */
```

#### Description

The `ipi3_data3_close` routine closes an IPI-3 data transfer end point, freeing any allocated resources.

#### Parameters

*TransferDescriptor*                      The transfer descriptor corresponding to the end point to be closed.

#### Return Values

If the IPI-3 end point is successfully closed, the value `IPI3_RC_OK` is returned. If an error is encountered, a negative value is returned that indicates the cause of the error.

#### Error Conditions

`IPI3_RC_INITIALIZE_FAIL`              The IPI-3 library did not correctly initialize.  
  
`IPI3_RC_NOT_OPEN`                      *TransferDescriptor* does not correspond to a currently open IPI-3 end point.

#### See Also

`ipi3_data3_open`.

#### Notes

None.

### 4.1.1.3. ipi3\_data3\_read

#### Purpose

Initiate an asynchronous IPI-3 read operation.

#### Synopsis

```
#include "ipi3defs.h"
#include "ipi3rc.h"

int
ipi3_data3_read(
    int          TransferDescriptor,      /* IN */
    unsigned long* RetTransferID,        /* OUT */
    unsigned long Count,                 /* IN */
    char         *Buf);                  /* OUT */
```

#### Description

The `ipi3_data3_read` routine initiates the master side of an IPI-3 read request. The routine returns after the end point is prepared for the operation, and returns the transfer identifier generated for this request.

#### Parameters

<i>TransferDescriptor</i>	The transfer descriptor corresponding to the end point over which the data is to be read.
<i>RetTransferID</i>	Pointer to an area which will contain the returned transfer identifier.
<i>Count</i>	The number of bytes to be read.
<i>Buf</i>	Pointer to an area which will contain the received data.

#### Return Values

If the operation is successfully initialized, then the value `IPI3_RC_OK` is returned. If an error is encountered, a negative value is returned that indicates the cause of the error.

#### Error Conditions

<code>IPI3_RC_INITIALIZE_FAIL</code>	The IPI-3 library did not correctly initialize.
<code>IPI3_RC_TOO_BIG</code>	The <i>Count</i> parameter exceeds the maximum transfer size supported by the IPI-3 library.
<code>IPI3_RC_NOT_OPEN</code>	<i>TransferDescriptor</i> does not correspond to a currently open IPI-3 end point.
<code>IPI3_RC_BUSY</code>	The end point corresponding to <i>TransferDescriptor</i> is currently in the process of satisfying another request.
<code>IPI3_RC_COMM_FAIL</code>	Could not perform initialization of the transfer operation.

### See Also

`ipi3_data3_write`, `ipi3_data3_complete`.

### Notes

The IPI-3 data transfer library currently supports a maximum transfer size of 64MB under AIX.

### 4.1.1.4. ipi3\_data3\_write

#### Purpose

Initiate an asynchronous IPI-3 write operation.

#### Synopsis

```
#include "ipi3defs.h"
#include "ipi3rc.h"

int
ipi3_data3_write(
    int          TransferDescriptor,      /* IN */
    unsigned long *RetTransferID,        /* OUT */
    unsigned long Count,                 /* IN */
    char         *Buf);                  /* IN */
```

#### Description

The `ipi3_data3_write` routine initiates the master side of an IPI-3 write request. The routine returns after the end point is prepared for the operation, and returns the transfer identifier generated for this request.

#### Parameters

<i>TransferDescriptor</i>	The transfer descriptor corresponding to the end point over which the data is to be written.
<i>RetTransferID</i>	Pointer to an area which will contain the returned transfer identifier.
<i>Count</i>	The number of bytes to be written.
<i>Buf</i>	Pointer to an area which contains the data to be written.

#### Return Values

If the operation is successfully initialized, then the value `IPI3_RC_OK` is returned. If an error is encountered, a negative value is returned that indicates the cause of the error.

#### Error Conditions

<code>IPI3_RC_INITIALIZE_FAIL</code>	The IPI-3 library did not correctly initialize.
<code>IPI3_RC_TOO_BIG</code>	The <i>Count</i> parameter exceeds the maximum transfer size supported by the IPI-3 library.
<code>IPI3_RC_NOT_OPEN</code>	<i>TransferDescriptor</i> does not correspond to a currently open IPI-3 end point.
<code>IPI3_RC_BUSY</code>	The end point corresponding to <i>TransferDescriptor</i> is currently in the process of satisfying another request.
<code>IPI3_RC_COMM_FAIL</code>	Could not perform initialization of the transfer operation.

### See Also

`ipi3_data3_read`, `ipi3_data3_complete`.

### Notes

The IPI-3 data transfer library currently supports a maximum transfer size of 64MB under AIX.

### 4.1.1.5. ipi3\_data3\_complete

#### Purpose

Wait for an IPI-3 data transfer to complete.

#### Synopsis

```
#include "ipi3defs.h"
#include "ipi3rc.h"
```

```
int
ipi3_data3_complete(
    int    TransferDescriptor,    /* IN */
    long   ActualSize);          /* IN */
```

#### Description

The **ipi3\_data3\_complete** routine waits for an outstanding IPI-3 data transfer operation to complete. If the transfer size will be less than was originally requested (e.g., when attempting to read beyond end-of-file), the *ActualSize* parameter indicates the expected number of bytes.

#### Parameters

<i>TransferDescriptor</i>	The transfer descriptor corresponding to the end point over which the data transfer is being performed.
<i>ActualSize</i>	The number of bytes actually expected, a value of -1 indicates that the byte passed to <b>ipi3_data3_read</b> or <b>ipi3_data3_write</b> is the amount expected.

#### Return Values

If the data transfer completes successfully, the number of bytes transferred is returned. If an error is encountered, a negative value is returned that indicates the cause of the error.

#### Error Conditions

IPI3_RC_INVALID_TD	<i>TransferDescriptor</i> does not correspond to a currently open IPI-3 end point.
IPI3_RC_EINTR	A signal interrupted this routine, the transfer operation is still in progress.
IPI3_RC_COMM_FAIL	An error occurred during the transfer operation.
IPI3_RC_TIMED_OUT	The transfer operation did not complete in the allotted time.

#### See Also

**ipi3\_data3\_read**, **ipi3\_data3\_write**.

#### Notes

None.

### 4.1.1.6. ipi3\_data3\_cancel

#### Purpose

Cancel a pending IPI-3 data transfer.

#### Synopsis

```
#include "ipi3defs.h"
#include "ipi3rc.h"

int
ipi3_data3_cancel(
    int    TransferDescriptor);    /* IN */
```

#### Description

The `ipi3_data3_cancel` routine aborts an outstanding IPI-3 data transfer operation.

#### Parameters

*TransferDescriptor*                      The transfer descriptor corresponding to the end point for which the operation is to be aborted.

#### Return Values

If the transfer was successfully aborted, the value `IPI3_RC_OK` is returned. If an error is encountered, a negative value is returned that indicates the cause of the error.

#### Error Conditions

`IPI3_RC_UNKOWN_STAT`                      An unexpected error occurred.

#### See Also

`ipi3_data3_read`, `ipi3_data3_write`, `ipi3_data3_complete`.

#### Notes

None.

### 4.1.2. IPI-3 Data Transfer Library Data Definitions

#### 4.1.2.1. IPI-3 Interface Address Structure - IPI3\_INTERFACE\_STRUCT

##### **Description**

The IPI-3 Interface Address Structure contains the information necessary to identify the IPI-3 end point to be used for a data transfer operation. This structure is returned from the **ipi3\_data3\_open** routine, and the information must be passed in an IOD or Mover Protocol message to allow the HPSS Mover to communicate with the client using IPI-3 over HIPPI.

##### **Format**

The IPI-3 Interface Address Structure has the following format:

```
typedef struct {
    short    interface;
    char     *name;
} IPI3_INTERFACE_STRUCT;
```

##### *interface*

Contains an indication of the network medium to be used for the data transfer. Currently, the only supported value is IPI3\_HIPPI.

##### *name*

Contains the name of the IPI-3 interface, as defined in the IPI-3 configuration files. This value is used to look up the requisite address information on the node on which the HPSS Movers are running, to allow communication with the client node.

### 4.1.3. Mover Socket (Parallel TCP/IP Data Transfer) Functions

#### 4.1.3.1. 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**

<i>SD</i>	File descriptor that refers to the TCP/IP connection on which the data is to be sent.
<i>XferID</i>	The transfer identifier to send in the parallel data transfer header.
<i>Offset</i>	The transfer offset at which this data begins.
<i>Buffer</i>	Pointer to the data buffer to be sent.
<i>Length</i>	The amount of data to be sent.
<i>Ticket</i>	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**

<code>EINVAL</code>	An input parameter is invalid.
<code>EIO</code>	An unexpected error occurred.
<code>EPIPE</code>	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.

#### 4.1.3.2. 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 *SecTimeout* elapses between messages from the peer entity, the routine will return an error indication.

##### Parameters

<i>SD</i>	File descriptor that refers to the TCP/IP connection on which the data is to be sent.
<i>XferID</i>	The transfer identifier to send in the parallel data transfer header.
<i>Offset</i>	The transfer offset at which this data begins.
<i>Buffer</i>	Pointer to the data buffer to be sent.
<i>Length</i>	The amount of data to be sent.
<i>Ticket</i>	Pointer to a security ticket to be sent in the parallel data transfer header.
<i>SecTimeout</i>	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.
--------	--------------------------------

## Chapter 4. Supplemental Data Transfer Functions

---

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.

### 4.1.3.3. 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 *SecTimeout* 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

<i>SD</i>	File descriptor that refers to the TCP/IP connection on which the data is to be sent.
<i>XferID</i>	The transfer identifier to send in the parallel data transfer header.
<i>Offset</i>	The transfer offset at which this data begins.
<i>Buffer</i>	Pointer to the data buffer to be sent.
<i>Length</i>	The amount of data to be sent.
<i>Ticket</i>	Pointer to a security ticket to be sent in the parallel data transfer header.
<i>SecTimeout</i>	Number of seconds to wait if the connection becomes idle before returning; zero specifies infinite wait.
<i>WriteSize</i>	Maximum number of bytes to be specified with each network write request.

## Chapter 4. Supplemental Data Transfer Functions

---

### 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.

### 4.1.3.4. 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

<i>SD</i>	File descriptor that refers to the TCP/IP connection on which the data is to be requested and received.
<i>XferID</i>	The transfer identifier to send in the parallel data transfer header.
<i>Offset</i>	The transfer offset at which this data begins.
<i>Buffer</i>	Pointer to the data buffer in which the received data will be stored.
<i>Length</i>	The amount of data to be received.
<i>Ticket</i>	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_rcv_hdr`, `mover_socket_get_buffer_timeout`.

**Notes**

The security ticket is currently unused by the HPSS Mover and clients.

**4.1.3.5. 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 *SecTimeout* elapses between messages from the peer entity, the routine will return an error indication.

**Parameters**

<i>SD</i>	File descriptor that refers to the TCP/IP connection on which the data is to be requested and received.
<i>XferID</i>	The transfer identifier to send in the parallel data transfer header.
<i>Offset</i>	The transfer offset at which this data begins.
<i>Buffer</i>	Pointer to the data buffer in which the received data will be stored.
<i>Length</i>	The amount of data to be received.
<i>Ticket</i>	Pointer to a security ticket to be sent in the parallel data transfer header.
<i>SecTimeout</i>	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_rcv_hdr_timeout`, `mover_set_get_buffer`.

**Notes**

The security ticket is currently unused by the HPSS Mover and clients.

**4.1.3.6. mover\_socket\_rcv\_data****Purpose**

Wait for a parallel data transfer header, and receive the specified data.

**Synopsis**

```
#include "pdata.h"
int
mover_socket_rcv_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\_rcv\_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**

<i>SD</i>	File descriptor that refers to the TCP/IP connection on which the header and data is to be received.
<i>XferID</i>	The transfer identifier to expect in the parallel data transfer header.
<i>Offset</i>	The transfer offset at which this data begins.
<i>Buffer</i>	Pointer to the data buffer in which the received data will be stored.
<i>BufSize</i>	The size of the passed buffer.
<i>BytesRecvd</i>	Place to return the number of bytes of data actually received.
<i>NumOfPackets</i>	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_rcv_hdr`, `mover_waitfor_data`, `mover_socket_rcv_data_timeout`.

**Notes**

None.

#### 4.1.3.7. 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 *SecTimeout* elapses between messages from the peer entity, the routine will return an error indication.

##### Parameters

<i>SD</i>	File descriptor that refers to the TCP/IP connection on which the header and data is to be received.
<i>XferID</i>	The transfer identifier to expect in the parallel data transfer header.
<i>Offset</i>	The transfer offset at which this data begins.
<i>Buffer</i>	Pointer to the data buffer in which the received data will be stored.
<i>BufSize</i>	The size of the passed buffer.
<i>BytesRecvd</i>	Place to return the number of bytes of data actually received.
<i>NumOfPackets</i>	Number of incoming parallel data transfer headers to process.
<i>SecTimeout</i>	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_rcv_hdr_timeout`, `mover_waitfor_data_timeout`, `mover_socket_rcv_data`.

### Notes

None.

**4.1.3.8. 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**

<i>SD</i>	File descriptor that refers to the TCP/IP connection on which the header is to be received, and the data sent.
<i>XferID</i>	The transfer identifier to expect in the parallel data transfer header.
<i>Offset</i>	The transfer offset at which this data begins.
<i>Buffer</i>	Pointer to the data buffer in which the data to be sent is located.
<i>BufLength</i>	The size of the passed buffer.
<i>BytesRecvd</i>	Place to return the number of bytes of data actually sent.
<i>NumOfPackets</i>	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.
--------	---



#### 4.1.3.9. 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 *SecTimeout* elapses between messages from the peer entity, the routine will return an error indication.

**Parameters**

<i>SD</i>	File descriptor that refers to the TCP/IP connection on which the header is to be received, and the data sent.
<i>XferID</i>	The transfer identifier to expect in the parallel data transfer header.
<i>Offset</i>	The transfer offset at which this data begins.
<i>Buffer</i>	Pointer to the data buffer in which the data to be sent is located.
<i>BufLength</i>	The size of the passed buffer.
<i>BytesRecvd</i>	Place to return the number of bytes of data actually sent.
<i>NumOfPackets</i>	Number of incoming parallel data transfer headers to process.
<i>SecTimeout</i>	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\_rcv\_hdr\_timeout, pdata\_send\_hdr\_and\_data\_timeout, mover\_waitfor\_requests\_timeout, mover\_send\_requested\_data.**

### Notes

None.

4.1.3.10. 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 *SecTimeout* 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**

<i>SD</i>	File descriptor that refers to the TCP/IP connection on which the header is to be received, and the data sent.
<i>XferID</i>	The transfer identifier to expect in the parallel data transfer header.
<i>Offset</i>	The transfer offset at which this data begins.
<i>Buffer</i>	Pointer to the data buffer in which the data to be sent is located.
<i>BufLength</i>	The size of the passed buffer.
<i>BytesRecvd</i>	Place to return the number of bytes of data actually sent.
<i>NumOfPackets</i>	Number of incoming parallel data transfer headers to process.
<i>SecTimeout</i>	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_rcv_hdr_timeout`, `pdata_send_hdr_and_data_timeout_size`,  
`mover_waitfor_requests_timeout`, `mover_send_requested_data_size`.

### Notes

None.

### 4.1.3.11. 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

<i>SD</i>	File descriptor that refers to the listen socket.
<i>XferID</i>	The transfer identifier to expect in the parallel data transfer headers.
<i>Offset</i>	The transfer offset at which this data begins.
<i>Buffer</i>	Pointer to the data buffer in which the data will be stored.
<i>BufSize</i>	The size of the passed buffer.
<i>BytesMoved</i>	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_rcv_hdr`, `mover_waitfor_data_timeout`.

**Notes**

This routine currently supports 32 simultaneous open connections.

4.1.3.12. 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_socket_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 *SecTimeout* 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**

<i>SD</i>	File descriptor that refers to the listen socket.
<i>XferID</i>	The transfer identifier to expect in the parallel data transfer headers.
<i>Offset</i>	The transfer offset at which this data begins.
<i>Buffer</i>	Pointer to the data buffer in which the data will be stored.
<i>BufSize</i>	The size of the passed buffer.
<i>BytesMoved</i>	Place to return the number of bytes of data actually received.
<i>SecTimeout</i>	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\_rcv\_hdr\_timeout, mover\_waitfor\_data.**

**Notes**

This routine currently supports 32 simultaneous open connections.

**4.1.3.13. 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**

<i>SD</i>	File descriptor that refers to the listen socket.
<i>XferID</i>	The transfer identifier to expect in the parallel data transfer headers.
<i>Offset</i>	The transfer offset at which this data begins.
<i>Buffer</i>	Pointer to the data buffer in which the data to be sent is located.
<i>BufSize</i>	The size of the passed buffer.
<i>BytesMoved</i>	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.

## Chapter 4. Supplemental Data Transfer Functions

---

EIO	An unexpected error occurred.
EPIPE	The connection closed while the header was being sent.

### See also

`pdata_rcv_hdr`, `pdata_send_hdr_and_data`, `mover_waitfor_requests_timeout`.

### Notes

This routine currently supports 32 simultaneous open connections.

4.1.3.14. 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 *SecTimeout* 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**

<i>SD</i>	File descriptor that refers to the listen socket.
<i>XferID</i>	The transfer identifier to expect in the parallel data transfer headers.
<i>Offset</i>	The transfer offset at which this data begins.
<i>Buffer</i>	Pointer to the data buffer in which the data to be sent is located.
<i>BufSize</i>	The size of the passed buffer.
<i>BytesMoved</i>	Place to return the number of bytes of data actually sent.
<i>SecTimeout</i>	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_rcv_hdr_timeout`, `pdata_send_hdr_and_data_timeout`, `mover_waitfor_requests`.

### Notes

This routine currently supports 32 simultaneous open connections.

## 4.1.4. Mover Protocol APIs

### 4.1.4.1. mvrprot\_rcv\_initmsg

#### Purpose

Receive a mover protocol initiator message.

#### Synopsis

```
#include "mvr_protocol.h"
```

```
long  
mvrprot_rcv_initmsg(  
    int          SockFD,      /* IN */  
    initiator_msg_t* InitPtr); /* OUT */
```

#### Description

The `mvrprot_rcv_initmsg` routine receives a mover protocol initiator message over the specified socket.

#### Parameters

<i>SockFD</i>	Descriptor that refers to the open connection over which the message is to be received.
<i>InitPtr</i>	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

<code>HPSS_ECONN</code>	The connection was closed while trying to receive the message.
<code>HPSS_EINVAL</code>	The verification of the checksum sent with the message failed.

#### See also

`mvrprot_send_initmsg`.

#### Notes

None.



#### 4.1.4.3. mvrprot\_rcv\_ipaddr

##### Purpose

Receive a mover protocol initiator TCP/IP address.

##### Synopsis

```
#include "mvr_protocol.h"
```

```
long
```

```
mvrprot_rcv_ipaddr(
```

```
    int
```

```
    SockFD,
```

```
    /* IN */
```

```
    initiator_ipaddr_t*
```

```
    InitIpPtr);
```

```
    /* OUT */
```

##### Description

The **mvrprot\_rcv\_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.

### 4.1.4.4. `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_rcv_ipaddr`.

#### Notes

None.

#### 4.1.4.5. mvrprot\_rcv\_ipi3addr

##### Purpose

Receive a mover protocol initiator IPI-3 address.

##### Synopsis

```
#include "mvr_protocol.h"
```

```
long
```

```
mvrprot_rcv_ipi3addr(  
    int
```

```
    initiator_ipi3addr_t  SockFD,          /* IN */
```

```
                        *InitIpi3Ptr); /* OUT */
```

##### Description

The `mvrprot_rcv_ipi3addr` routine receives a mover protocol IPI-3 address over the specified socket.

##### Parameters

*SockFD*

Descriptor that refers to the open connection over which the address is to be received.

*InitIpi3Ptr*

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_ipi3addr`.

##### Notes

None.

### 4.1.4.6. `mvrprot_send_ipi3addr`

#### Purpose

Send a mover protocol initiator IPI-3 address.

#### Synopsis

```
#include "mvr_protocol.h"
```

```
long
```

```
mvrprot_send_ipi3addr(
```

```
    int SockFD,          /* IN */
```

```
    initiator_ipi3addr_t *InitIpi3Ptr); /* IN */
```

#### Description

The `mvrprot_send_ipi3addr` routine sends a mover protocol IPI-3 address over the specified socket.

#### Parameters

*SockFD* Descriptor that refers to the open connection over which the address is to be sent.

*InitIpi3Ptr* 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_rcv_ipi3addr`.

#### Notes

None.

#### 4.1.4.7. mvrprot\_rcv\_shmaddr

##### Purpose

Receive a mover protocol initiator shared memory address.

##### Synopsis

```
#include "mvr_protocol.h"
```

```
long
```

```
mvrprot_rcv_shmaddr(
```

```
    int
```

```
    SockFD,          /* IN */
```

```
    initiator_shmaddr_t
```

```
    *InitShmPtr);  /* OUT */
```

##### Description

The **mvrprot\_rcv\_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.

### 4.1.4.8. `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_rcv_shmaddr`.

#### Notes

None.





## 4.1.5. Mover Protocol Data Structures

### 4.1.5.1. 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:

## Chapter 4. Supplemental Data Transfer Functions

---

NET_ADDRESS	TCP/IP transfer.
IPI_ADDRESS	IPI-3 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.

### 4.1.5.2. 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.

### 4.1.5.3. 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. 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.

### 4.1.5.4. Mover Protocol IPI-3 Address Structure - `initiator_ipi3addr_t`

#### **Description**

The Mover Protocol IPI-3 Address Structure is used to communicate IPI-3 addressing information.

#### **Format**

The Mover Protocol IPI-3 Address Structure has the following format:

```
typedef struct initiator_ipi3addr {
    u_signed64      Delimiter;
    unsigned32      Flags;
    ipiaddress_t    Ipi3Addr;
    char            SecurityTicket[MVRPROT_SEC_TICKET_LEN];
    u_signed64      CheckSum;
} initiator_ipi3addr_t;
```

#### Delimiter

Contains a distinct value to identify the message boundary.

#### Flags

Contains flags specific to the IPI-3 address. This field is currently unused.

#### Ipi3Addr

Contains the IPI-3 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.

### 4.1.5.5. 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.

## 4.1.6. Parallel Data Transfer Functions

### 4.1.6.1. 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.

### 4.1.6.2. `pdata_rcv_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_rcv_hdr_timeout(  
    int          SocketDescriptor,          /* IN */  
    pdata_hdr_t  *PdataHeaderPtr,          /* OUT */  
    int          SecTimeout);              /* IN */
```

#### Description

The `pdata_rcv_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 *SecTimeout* elapses between messages from the peer entity, the routine will return an error indication.

#### Parameters

<i>SocketDescriptor</i>	File descriptor referring to the open TCP/IP connection over which the header is to be received.
<i>PdataHeaderPtr</i>	Pointer to the area where the incoming header will be stored.
<i>SecTimeout</i>	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

<code>EBADF</code>	<i>SocketDescriptor</i> does not refer to an open connection.
<code>EINVAL</code>	Header checksum or delimiter is invalid.
<code>ETIMEDOUT</code>	The specified timeout expired between communication with the peer.

#### See also

`mover_socket_rcv_data_timeout`, `mover_waitfor_data_timeout`, `pdata_rcv_hdr`.

#### Notes

None.

### 4.1.6.3. 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

<i>SocketDescriptor</i>	File descriptor referring to the open TCP/IP connection over which the header is to be sent.
<i>PdataHeaderPtr</i>	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	<i>SocketDescriptor</i> 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.

### 4.1.6.4. `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 `SecTimeout` elapses between messages from the peer entity, the routine will return an error indication.

#### Parameters

<i>SocketDescriptor</i>	File descriptor referring to the open TCP/IP connection over which the header is to be sent.
<i>PdataHeaderPtr</i>	Pointer to the header to be sent.
<i>SecTimeout</i>	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	<i>SocketDescriptor</i> 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.

**4.1.6.5. 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**

<i>SocketDescriptor</i>	File descriptor referring to the open TCP/IP connection over which the header is to be sent.
<i>PdataHeaderPtr</i>	Pointer to the header to be sent.
<i>DataBuffer</i>	Pointer to the data to be sent.
<i>DataLength</i>	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	<i>SocketDescriptor</i> does not refer to an open connection.
EINVAL	The <i>DataLength</i> 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.

### 4.1.6.6. `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 *SecTimeout* elapses between messages from the peer entity, the routine will return an error indication.

#### Parameters

<i>SocketDescriptor</i>	File descriptor referring to the open TCP/IP connection over which the header is to be sent.
<i>PdataHeaderPtr</i>	Pointer to the header to be sent.
<i>DataBuffer</i>	Pointer to the data to be sent.
<i>DataLength</i>	Amount of data, in bytes, to be sent (excluding the header).
<i>SecTimeout</i>	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	<i>SocketDescriptor</i> does not refer to an open connection.
EINVAL	The <i>DataLength</i> 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.

### 4.1.6.7. `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 *SecTimeout* 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

<i>SocketDescriptor</i>	File descriptor referring to the open TCP/IP connection over which the header is to be sent.
<i>PdataHeaderPtr</i>	Pointer to the header to be sent.
<i>DataBuffer</i>	Pointer to the data to be sent.
<i>DataLength</i>	Amount of data, in bytes, to be sent (excluding the header).
<i>SecTimeout</i>	Number of seconds to wait if the connection becomes idle before returning; zero specifies infinite wait.
<i>WriteSize</i>	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

<code>EBADF</code>	<i>SocketDescriptor</i> does not refer to an open connection.
--------------------	---

EINVAL	The <i>DataLength</i> 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.

### 4.1.7. Parallel Data Transfer Data Definitions

#### 4.1.7.1. 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.

## 4.1.8. Network Options Functions

### 4.1.8.1. 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**

<i>NetAddr</i>	The IP address of the machine or network for which a match is requested in the HPSS network options table.
<i>RetTableEntryPtr</i>	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 value 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.

### 4.1.8.2. `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

<i>NetAddr</i>	The IP address of the machine or network for which a match is requested in the HPSS network options table.
<i>RetTableEntryPtr</i>	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.

## 4.1.9. Network Options Data Definitions

### 4.1.9.1. 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

## Chapter 4. Supplemental Data Transfer Functions

---

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 algorithm should not be disabled; a non-zero value indicates that the algorithm should be disabled.

### *Reserved?*

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

<i>ll1</i>	Specifies the first unsigned 64-bit integer.
<i>ll2</i>	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**

<i>ll1</i>	Specifies the result of the addition of the 2 unsigned 64-bit operands.
<i>ll2</i>	Specifies the first unsigned 64-bit integer addition operand.
<i>ll3</i>	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

<i>ll1</i>	Specifies the first unsigned 64-bit integer.
<i>ll2</i>	Specifies the second unsigned 64-bit integer.

#### Return values

Upon completion, an unsigned 64-bit integer is returned.

#### Error conditions

None.

#### See also

`not64m`, or `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**

<i>l1</i>	Specifies the unsigned 32-bit integer which will occupy the high order portion of the unsigned 64-bit integer.
<i>l2</i>	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    ll);    /* IN */
```

#### Description

The `cast64m` macro is called to cast an unsigned 32-bit integer into an unsigned 64-bit integer.

#### Parameters

*ll* 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          ll,          /* IN */  
    unsigned32         ll);        /* IN */
```

**Description**

The `div64m` macro is called to divide an unsigned 64-bit value by an unsigned 32-bit value.

**Parameters**

*ll* Specifies an unsigned 64-bit integer numerator.

*ll* 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

<i>ll1</i>	Specifies an unsigned 64-bit integer numerator.
<i>ll2</i>	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          ll,          /* IN */  
    unsigned32         ll);        /* 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**

*ll* Specifies an unsigned 64-bit integer numerator.

*ll* 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

<i>ll1</i>	Specifies an unsigned 64-bit integer numerator.
<i>ll2</i>	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

<i>ll1</i>	Specifies the first unsigned 64-bit integer.
<i>ll2</i>	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  $ll1 \geq ll2$ . 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

<i>ll1</i>	Specifies the first unsigned 64-bit integer.
<i>ll2</i>	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      ll);          /* 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**

*ll* 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**

<i>ll1</i>	Specifies the first unsigned 64-bit integer.
<i>ll2</i>	Specifies the second unsigned 64-bit integer.

#### **Return values**

1 is returned if  $ll1 \leq ll2$ . 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    ll);    /* 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**

*ll* 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

<i>ll1</i>	Specifies the first unsigned 64-bit integer.
<i>ll2</i>	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      ll,          /* IN */  
    unsigned32     ll);         /* 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**

*ll* Specifies the unsigned 64-bit integer to divide.

*ll* 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

<i>ll1</i>	Specifies the unsigned 64-bit integer to modulus.
<i>ll2</i>	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      ll,          /* IN */  
    unsigned32     ll);         /* IN */
```

**Description**

The `mul64m` macro is called to multiply an unsigned 64-bit integer value by an unsigned 32-bit integer value.

**Parameters**

*ll* Specifies the unsigned 64-bit integer multiplier.

*ll* 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    ll);    /* IN */
```

**Description**

The *not64m* macro is called to perform a bitwise NOT of an unsigned 64-bit integer value.

**Parameters**

*ll* 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

<i>ll1</i>	Specifies the first unsigned 64-bit integer.
<i>ll2</i>	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          ll,          /* IN */  
    unsigned32         ll);        /* IN */
```

**Description**

The *shl64m* macro is called to shift an unsigned 64-bit integer left by an unsigned 32-bit integer count.

**Parameters**

*ll* Specifies an unsigned 64-bit integer to be shifted.

*ll* 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    ll,          /* IN */  
    unsigned32    ll);        /* IN */
```

#### Description

The *shr64m* macro is called to shift an unsigned 64-bit integer right by an unsigned 32-bit integer count.

#### Parameters

*ll* Specifies an unsigned 64-bit integer to be shifted.

*ll* 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**

<i>ll1</i>	Specifies an unsigned 64-bit integer.
<i>ll2</i>	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

<i>ll1</i>	Specifies the result of the subtraction of the 2 unsigned 64-bit operands.
<i>ll2</i>	Specifies the first unsigned 64-bit integer subtraction operand.
<i>ll3</i>	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
- Data Definitions

### 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 han-

dle 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:

<code>MPI_ERR_ENOCONNECT</code>	Unable to connect to HPSS file system; HPSS may be down.
<code>MPI_ERR_EPERM</code>	The user's credentials could not be established; DCE may be down.
<code>MPI_ERR_ETIMEDOUT</code>	An operation timed out or received a communication error, and the operation could not be successfully retried; HPSS or DCE may be down.
<code>MPI_ERR_FILE_IN_USE</code>	The file is currently being used by another process.
<code>MPI_ERR_INTERN</code>	An internal MPI-IO error occurred.
<code>MPI_ERR_IO</code>	An internal HPSS error occurred.
<code>MPI_ERR_NOT_INITIALIZED</code>	MPI (MPI-IO) is not initialized.

The following errors can occur with any collective call:

<code>MPI_ERR_DUP_CLIENT</code>	Duplicate collective operation for a client process.
<code>MPI_ERR_EXPECTED_ATOMICITY</code>	Collective call to <code>MPI_File_set_atomicity</code> expected.
<code>MPI_ERR_EXPECTED_CLOSE</code>	Collective call to <code>MPI_File_close</code> expected.
<code>MPI_ERR_EXPECTED_OPEN</code>	Collective call to <code>MPI_File_open</code> expected.
<code>MPI_ERR_EXPECTED_SEEK</code>	Collective call to <code>MPI_File_seek_shared</code> expected.
<code>MPI_ERR_EXPECTED_SIZE</code>	Collective call to <code>MPI_File_set_size</code> or <code>MPI_File_preallocate</code> expected.
<code>MPI_ERR_EXPECTED_SYNC</code>	Collective call to <code>MPI_File_sync</code> expected.
<code>MPI_ERR_EXPECTED_VIEW</code>	Collective call to <code>MPI_File_set_view</code> expected.
<code>MPI_ERR_MEMBER</code>	Client is not a member of the communicator group for an open file.

See §6.1.1.6 (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 pro-

cesses, 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.

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., *filenames* 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

<i>comm</i>	MPI communicator specifying which processes will access the file.
<i>filename</i>	HPSS path name of the file (length must be less than or equal to HPSS_MAX_PATH_NAME).
<i>amode</i>	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.

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.

*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	<i>filename</i> or <i>fh</i> argument is NULL or <i>comm</i> argument is MPI_COMM_NULL.
MPI_ERR_ACCESS	Access permission is denied to the file or to a component of the <i>filename</i> path prefix.
MPI_ERR_AMODE	Invalid <i>amode</i> argument.
MPI_ERR_BAD_FILE	Invalid <i>filename</i> argument (empty or too long).
MPI_ERR_COMM	Invalid <i>comm</i> argument.
MPI_ERR_EISDIR	The specified <i>filename</i> 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 <i>filename</i> 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 <i>info</i> 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	<i>amode</i> or <i>filename</i> arguments or <i>info</i> 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

<code>MPI_ERR_ARG</code>	<i>fh</i> argument is NULL.
<code>MPI_ERR_ENOMEM</code>	Unable to allocate program space.
<code>MPI_ERR_FILE</code>	Invalid file handle.
<code>MPI_ERR_PENDING_RW</code>	There are uncompleted read or write operations pending for <i>fh</i> .

#### 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**

<i>filename</i>	HPSS path name of the file to delete.
<i>info</i>	File hints.

**Return Values**

On successful completion the function returns `MPI_SUCCESS`. Otherwise, it returns one of the error codes listed below.

**Error Conditions**

<code>MPI_ERR_ARG</code>	<i>filename</i> argument is NULL.
<code>MPI_ERR_ACCESS</code>	Access permission is denied to the file or to a component of the <i>filename</i> path prefix.
<code>MPI_ERR_BAD_FILE</code>	Invalid <i>filename</i> (empty or too long).
<code>MPI_ERR_EISDIR</code>	The specified <i>filename</i> is a directory.
<code>MPI_ERR_ENOMEM</code>	Unable to allocate program space.
<code>MPI_ERR_ENOTDIR</code>	A component of the <i>filename</i> path is not a directory.
<code>MPI_ERR_NO_SUCH_FILE</code>	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

<i>fh</i>	File handle.
<i>size</i>	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

<code>MPI_ERR_ENOMEM</code>	Unable to allocate program space.
<code>MPI_ERR_FILE</code>	Invalid file handle.
<code>MPI_ERR_NOT_SAME</code>	<i>size</i> arguments do not match across processes.
<code>MPI_ERR_OFFSET</code>	Negative <i>size</i> specified.
<code>MPI_ERR_PENDING_RW</code>	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**

<i>fh</i>	File handle.
<i>size</i>	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	<i>size</i> arguments do not match across processes.
MPI_ERR_OFFSET	Negative <i>size</i> 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**

<i>fh</i>	File handle.
<i>size</i>	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**

<code>MPI_ERR_ARG</code>	<i>size</i> argument is NULL.
<code>MPI_ERR_FILE</code>	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

<i>fh</i>	File handle.
<i>group</i>	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	<i>group</i> 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**

<i>fh</i>	File handle.
<i>amode</i>	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**

<code>MPI_ERR_ARG</code>	<i>amode</i> argument is NULL.
<code>MPI_ERR_FILE</code>	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) must appear in each process's *info* object.

#### Parameters

<i>fh</i>	File handle.
<i>info</i>	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	<i>info</i> 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**

<i>fh</i>	File handle.
<i>info_used</i>	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	<i>info_used</i> 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 *etypes*; file pointers point to the beginning of *etypes*. 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.

<i>disp</i>	Byte location of file offset zero.
<i>etype</i>	MPI_Datatype file unit.
<i>filetype</i>	MPI_Datatype that describes process's accessible <i>etypes</i> .
<i>datarep</i>	Data representation to be used for file data. See §6.1.3.
<i>info</i>	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	<i>datarep</i> argument is NULL or empty, or <i>etype</i> or <i>filetype</i> argument is MPI_DATATYPE_NULL.
MPI_ERR_DISPLACEMENT	Invalid <i>disp</i> .
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_ETYPE	Invalid <i>etype</i> .
MPI_ERR_EXTENT_CONVERSION	Error in user-defined extent function.
MPI_ERR_FILE	Invalid file handle.
MPI_ERR_FILETYPE	Invalid <i>filetype</i> .
MPI_ERR_HINTS	Invalid <i>info</i> .
MPI_ERR_NOT_SAME	<i>datarep</i> or <i>etype</i> arguments or <i>info</i> argument key values do not match across processes.
MPI_ERR_PENDING_RW	There are uncompleted read or write operations pending for <i>fh</i> .
MPI_ERR_UNSUPPORTED_DATAREP	Invalid <i>datarep</i> .

**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**

<i>fh</i>	File handle.
<i>disp</i>	The returned displacement.
<i>etype</i>	The returned <i>etype</i> .
<i>filetype</i>	The returned <i>filetype</i> .
<i>datarep</i>	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	<i>disp</i> , <i>etype</i> , <i>filetype</i> , or <i>datarep</i> 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 to 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**

<i>fh</i>	File handle.
<i>offset</i>	File offset in etypes at which to begin reading.
<i>buf</i>	User's memory buffer where data should be stored.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to read.
<i>datatype</i>	MPI datatype specifying data layout in <i>buf</i> .
<i>status</i>	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	<i>buf</i> or <i>status</i> argument is NULL.

MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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

<i>fh</i>	File handle.
<i>offset</i>	File offset in etypes at which to begin reading.
<i>buf</i>	User's memory buffer where data should be stored.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to read.
<i>datatype</i>	MPI datatype specifying data layout in <i>buf</i> .
<i>status</i>	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	<i>buf</i> or <i>status</i> argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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 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 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**

<i>fh</i>	File handle.
<i>offset</i>	File offset in etypes at which to begin writing.
<i>buf</i>	User's memory buffer where data should be read.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to write.
<i>datatype</i>	MPI datatype specifying how data is laid out in <i>buf</i> .
<i>status</i>	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	<i>buf</i> or <i>status</i> argument is NULL.

MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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**

<i>fh</i>	File handle.
<i>offset</i>	File offset in etypes at which to begin writing.
<i>buf</i>	User's memory buffer where data should be read.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to write.
<i>datatype</i>	MPI datatype specifying how data is laid out in <i>buf</i> .
<i>status</i>	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	<i>buf</i> or <i>status</i> argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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**

<i>fh</i>	File handle.
<i>offset</i>	File offset in etypes at which to begin reading.
<i>buf</i>	User's memory buffer where data should be stored.
<i>count</i>	Number of datatype-sized chunks of data to read.
<i>datatype</i>	MPI datatype specifying data layout in <i>buf</i> .
<i>request</i>	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	<i>buf</i> or <i>request</i> argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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\_irewrite\_at

**Purpose**

Write to a file at an explicit offset, noncollectively and without blocking.

**Synopsis**

```
#include <mpio.h>

int
MPI_File_irewrite_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\_irewrite\_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**

<i>fh</i>	File handle.
<i>offset</i>	File offset in etypes at which to begin writing.
<i>buf</i>	User's memory buffer where data should be read.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to write.
<i>datatype</i>	MPI datatype specifying how data is laid out in <i>buf</i> .
<i>request</i>	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	<i>buf</i> or <i>request</i> argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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**

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be stored.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to read.
<i>datatype</i>	MPI datatype specifying data layout in <i>buf</i> .
<i>status</i>	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	<i>buf</i> or <i>status</i> argument is NULL.

MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be stored.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to read.
<i>datatype</i>	MPI datatype specifying data layout in <i>buf</i> .
<i>status</i>	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	<i>buf</i> or <i>status</i> argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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**

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be read.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to write.
<i>datatype</i>	MPI datatype specifying how data is laid out in <i>buf</i> .
<i>status</i>	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	<i>buf</i> or <i>status</i> argument is NULL.

MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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\_ fwrite, 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**

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be read.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to write.
<i>datatype</i>	MPI datatype specifying how data is laid out in <i>buf</i> .
<i>status</i>	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	<i>buf</i> or <i>status</i> argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be stored.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to read.
<i>datatype</i>	MPI datatype specifying data layout in <i>buf</i> .
<i>request</i>	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	<i>buf</i> or <i>request</i> argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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\_iread

#### 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_iread(
    MPI_File      fh,          /* IN/OUT */
    void *        buf,        /* IN */
    int           count,      /* IN */
    MPI_Datatype  datatype,   /* IN */
    MPI_Request * request     /* OUT */
);
```

#### Description

**MPI\_File\_iread** 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

<i>fh</i>	File handle.
<i>buf</i>	User's buffer where data should be read.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to write.
<i>datatype</i>	MPI datatype specifying data layout in <i>buf</i> .
<i>request</i>	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	<i>buf</i> or <i>request</i> argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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

<i>fh</i>	File handle.
<i>offset</i>	Number of etypes to add or subtract from the position specified by <i>whence</i> .
<i>whence</i>	Flag specifying one of three locations in the file:  MPI_SEEK_SET - the pointer is set to <i>offset</i> .  MPI_SEEK_CUR - the pointer is set to the current pointer position plus <i>offset</i> .  MPI_SEEK_END - the pointer is set to the end of file plus <i>offset</i> .

#### 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 <i>offset</i> argument.
MPI_ERR_UNSUPPORTED_OPERATION	Operation not allowed with MPI_MODE_SEQUENTIAL.
MPI_ERR_WHENCE	Invalid <i>whence</i> 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

<i>fh</i>	File handle.
<i>offset</i>	Returned <i>offset</i> 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	<i>offset</i> 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

<i>fh</i>	File handle.
<i>offset</i>	Offset in etypes to convert.
<i>disp</i>	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	<i>disp</i> 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**

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be stored.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to read.
<i>datatype</i>	MPI datatype specifying data layout in <i>buf</i> .
<i>status</i>	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	<i>buf</i> or <i>status</i> argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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**

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be read.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to write.
<i>datatype</i>	MPI datatype specifying data layout in <i>buf</i> .
<i>status</i>	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	<i>buf</i> or <i>status</i> argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be stored.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to read.
<i>datatype</i>	MPI datatype specifying data layout in <i>buf</i> .
<i>request</i>	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

<code>MPI_ERR_ACCESS</code>	Access permission is denied to the file.
<code>MPI_ERR_ARG</code>	<i>buf</i> or <i>request</i> argument is NULL.
<code>MPI_ERR_BUFTYPE</code>	Invalid buffer <i>datatype</i> .
<code>MPI_ERR_COUNT</code>	Invalid <i>count</i> argument.
<code>MPI_ERR_ENOMEM</code>	Unable to allocate program space.
<code>MPI_ERR_EXTENT_CONVERSION</code>	Error in user-defined extent function.
<code>MPI_ERR_FILE</code>	Invalid file handle.
<code>MPI_ERR_NOT_SAME_VIEW</code>	Shared file pointer operation requires all processes have the same view.
<code>MPI_ERR_READ_CONVERSION</code>	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\_ fwrite\_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_ fwrite_shared(
    MPI_File      fh,          /* IN/OUT */
    void *        buf,        /* IN */
    int           count,      /* IN */
    MPI_Datatype  datatype,   /* IN */
    MPI_Request * request     /* OUT */
);
```

#### Description

**MPI\_File\_ fwrite\_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

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be read.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to write.
<i>datatype</i>	MPI datatype specifying data layout in <i>buf</i> .
<i>request</i>	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

<code>MPI_ERR_ACCESS</code>	Access permission is denied to the file.
<code>MPI_ERR_ARG</code>	<i>buf</i> or <i>request</i> argument is NULL.
<code>MPI_ERR_BUFTYPE</code>	Invalid buffer <i>datatype</i> .
<code>MPI_ERR_COUNT</code>	Invalid <i>count</i> argument.
<code>MPI_ERR_ENOMEM</code>	Unable to allocate program space.
<code>MPI_ERR_EXTENT_CONVERSION</code>	Error in user-defined extent function.
<code>MPI_ERR_FILE</code>	Invalid file handle.
<code>MPI_ERR_NO_SPACE</code>	Insufficient free space on the HPSS file system to create, write, or preallocate a file.
<code>MPI_ERR_NOT_SAME_VIEW</code>	Shared file pointer operation requires all processes have the same view.
<code>MPI_ERR_TOO_FRAGMENTED</code>	Write operation resulted in excessive file fragmentation.
<code>MPI_ERR_WRITE_CONVERSION</code>	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**

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be stored.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to read.
<i>datatype</i>	MPI datatype specifying data layout in <i>buf</i> .
<i>status</i>	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	<i>buf</i> or <i>status</i> argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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**

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be read.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to write.
<i>datatype</i>	MPI datatype specifying data layout in <i>buf</i> .
<i>status</i>	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	<i>buf</i> or <i>status</i> argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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**

<i>fh</i>	File handle.
<i>offset</i>	Number of etypes to add or subtract from the location specified by <i>whence</i> .
<i>whence</i>	Flag specifying one of three locations in the file:  MPI_SEEK_SET - the pointer is set to <i>offset</i> .  MPI_SEEK_CUR - the pointer is set to the current pointer position plus <i>offset</i> .  MPI_SEEK_END - the pointer is set to the end of file plus <i>offset</i> .

**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	<i>offset</i> and <i>whence</i> 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 <i>offset</i> argument.
MPI_ERR_WHENCE	Invalid <i>whence</i> 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

<i>fh</i>	File handle.
<i>offset</i>	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	<i>offset</i> 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

Initiate 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

<i>fh</i>	File handle.
<i>offset</i>	File offset in etypes at which to begin reading.
<i>buf</i>	User's memory buffer where data should be stored.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to read.
<i>datatype</i>	MPI datatype specifying data layout in <i>buf</i> .

#### 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	<i>buf</i> argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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 <i>fh</i> .
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**

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be stored.
<i>status</i>	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	<i>buf</i> or <i>status</i> 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 <i>fh</i> .
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

<i>fh</i>	File handle.
<i>offset</i>	File offset in etypes at which to begin writing.
<i>buf</i>	User's memory buffer where data should be read.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to write.
<i>datatype</i>	MPI datatype specifying data layout in <i>buf</i> .

#### 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	<i>buf</i> argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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 <i>fh</i> .
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**

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be read.
<i>status</i>	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	<i>buf</i> or <i>status</i> 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 <i>fh</i> .
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

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be stored.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to read.
<i>datatype</i>	MPI datatype specifying data layout in <i>buf</i> .

#### 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	<i>buf</i> argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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 <i>fh</i> .
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

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be stored.
<i>status</i>	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	<i>buf</i> or <i>status</i> 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 <i>fh</i> .
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

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be read.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to write.
<i>datatype</i>	MPI datatype specifying data layout in <i>buf</i> .

#### 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	<i>buf</i> argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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 <i>fh</i> .
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

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be read.
<i>status</i>	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	<i>buf</i> or <i>status</i> 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 <i>fh</i> .
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

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be stored.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to read.
<i>datatype</i>	MPI datatype specifying data layout in <i>buf</i> .

#### 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	<i>buf</i> argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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 <i>fh</i> .

**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

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be read.
<i>status</i>	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	<i>buf</i> or <i>status</i> 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 <i>fh</i> .
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

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be read.
<i>count</i>	Number of <i>datatype</i> -sized chunks of data to write.
<i>datatype</i>	MPI datatype specifying data layout in <i>buf</i> .

#### 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	<i>buf</i> argument is NULL.
MPI_ERR_BUFTYPE	Invalid buffer <i>datatype</i> .
MPI_ERR_COUNT	Invalid <i>count</i> 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 <i>fh</i> .

### 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**

<i>fh</i>	File handle.
<i>buf</i>	User's memory buffer where data should be read.
<i>status</i>	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	<i>buf</i> or <i>status</i> 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 <i>fh</i> .
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:

<code>MPI_PACKED</code>	1
<code>MPI_BYTE</code>	1
<code>MPI_CHAR</code>	1
<code>MPI_UNSIGNED_CHAR</code>	1
<code>MPI_SIGNED_CHAR</code>	1
<code>MPI_WCHAR</code>	2
<code>MPI_SHORT</code>	2
<code>MPI_UNSIGNED_SHORT</code>	2
<code>MPI_INT</code>	4
<code>MPI_UNSIGNED</code>	4
<code>MPI_LONG</code>	4
<code>MPI_UNSIGNED_LONG</code>	4
<code>MPI_FLOAT</code>	4
<code>MPI_DOUBLE</code>	8
<code>MPI_LONG_DOUBLE</code>	16
<code>MPI_CHARACTER</code>	1
<code>MPI_LOGICAL</code>	4
<code>MPI_INTEGER</code>	4
<code>MPI_REAL</code>	4

MPI_DOUBLE_PRECISION	8
MPI_COMPLEX	2*4
MPI_DOUBLE_COMPLEX	2*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**

<i>fh</i>	File handle.
<i>datatype</i>	MPI datatype for which to determine extent.
<i>extent</i>	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	<i>extent</i> argument is NULL or <i>datatype</i> 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 identifier.

#### 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

<i>datarep</i>	String literal to be used to identify the data representation.
<i>read_conversion_fn</i>	Callback function to convert data read from a file to the native representation.
<i>write_conversion_fn</i>	Callback function to convert data written to a file from the native representation.
<i>dtype_file_extent_fn</i>	Callback function to determine the extent of a predefined type in <i>datarep</i> .

#### Return Values

On successful completion the function returns `MPI_SUCCESS`. Otherwise, it returns one of the error codes listed below.

#### Error Conditions

<code>MPI_ERR_ARG</code>	Invalid <i>datarep</i> argument.
<code>MPI_ERR_DUP_DATAREP</code>	<i>datarep</i> duplicates a registered or predefined data representation identifier.
<code>MPI_ERR_ENOMEM</code>	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 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, non-atomic mode is set.

**Parameters**

<i>fh</i>	File handle.
<i>flag</i>	Specifies how the system should handle overlapping write requests from different processes. Possible values are:  <b>true</b> (nonzero) and <b>false</b> (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	<i>flag</i> 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

<i>fh</i>	File handle.
<i>flag</i>	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

<code>MPI_ERR_ARG</code>	<i>flag</i> argument is NULL.
<code>MPI_ERR_FILE</code>	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**

<b>MPI_ERR_ENOMEM</b>	Unable to allocate program space.
<b>MPI_ERR_FILE</b>	Invalid file handle.
<b>MPI_ERR_PENDING_RW</b>	There are uncompleted read or write operations pending for <i>fh</i> .

**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:

<code>MPI_ERR_FILE</code>	Invalid file handle.
<code>MPI_ERR_NOT_SAME</code>	Collective argument not the same on all processes, or collective routine called in a different order by different processes.
<code>MPI_ERR_AMODE</code>	Error related to the <i>amode</i> passed to <code>MPI_File_open</code> .
<code>MPI_ERR_UNSUPPORTED_DATAREP</code>	Unsupported <i>datarep</i> passed to <code>MPI_File_set_view</code> .
<code>MPI_ERR_UNSUPPORTED_OPERATION</code>	Unsupported operation, such as seeking on a file which supports sequential access only.
<code>MPI_ERR_NO_SUCH_FILE</code>	File does not exist.
<code>MPI_ERR_FILE_EXISTS</code>	File exists.
<code>MPI_ERR_BAD_FILE</code>	Invalid file name.
<code>MPI_ERR_ACCESS</code>	Permission denied.
<code>MPI_ERR_NO_SPACE</code>	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 <i>whence</i> 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_atomicsity 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

<i>function</i>	User-defined error handling function.
<i>errhandler</i>	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	<i>function</i> or <i>errhandler</i> 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**

<i>fh</i>	File handle.
<i>errhandler</i>	File error handler to associate with <i>fh</i> .

**Return Values**

On successful completion the function returns `MPI_SUCCESS`. Otherwise, it returns one of the error codes listed below.

**Error Conditions**

<code>MPI_ERR_ARG</code>	Invalid <i>errhandler</i> argument.
<code>MPI_ERR_FILE</code>	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

<i>fh</i>	File handle.
<i>errhandler</i>	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	<i>errhandler</i> 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**

<i>fh</i>	File handle.
<i>errorcode</i>	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**

<code>MPI_ERR_ARG</code>	<i>fh</i> 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 is not required to support all the reserved keys, and it is allowed to add others. The following keys are currently supported in MPI-IO:

<b>“access_style”</b>	Specifies the manner in which a file will be accessed until the file is closed. The value is one of the following: <b>“read_once”</b> , <b>“write_once”</b> , <b>“read_mostly”</b> , <b>“write_mostly”</b> , <b>“sequential”</b> , <b>“reverse_sequential”</b> , <b>“random”</b> , or <b>“collective_only”</b> . Currently, only <b>“collective_only”</b> is used to optimize accesses. All processes must specify the same value for this key.
<b>“conversion_buffer_size”</b>	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.
<b>“filename”</b>	Specifies the file name used when a file was opened. This key is only supported in <b>MPI_File_get_info</b> .
<b>“file_perm”</b>	Specifies the file permissions to use for file creation, which is only useful when passed to <code>MPI_File_open</code> with an <i>amode</i> of <code>MPI_MODE_CREATE</code> . The value must contain a valid HPSS permission specification for the Mode argument to <b>hpss_Chmod</b> (e.g., “0744”). All processes must specify the same value for this key.
<b>“nb_proc”</b>	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.
<b>“striping_factor”</b>	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.
<b>“striping_unit”</b>	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.

<b>“hpss_cos”</b>	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.
<b>“hpss_sclasstype”</b>	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.
<b>“hpss_max_file_size”</b>	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.
<b>“hpss_min_file_size”</b>	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.
<b>“hpss_access_size”</b>	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`.



### 6.1.6.2. 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

<i>info</i>	MPI_Info object to update.
<i>key</i>	Key to add or update.
<i>value</i>	Value for the associated <i>key</i> .

#### 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	<i>info</i> is MPI_INFO_NULL, or <i>key</i> or <i>value</i> is NULL.
MPI_ERR_ENOMEM	Unable to allocate program space.
MPI_ERR_INFO_KEY	<i>key</i> length exceeds MPI_MAX_INFO_KEY.
MPI_ERR_INFO_VALUE	<i>value</i> 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

<i>info</i>	MPI_Info object to update.
<i>key</i>	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	<i>key</i> argument is NULL or <i>info</i> argument is MPI_INFO_NULL.
MPI_ERR_INFO_KEY	<i>key</i> length exceeds MPI_MAX_INFO_KEY.
MPI_ERR_INFO_NOKEY	<i>key</i> is not defined in <i>info</i> .

#### 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

<i>info</i>	MPI_Info object to examine.
<i>key</i>	Key to retrieve.
<i>valuelen</i>	Maximum number of characters available in <i>value</i> .
<i>value</i>	Returned value associated with <i>key</i> .
<i>flag</i>	Indicator of whether or not <i>key</i> is defined in <i>info</i> .

##### 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	<i>key</i> , <i>value</i> or <i>flag</i> argument is NULL or <i>info</i> argument is MPI_INFO_NULL.
MPI_ERR_INFO_KEY	<i>key</i> 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**

<i>info</i>	MPI_Info object to examine.
<i>key</i>	Key to retrieve.
<i>valuelen</i>	Returned length of the associated value for <i>key</i> .
<i>flag</i>	Returned indicator of whether <i>key</i> was defined in <i>info</i> .

**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	<i>key</i> , <i>valuelen</i> , or <i>flag</i> argument is NULL or <i>info</i> argument is MPI_INFO_NULL.
MPI_ERR_INFO_KEY	<i>key</i> 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**

<i>info</i>	MPI_Info object to examine.
<i>nkeys</i>	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	<i>nkeys</i> argument is NULL or <i>info</i> 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

<i>info</i>	MPI_Info object to examine.
<i>n</i>	Index of key to retrieve.
<i>key</i>	Returned <i>key</i> .

#### 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	<i>info</i> is MPI_INFO_NULL or <i>key</i> is NULL.
MPI_ERR_INFO_NOKEY	<i>n</i> is invalid index for <i>info</i> .

#### 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**

<i>info</i>	MPI_Info object to duplicate.
<i>newinfo</i>	Returned duplicate of <i>info</i> 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	<i>newinfo</i> argument is NULL.
MPI_ERR_ENOMEM	Unable to allocate program space.

**See Also**

**MPI\_Info\_free**, **MPI\_Info\_create**.

**Notes**

Noncollective.



## 6.2. 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.

### 6.2.1. 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.

### 6.2.2. 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;
```



## 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/usr/lpp/hpss/include -I/usr/lpp/encina/include

LIB_DIRS   = -L/usr/lpp/hpss/lib \
            -L/usr/lpp/encina/lib

LIBS       = $(LIB_DIRS) \
            -ldce \
            -lEncSfs \
            -lEncina \
            -lhpss \
            /usr/lpp/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, reysize;
    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          reysize64, rec_cnt, i;
```

```

u_signed64                word64_num;

/* Initialize local variables */

rc, rc2 = 0;
word64_num = cast64m( 1 );
flags = O_RDONLY;
reclsize = 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
 */

```

```
resize64 = cast64m( resize );

for (;;) {

    if ((rc = hpss_Read( fd, buf_ptr, resize )) < 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, IPI-3, 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|ipi]* [-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|ipi
*               By default, TCP and shared memory transfers are enabled
*               (as well as IPI-3 if the program is compiled for IPI
*               support). The -p option can be repeatedly used to

```

## Appendix A - Programming Examples

---

\* specify which transfer options to make available. For  
\* example, to restrict transfers to TCP only, use  
\* "-p tcp". To make both TCP and IPI-3 available as  
\* options (but not shared memory), use "-p tcp -p ipi".  
\*  
\* -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, IPI-3, 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.  
\*  
\* This program must be compiled with the -DIPI3\_SUPPORT option in order  
\* to support IPI-3 data transfers and it must be executed on a machine  
\* that support HIPPI and IPI-3.  
\*

```

*      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_SBMX     (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_setopts (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 {

```

## Appendix A - Programming Examples

---

```
int      active;          /* Whether thread/connection is active */
pthread_t threadId;      /* Id of the transfer thread */
int      controlSocketFd; /* Control socket descriptor */
int      ipiFd;          /* IPI-3 transfer file 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;
} tcpghost_t;

unsigned32 NumHosts;          /* -h argument counter */
tcpghost_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], "ipi")) {
            src.Flags |= XFEROPT_IPI3;
        }
        else if (!strcmp(argv[0], "shm")) {
            src.Flags |= XFEROPT_SHMEM;
        }
        else {
            printf ("Invalid transfer protocol - use tcp, shm, or ipi\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|ipil]* [-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|ipi\n");
printf("\tBy default, TCP and shared memory transfers are enabled (as ");
printf("well as\n");
printf("\tIPI-3 if the program is compiled for IPI support).  The -p ");
printf("option can\n");
printf("\tbe repeatedly used to specify which transfer options to ");
printf("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 (use IPI-3
* only if it was compiled in)
*/
if (!src.Flags) {
    src.Flags = XFEROPT_IP | XFEROPT_SHMEM;
```

```

#if defined(IPI3_SUPPORT)
    src.Flags |= XFEROPT_IPI3;
#endif
}
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_family;
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;
}

/*=====

```

## Appendix A - Programming Examples

---

```
* 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 */

#ifdef IPI3_SUPPORT
    int                ipi3Descriptor, ipi3ThreadId;
    IPI3_INTERFACE_STRUCT ipi3Addr;
    initiator_ipi3addr_t ipi3Addr;
#endif

    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.

```

## Appendix A - Programming Examples

---

```
* 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_initmsg 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" : "IPI");
}

/* 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 offered 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;

#ifdef IPI3_SUPPORT

case IPI_ADDRESS:

    /* Allocate the memory buffer the first time through
     */
    if (!buffer) {
        buffer = malloc (BufferSize);
        if (!buffer) {
```

```

        perror ("malloc");
        TransferStatus = errno;
        continue;
    }

    /* Open the IPI3 device
    */
    Connections[index].ipiFd = ipi3_data3_open (&ipi3Addr);

    if (Connections[index].ipiFd < 0) {
        printf ("ipi3_data3_open returned %d\n", Connections[index].ipiFd);
        TransferStatus = Connections[index].ipiFd;
        Connections[index].ipiFd = 0;
        continue;
    }
}

status = ipi3_data3_write (Connections[index].ipiFd, &ipi3ThreadId,
                          low32m (initReply.Length), buffer);

if (status < 0) {
    hpss_error ("ipi3_data3_write", status);
    TransferStatus = status;
    continue;
}

ipiAddr.Flags = 0;
ipiAddr.Ipi3Addr.IPI3TransferID      = ipi3ThreadId;
ipiAddr.Ipi3Addr.IPI3Offset          = cast64m(0);
ipiAddr.Ipi3Addr.IPI3Addr.Interface = ipi3Addr.interface;

strcpy ((char*)ipiAddr.Ipi3Addr.IPI3Addr.Name,
        ipi3Addr.name);

status = mvrprot_send_ipi3addr (Connections[index].controlSocketFd,
                                &ipiAddr);

if (status != HPSS_E_NOERROR) {
    hpss_error ("mvrprot_send_ipi3addr", status);
    ipi3_data3_cancel (Connections[index].ipiFd);
    TransferStatus = status;
    continue;
}

status = ipi3_data3_complete (Connections[index].ipiFd, -1);

if (status < 0) {
    hpss_error ("ipi3_data3_complete", status);
    TransferStatus = status;
    continue;
}

```

```
        break;
#endif

    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;
    case IPI_ADDRESS:
        printf ("IPI\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;

#ifdef IPI3_SUPPORT

case IPI_ADDRESS:

    (void) ipi3_data3_close (Connections[index].ipiFd);

    /* Free the buffer if it was allocated
    */
    if (buffer) (void) free(buffer);
    break;

#endif

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_KEEPAALIVE, &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, IPI-3, 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|ipi]* [-o] [-x sbmax] <path>
 *
 *   -v           Prints verbose output
 *   -c           Prints low-level control information
 *   -t           Print transfer rate in whole bytes per second
 *
 */

```

## Appendix A - Programming Examples

---

```
*      -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|ipi
*          By default, TCP and shared memory transfers are enabled
*          (as well as IPI-3 if the program is compiled for IPI
*          support).  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".  To make both TCP and IPI-3 available as
*          options (but not shared memory), use "-p tcp -p ipi".
*
*      -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, IPI-3, 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).
*
*      This program must be compiled with the -DIPI3_SUPPORT option in order
*      to support IPI-3 data transfers and it must be executed on a machine
*      that support HIPPI and IPI-3.
*
*      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"

```

## Appendix A - Programming Examples

---

```
/* Define program default values */

#define DEFAULT_BUFFER_SIZE      (4*1048576)
#define DEFAULT_MAX_CONNECTIONS  32
#define DEFAULT_SOCKET_SBM_MAX  (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_setopts (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      ipiFd;           /* IPI-3 transfer file 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;

```

## Appendix A - Programming Examples

---

```
/* 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 shmids *)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], "ipi")) {
            sink.Flags |= XFEROPT_IPI3;
        }
    }
}
```

```

    }
    else if (!strcmp(argv[0], "shm")) {
        sink.Flags |= XFEROPT_SHMEM;
    }
    else {
        printf ("Invalid transfer protocol - use tcp, shm, or ipi\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|ipi]* [-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|ipi\n");
printf("\tBy default, TCP and shared memory transfers are enabled ");
printf("as well as\n");
printf("\tIPI-3 if the program is compiled for IPI support).  The ");
printf("-p option can\n");
printf("\tbe repeatedly used to specify which transfer options to ");
printf("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 (use IPI-3
* only if it was compiled in)
*/
if (!sink.Flags) {
    sink.Flags = XFEROPT_IP | XFEROPT_SHMEM;
```

```

#if defined(IPI3_SUPPORT)
    sink.Flags |= XFEROPT_IPI3;
#endif
}
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) {

```

## Appendix A - Programming Examples

---

```
    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 shmids *)NULL);
        }
    }

    exit (status);
}

```

## Appendix A - Programming Examples

---

```
/*=====
 * 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 */

#ifdef IPI3_SUPPORT
    int                ipi3Descriptor, ipi3ThreadId;
    IPI3_INTERFACE_STRUCT ipi3Addr;
    initiator_ipi3addr_t ipiAddr;
#endif

    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_initmsg 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_rcv_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" : "IPI");
}

/* 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 offered 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_rcv_data (transferSocketFd, cast64m (RequestId),
                                initMessage.Offset, buffer,
                                low32m (initReply.Length),
                                &bytesReceived, 1);

if (status <= 0) {
    hpss_error ("mover_socket_rcv_data", status);
    TransferStatus = status;
}
break;

#ifdef IPI3_SUPPORT

case IPI_ADDRESS:

    /* Allocate the memory buffer the first time through
    */
    if (!buffer) {
        buffer = malloc (BufferSize);
        if (!buffer) {
            perror ("malloc");
            TransferStatus = errno;
            continue;
        }

        /* Open the IPI3 device
        */
        Connections[index].ipiFd = ipi3_data3_open (&ipi3Addr);

        if (Connections[index].ipiFd < 0) {
            printf ("ipi3_data3_open returned %d\n", Connections[index].ipiFd);
            TransferStatus = Connections[index].ipiFd;
            Connections[index].ipiFd = 0;
        }
    }
}

```

```
        continue;
    }
}

status = ipi3_data3_read (Connections[index].ipiFd, &ipi3ThreadId,
                        low32m (initReply.Length), buffer);

if (status < 0) {
    hpss_error ("ipi3_data3_read", status);
    TransferStatus = status;
    continue;
}

ipiAddr.Flags = 0;
ipiAddr.Ipi3Addr.IPI3TransferID      = ipi3ThreadId;
ipiAddr.Ipi3Addr.IPI3Offset          = cast64m(0);
ipiAddr.Ipi3Addr.IPI3Addr.Interface = ipi3Addr.interface;

strcpy ((char*)ipiAddr.Ipi3Addr.IPI3Addr.Name,
        ipi3Addr.name);

status = mvrprot_send_ipi3addr (Connections[index].controlSocketFd,
                               &ipiAddr);

if (status != HPSS_E_NOERROR) {
    hpss_error ("mvrprot_send_ipi3addr", status);
    ipi3_data3_cancel (Connections[index].ipiFd);
    TransferStatus = status;
    continue;
}

status = ipi3_data3_complete (Connections[index].ipiFd, -1);

if (status < 0) {
    hpss_error ("ipi3_data3_complete", status);
    TransferStatus = status;
    continue;
}
break;
#endif

default:
    break;

} /* end switch */

/* Get a transfer completion message from the Mover
*/
status = mvrprot_rcv_compmsg (Connections[index].controlSocketFd,
                             &completionMessage);
```

```

if (status != HPSS_E_NOERROR) {
    hpss_error ("mvrprot_recv_compmg", 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;
    case IPI_ADDRESS:
        printf ("IPI\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;
    }
}

```

## Appendix A - Programming Examples

---

```
    }
    break;

#if defined(IPI3_SUPPORT)

    case IPI_ADDRESS:

        (void) ipi3_data3_close (Connections[index].ipiFd);

        /* Free the buffer if it was allocated
         */
        if (buffer) (void) free(buffer);
        break;

#endif

    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_setopts - 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_setopts (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_KEEPAALIVE, &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:

## Appendix A - Programming Examples

---

```
    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 recursively 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 DIRECTORY_OBJECT:

    printf (" (DIRECTORY)\n");
    list_directory (path, indent_level+1);
    break;

case FILE_OBJECT:
    printf (" (FILE, COS %lu, ", attrs->ClassOfService);
    print_u_signed64 (attrs->FileSize);
    printf (" bytes)\n");
    break;

case 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 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)

```

## Appendix A - Programming Examples

---

```
{
    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

```

## Appendix A - Programming Examples

---

```
*      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;
    case API_TRANSFER_IPI3:
        printf("IPI3\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;
}

```

## Appendix A - Programming Examples

---

```
/*=====
 * 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 IPI3 and host name\n");

        strcpy (config.HostName, "hpss-f");
        config.TransferType = API_TRANSFER_IPI3;

        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

#### 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)

```

## Appendix B - Makefile Example

---

```
CFLAGS=$(PLATFORM_CFLAGS) $(INCLUDES) $(PLATFORM_INCLUDES)

LDLFLAGS=$(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
```

### Include Makefile for AIX 4.2 and 4.3:

```
#####
#   aix42-43.mk - AIX 4.2 and 4.3 systems
#####

PLATFORM_ROOT_DIR= /usr/lpp/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
```

---

**Include Makefile for Non-DCE AIX 4.2:**

```
#=====
#   aix42-nodce.mk - Non-DCE AIX 4.2 systems
#=====

PLATFORM_ROOT_DIR= /usr/lpp/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
```

**Include Makefile for Non-DCE AIX 4.3:**

```
#=====
#   aix43-nodce.mk - Non-DCE AIX 4.3 systems
#=====

PLATFORM_ROOT_DIR= /usr/lpp/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
```

**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 \
```

## Appendix B - Makefile Example

---

```
-Dhpss_idl_types_v0_0_included -DTHREADS_ENABLED \  
-DTHREAD_SAFE -D_XOPEN_SOURCE -D_XOPEN_SOURCE_EXTENDED=1 \  
-D__EXTENSIONS__
```

### Include Makefile for Solaris 2.6:

```
#####  
# solaris26.mk - Solaris 2.6 systems  
#####  
  
PLATFORM_ROOT_DIR= /usr/lpp/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__
```

### Include Makefile for Non-DCE IRIX 6.4:

```
#####  
# irix64-nodce.mk - Non-DCE IRIX 6.4 systems  
#####  
  
PLATFORM_ROOT_DIR= /usr/lpp/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 or IPI-3) 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 responsi-

## Appendix C - Notes

---

ble, 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:

## Appendix C - Notes

---

```
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, IPI-3, 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
IPI	Intelligent Peripheral Interface
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 Protocol
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



## 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*, November 1999.
4. *HPSS Programmer's Reference Guide, Volume 2*, November 1999.
5. *HPSS System Administration Guide*, November 1999.
6. *HPSS User's Guide*, November 1999.
7. *Institute of Electrical and Electronics Engineers (IEEE) Mass Storage System Reference Model (MSSRM) (Version 5)*.
8. *Network File System Specification, RFC-1094*, DDN Network Information Center, SRI International, Menlo Park, Ca.
9. *OSF DCE User's Guide and Reference*, Prentice Hall, Englewood Cliffs, N. J.
10. *P1003.1a Draft Revision to Information Technology - POSIX Part 1: System Application Program Interface (API) [C Language]*.



# Index

## Numerics

### 64-bit Arithmetic Library

- add64\_3m, 5-3
- add64m, 1-9, 5-2
- and64m, 1-9, 5-4
- bld64m, 1-9, 5-5
- cast64m, 1-9, 5-6
- Components, 1-9
- Constraints, 1-10
- div\_2xcl64m, 1-9
- div\_cl64m, 1-9, 5-9, 5-10
- div2x64m, 1-9, 5-8
- div64m, 1-9, 5-7
- eq64m, 1-9, 5-12
- eqz64m, 1-9, 5-11
- ge64m, 1-10, 5-13
- gt64m, 1-10, 5-14
- high32m, 1-9, 5-15
- le64m, 1-9, 5-16
- Libraries, 1-10
- low32m, 1-9, 5-17
- lt64m, 1-9, 5-18
- mod2x64m, 1-10, 5-20
- mod64m, 1-10, 5-19
- mul64m, 1-10, 5-21
- neq64m, 1-10
- neqz64m, 1-10, 5-22
- not64m, 1-10, 5-23
- or64m, 1-10, 5-24
- Purpose, 1-9
- shl64m, 1-10, 5-25
- shr64m, 1-10, 5-26
- sub64\_3m, 5-28
- sub64m, 1-10, 5-27

## A

- acct\_rec\_t, 2-154
- add64\_3m, 5-3
- add64m, 1-9, 5-2

address\_t 3-8  
and64m, 1-9, 5-4  
api\_config\_t, 2-155  
B  
bf\_attrib\_md\_t, 2-147  
bf\_attrib\_t, 2-146  
bf\_vv\_attrib\_t, 2-152  
bf\_xattrib\_t, 2-146  
bfs\_owner\_rec\_t, 2-150  
bfs\_sc\_attrib\_t, 2-151  
bfs\_stats\_t, 2-154  
bld64m, 1-9, 5-5  
C  
cast64m, 1-9, 5-6  
class of service, 1-14  
Client API  
    components, 1-1  
    constraints, 1-3  
    environment variables (also see Environment Variables), 1-3  
    libraries (also see Libraries), 1-3  
Client API Functions 2-1  
Client API, 1-16  
collective functions, 6-1  
completion\_msg\_t, 4-51  
COS, 1-14  
D  
DCE User Accounts, 1-16  
dce\_login, 1-16  
directory operations, A-65  
displacement, 6-4  
div\_2xcl64m, 1-9, 5-10  
div\_cl64m, 1-9, 5-9  
div2x64m, 1-9, 5-8  
div64m, 1-9, 5-7  
E  
e64m, 5-16  
end of file, 6-4  
Environment Variables  
    HPSS\_BUSY\_DELAY, 1-5, 1-14

- HPSS\_BUSYRETRIES, 1-5
- HPSS\_DEBUG, 1-4
- HPSS\_DEBUGPATH, 1-4
- HPSS\_DESC\_NAME, 1-4
- HPSS\_DMAP\_WRITE\_UPDATES, 1-5
- HPSS\_HOSTNAME, 1-4, 1-8
- HPSS\_KTAB\_PATH, 1-4
- HPSS\_LS\_NAME, 1-4, 1-14
- HPSS\_MAX\_CONN, 1-4
- HPSS\_NUMRETRIES, 1-5
- HPSS\_PRINCIPAL, 1-4
- HPSS\_REGISTRY\_SITE\_NAME, 1-5
- HPSS\_RETRY\_STAGE\_INP, 1-5
- HPSS\_REUSE\_CONNECTIONS, 1-5
- HPSS\_SERVER\_NAME, 1-4
- HPSS\_TOTAL\_DELAY, 1-5
- HPSS\_TRANSFER\_TYPE, 1-4
- HPSS\_USE\_PORT\_RANGE, 1-5
- MPIO\_DEBUG, 1-14
- MPIO\_KEYTAB\_PATH, 1-14
- MPIO\_LOGIN\_NAME, 1-14
- eq64m, 1-9, 5-12
- eqz64m, 1-9, 5-11
- etype, 6-4
- F
- file family, 1-15
- file handle, 6-4
- file hints, 6-108
- file pointer, 6-4
- filetype, 6-4
- G
- ge64m, 1-10, 5-13
- get and set file attributes, A-69
- get and set the client API library configuration, A-76
- get file system information for a COS, A-75
- gt64m, 1-10, 5-14
- H
- high32m, 1-9, 5-15
- HPSS
  - overview, 1-1

hpss\_Access, 1-2, 2-3  
HPSS\_BUSY\_DELAY, 1-5, 1-14  
HPSS\_BUSY\_RETRIES, 1-5  
hpss\_Chacct, 1-2, 2-5  
hpss\_Chdir, 1-2, 2-6  
hpss\_Chmod, 1-2, 2-7, 6-108  
hpss\_Chown, 1-2, 2-9  
hpss\_Chroot, 1-2, 2-11  
hpss\_ClientAPIInit, 1-2  
hpss\_ClientAPIReset, 1-2, 2-13  
hpss\_Close, 1-2, 2-14, A-6, A-8, A-73  
hpss\_Closedir, 1-2, 2-15, A-69  
hpss\_cos\_hints\_t, 2-130  
hpss\_cos\_priorities\_t, 2-132  
hpss\_Create, 1-2, 2-16  
HPSS\_DEBUG, 1-4  
HPSS\_DEBUGPATH, 1-4  
hpss\_DeleteACL, 1-2  
hpss\_DeleteAcl, 2-18  
HPSS\_DESC\_NAME, 1-4  
HPSS\_DMAP\_WRITE\_UPDATES, 1-5, 1-8  
hpss\_Fclear, 1-2, 2-20  
hpss\_FclearOffset, 2-21  
hpss\_fileattr\_t, 2-136  
hpss\_FileGetAttributes, 1-2, 2-22, A-70, A-71  
hpss\_FileGetXAttributes, 1-2, 2-23  
hpss\_FileSetAttributes, 1-2, 2-25, A-71  
hpss\_FilesetCreate, 1-2, 2-28  
hpss\_FilesetDelete, 2-30  
hpss\_FilesetDeletes, 1-2  
hpss\_FilesetGetAttributes, 1-2, 2-31  
hpss\_FilesetListAll, 1-2, 2-33  
hpss\_FilesetSetAttributes, 1-2, 2-35  
hpss\_Fpreallocate, 1-2  
hpss\_Fstat, 1-2, 2-38  
hpss\_Ftruncate, 1-2, 2-39  
hpss\_GetAcct, 1-2, 2-40  
hpss\_GetACL, 1-2, 2-41  
hpss\_GetBFSSStats, 1-3, 2-43

hpss\_GetConfiguration, 1-2, 2-44, A-78  
hpss\_Getcwd, 1-2, 2-45  
hpss\_GetListAttrs, 1-2, 2-46, A-67  
HPSS\_HOSTNAME, 1-4, 1-8  
hpss\_JunctionCreate, 1-2, 2-48  
hpss\_JunctionDelete, 1-2, 2-50  
HPSS\_KTAB\_PATH, 1-4  
hpss\_Link, 1-2, 2-51, A-74  
hpss\_LoadDefaultThreadState, 1-2, 2-54  
hpss\_LoadThreadState, 1-2, 2-53  
HPSS\_LS\_NAME, 1-4, 1-8, 1-14  
hpss\_Lseek, 1-2, 2-55  
hpss\_LseekOffset, 1-2  
hpss\_Lstat, 1-2, 2-57, A-72  
HPSS\_MAX\_CONN, 1-4  
hpss\_Migrate, 1-2, 2-59  
hpss\_Mkdir, 1-2, 2-61  
HPSS\_NUMRETRIES, 1-5  
hpss\_Open, 1-2, 1-15, 2-63, A-4, A-7, A-73  
hpss\_OpenBitfile, 1-2, 2-66  
hpss\_Opendir, 1-2, 2-68, A-67  
HPSS\_PRINCIPAL, 1-4  
hpss\_Purge, 1-2, 2-70  
hpss\_PurgeLock, 1-2, 2-72  
hpss\_PurgeLoginContext, 1-2, 2-73  
hpss\_PVRetrievals, 1-7, 2-129  
hpss\_Read, 1-2, 2-74, A-6, A-8  
hpss\_ReadAttrs, 1-2, 2-76  
hpss\_Readdir, 1-2, 2-78, A-67  
hpss\_Readlink, 1-2, 2-80, A-68  
hpss\_ReadList, 1-2, 2-82, 4-1, A-1, A-37, B-1  
    example, A-37  
HPSS\_REGISTRY\_SITE\_NAME, 1-5  
hpss\_Rename, 1-2, 2-85  
hpss\_ReopenBitfile, 1-2, 2-87  
HPSS\_RETRY\_STAGE\_INP, 1-5  
HPSS\_REUSE\_CONNECTIONS, 1-5  
hpss\_Rewinddir, 1-2, 2-89  
hpss\_Rmdir, 1-2, 2-90

HPSS\_SERVER\_NAME, 1-4, 1-8  
 hpss\_SetAcct, 1-2, 2-94  
 hpss\_SetACL, 1-2, 2-92  
 hpss\_SetBFSSStats, 1-3, 2-95  
 hpss\_SetConfiguration, 1-2, 2-96, A-78  
 hpss\_SetCOSByHints, 2-97  
 hpss\_SetFileOffset, 1-2, 2-99  
 hpss\_SetLoginContext, 1-2, 2-101  
 hpss\_Stage, 1-2, 2-102  
 hpss\_StageCallBack, 2-104  
 hpss\_Stat, 1-2, 2-106, A-72  
 hpss\_Statfs, 1-2, 2-108, A-75  
 hpss\_Symlink, 1-2, 2-109, A-74  
 hpss\_ThreadCleanUp, 1-2, 2-1, 2-111  
 HPSS\_TOTAL\_DELAY, 1-5  
 HPSS\_TRANSFER\_TYPE, 1-4, 1-8  
 hpss\_Truncate, 1-2, 2-112  
 hpss\_Umask, 1-2, 2-114  
 hpss\_Unlink, 1-2, 2-115, A-73, A-75  
 hpss\_UpdateACL, 1-2, 2-117  
 HPSS\_USE\_PORT\_RANGE, 1-5  
 hpss\_Utime, 1-2, 2-119  
 hpss\_Write, 1-2, 2-121, A-2, A-5, A-73  
 hpss\_WriteList ,  
     example, A-9  
 hpss\_WriteList, 1-2, 2-123, 4-1, A-1, A-9, B-1  
 hpss\_XLoadThreadState, 2-125  
 hpssuser, 1-16  
**I**  
 I/O Descriptor  
     components, 3-1  
     purpose, 3-1  
 I/O Reply  
     components, 3-2  
     purpose, 3-1  
 individual file pointer, 6-4  
 initiator\_ipaddr\_t, 4-52  
 initiator\_ipi3addr\_t, 4-53  
 initiator\_msg\_t, 4-49

initiator\_shmaddr\_t, 4-54

IOD\_t, 3-3

IOR\_t, 3-17

### IPI-3 Data Transfer Functions

ipi3\_data3\_cancel, 4-9

ipi3\_data3\_close, 4-3

ipi3\_data3\_complete, 4-8

ipi3\_data3\_open, 4-2

ipi3\_data3\_read, 4-4

ipi3\_data3\_write, 4-6

### IPI-3 Data Transfer Library Data Definitions

IPI-3 interface address, 4-10

ipi3\_data3\_cancel, 4-9

ipi3\_data3\_close, 4-3

ipi3\_data3\_complete, 4-8

ipi3\_data3\_open, 4-2, 4-10

ipi3\_data3\_read, 4-4, 4-6, 4-8

ipi3\_data3\_write, 4-8

IPI3\_INTERFACE\_STRUCT, 4-10

## K

kdestroy, 1-16

keytab file, 1-14

kinit, 1-16

klist, 1-16

## L

le64m, 1-9

### Libraries

libdce.a, 1-3, 1-8, 1-13

libEncClient.a, 1-13

libEncina.a, 1-3, 1-8, 1-13

libhpss.a, 1-3, 1-8, 1-13

libhpss\_ipi.a, 1-3, 1-13

libhpssipi3.a, 1-13

libipi3.a, 1-13

libmpi.a, 1-13

libmpioapi.a, 1-12

link operations, A-71

low32m, 1-9

lshpss, 1-15

lt64m, 1-9, 5-18

## M

### makefile

sample, A-1, B-1

### Math Library Data Definitions

u\_signed64, 5-29

unsigned32, 5-29

mod2x64m, 1-10, 5-20

mod64m, 1-10, 5-19

### Mover Protocol Data Structures

completion message, 4-51

Initiator Message, 4-49

IPI-3 address, 4-53

shared memory address, 4-54

TCP/IP address, 4-52

### Mover Socket Functions

mover\_socket\_get\_buffer, 4-17

mover\_socket\_get\_buffer\_timeout, 4-19

mover\_socket\_recv\_data, 4-21

mover\_socket\_recv\_data\_timeout, 4-23

mover\_socket\_send\_buffer, 4-11

mover\_socket\_send\_buffer\_timeout, 4-13

mover\_socket\_send\_buffer\_timeout\_size, 4-15

mover\_socket\_send\_requested\_data, 4-25

mover\_socket\_send\_requested\_data\_timeout, 4-27

mover\_socket\_send\_requested\_data\_timeout\_size, 4-29

mover\_socket\_waitfor\_data, 4-31

mover\_socket\_waitfor\_data\_timeout, 4-33

mover\_waitfor\_requests, 4-35

mover\_waitfor\_requests\_timeout, 4-37

mvrprot\_recv\_compmsg, 4-47

mvrprot\_recv\_ipaddr, 4-41

mvrprot\_recv\_ipi3addr, 4-43

mvrprot\_recv\_shmaddr, 4-45

mvrprot\_send\_compmsg, 4-48

mvrprot\_send\_ipaddr, 4-42

mvrprot\_send\_ipi3addr, 4-44

mvrprot\_send\_shmaddr, 4-46

mover to mover protocol, A-9, A-37

mover\_socket\_get\_buffer, 4-17

mover\_socket\_get\_buffer\_timeout, 4-19

mover\_socket\_recv\_data, 4-21

mover\_socket\_recv\_data\_timeout, 4-23  
mover\_socket\_send\_buffer, 4-11  
mover\_socket\_send\_buffer\_timeout, 4-13  
mover\_socket\_send\_buffer\_timeout\_size, 4-15  
mover\_socket\_send\_requested\_data, 4-25  
mover\_socket\_send\_requested\_data\_timeout, 4-27  
mover\_socket\_send\_requested\_data\_timeout\_size, 4-29  
mover\_socket\_waitfor\_data, 4-31  
mover\_socket\_waitfor\_data\_timeout, 4-33  
mover\_waitfor\_requests, 4-35  
mover\_waitfor\_requests\_timeout, 4-37  
MPI\_File\_call\_errhandler, 1-12, 6-107  
MPI\_File\_close, 1-11, 6-3, 6-4, 6-8  
MPI\_File\_create\_errhandler, 1-12, 6-100, 6-104  
MPI\_File\_delete, 1-11, 6-9, 6-100, 6-109  
MPI\_File\_get\_amode, 1-11, 6-17  
MPI\_File\_get\_atomicity, 1-12, 6-98  
MPI\_File\_get\_byte\_offset, 1-11  
MPI\_File\_get\_errhandler, 1-12, 6-106  
MPI\_File\_get\_group, 1-11, 6-16  
MPI\_File\_get\_info, 1-11, 6-19, 6-108, 6-109  
MPI\_File\_get\_nthkey, 6-116  
MPI\_File\_get\_position, 1-11  
MPI\_File\_get\_position\_shared, 1-12, 6-68  
MPI\_File\_get\_size, 1-11, 6-15  
MPI\_File\_get\_type\_extent, 1-12, 6-91, 6-93  
MPI\_File\_get\_valuelen, 6-114  
MPI\_File\_get\_view, 1-11, 6-22  
MPI\_File\_iread, 1-11  
MPI\_File\_iread\_at, 1-11  
MPI\_File\_iread\_shared, 1-12, 6-58  
MPI\_File\_iwrite, 1-11  
MPI\_File\_iwrite\_at, 1-11  
MPI\_File\_iwrite\_shared, 1-12  
MPI\_File\_open, 1-11, 6-1, 6-2, 6-3, 6-4, 6-5, 6-100, 6-109  
MPI\_File\_preallocate, 1-11, 6-3  
MPI\_File\_read\_all\_begin, 1-11, 6-76  
MPI\_File\_read\_all\_end, 1-11, 6-78  
MPI\_File\_read\_at\_all\_begin, 1-11, 6-69

MPI\_File\_read\_at\_all\_end, 1-11, 6-71  
MPI\_File\_read\_ordered, 1-12, 6-62  
MPI\_File\_read\_ordered\_begin, 1-12, 6-83  
MPI\_File\_read\_ordered\_end, 1-12, 6-85  
MPI\_File\_read\_shared, 1-12  
MPI\_File\_seek, 1-11  
MPI\_File\_seek\_shared, 1-12, 6-3, 6-66  
MPI\_File\_set\_atomicity, 1-12, 6-3, 6-97  
MPI\_File\_set\_errhandler, 1-12, 6-100, 6-105  
MPI\_File\_set\_info, 1-11, 6-18, 6-109  
MPI\_File\_set\_size, 1-11, 6-3, 6-11  
MPI\_File\_set\_view, 1-11, 6-1, 6-3, 6-20, 6-90, 6-100, 6-109  
MPI\_File\_sync 6-99  
MPI\_File\_sync, 1-12, 6-3  
MPI\_File\_write, 1-11  
MPI\_File\_write\_all, 1-11  
MPI\_File\_write\_all\_begin, 1-11, 6-79  
MPI\_File\_Write\_all\_end, 1-11  
MPI\_File\_write\_all\_end, 6-81  
MPI\_File\_write\_at, 1-11  
MPI\_File\_write\_at\_all, 1-11  
MPI\_File\_write\_at\_all\_begin, 1-11, 6-72  
MPI\_File\_write\_at\_all\_end, 1-11, 6-74  
MPI\_File\_write\_ordered, 1-12, 6-64  
MPI\_File\_write\_ordered\_begin, 1-12, 6-86  
MPI\_File\_write\_ordered\_end, 1-12, 6-88  
MPI\_File\_write\_shared, 1-12, 6-56  
MPI\_Info, 6-109  
MPI\_Info\_create, 1-12, 6-109, 6-110  
MPI\_Info\_delete, 1-12, 6-112  
MPI\_Info\_dup, 1-12, 6-117  
MPI\_Info\_free, 1-12, 6-109, 6-118  
MPI\_Info\_get, 1-12, 6-109, 6-113  
MPI\_Info\_get\_nkeys, 1-12, 6-115  
MPI\_Info\_get\_nthkey, 1-12  
MPI\_Info\_get\_valuelen, 1-12  
MPI\_Info\_set, 1-12, 6-109, 6-111  
MPI\_Offset 6-119  
MPI\_Offset, 6-119

- MPI\_Register\_datarep, 1-12, 6-94
- MPI\_Test, 6-1, 6-103
- MPI\_Wait, 6-1, 6-103
- MPI-IO access, 1-16
- MPI-IO API
  - collective functions, 6-1
  - Components, 1-11
  - Constraints, 1-12
  - Environment Variables, 1-13
  - Libraries, 1-12
  - noncollective functions, 6-1
  - Purpose, 1-11
- MPI-IO file, 6-4
- MPI-IO Standard Data Definitions
  - MPI\_Offset, 6-119
- MPIO\_DEBUG, 1-14
- MPIO\_KEYTAB\_PATH, 1-14
- MPIO\_LOGIN\_NAME, 1-14
- mul64m, 1-10, 5-21
- mvrprot\_recv\_compmsg, 4-47
- mvrprot\_recv\_ipaddr, 4-41
- mvrprot\_recv\_ipi3addr, 4-43
- mvrprot\_recv\_shmaddr, 4-45
- mvrprot\_send\_compmsg, 4-48
- mvrprot\_send\_ipaddr, 4-42
- mvrprot\_send\_ipi3addr, 4-44
- mvrprot\_send\_shmaddr, 4-46
- N
  - neq64m, 1-10
  - neqz64m, 1-10, 5-22
  - netopt\_FindEntry, 4-65
  - netopt\_GetWriteSize, 4-66
- Network Options Functions
  - netopt\_FindEntry, 4-65
  - netopt\_GetWriteSize, 4-66
- noncollective functions, 6-1
- Non-DCE Client API
  - components, 1-7
  - constraints, 1-7
  - environment variables (also see Environment Variables), 1-8

- libraries (also see Libraries), 1-8
- not64m, 1-10, 5-23
- ns\_ACLConfArray\_t, 2-158
- ns\_ACLEntry\_t, 2-158
- ns\_Attrs\_t, 2-137
- ns\_ObjHandle\_t, 2-144, 2-145

## O

- offset, 6-4
- or64m, 1-10, 5-24
- ow32m, 5-17

## P

### Parallel Data Transfer Data Definitions

- header, 4-64

### Parallel Data Transfer Functions

- pdata\_rcv\_hdr, 4-55
- pdata\_rcv\_hdr\_timeout, 4-56
- pdata\_send\_hdr, 4-57
- pdata\_send\_hdr\_and\_data, 4-59
- pdata\_send\_hdr\_and\_data\_timeout, 4-60
- pdata\_send\_hdr\_and\_data\_timeout\_size, 4-62
- pdata\_send\_hdr\_timeout, 4-58
- participating process, 6-1
- pdata\_hdr\_t, 4-64
- pdata\_rcv\_hdr, 4-55
- pdata\_rcv\_hdr\_timeout, 4-56
- pdata\_send\_hdr, 4-57
- pdata\_send\_hdr\_and\_data, 4-59
- pdata\_send\_hdr\_and\_data\_timeout, 4-60
- pdata\_send\_hdr\_and\_data\_timeout\_size, 4-62
- pdata\_send\_hdr\_timeout, 4-58
- pv\_list\_element\_t, 2-153
- pv\_list\_t, 2-153

## R

- rgy\_edit, 1-14

## S

- shared file pointer, 6-4
- shl64m, 1-10, 5-25
- shr64m, 1-10, 5-26
- size, 6-4

- srcsinkdesc\_t 3-6
- srcsinkreply\_t, 3-21
- storage class, 1-15
- storage hierarchy, 1-15
- Structures
  - account record, 2-154
  - address, 3-8
  - API configuration, 2-155
  - bitfile metadata attributes, 2-147
  - bitfile owner record, 2-150
  - bitfile server statistics 2-154
  - bitfile server storage class attributes 2-151
  - bitfile server virtual volume attributes 2-152
  - bitfile volatile and extended metadata attributes, 2-146
  - bitfile volatile and metadata attributes, 2-146
  - COS priorities, 2-132
  - file attribute, 2-136
  - file creation hint, 2-130
  - I/O descriptor, 3-3
  - I/O reply, 3-17
  - NS ACL conformant array, 2-158
  - NS ACL entry, 2-158
  - NS attribute, 2-137
  - NS directory entry, 2-145
  - NS object handle, 2-144
  - source/sink descriptor, 3-6
  - source/sink reply, 3-21
  - storage server physical volume attributes 2-153
  - storage server physical volume attributes conformant array 2-153
- sub64\_3m, 5-28
- sub64m, 1-10, 5-27
- U
  - u\_signed64, 5-29
  - unsigned32, 5-29
- User IDs, 1-16
- V
  - view, 6-4